cost. Factor all of this in when you do your research on these five types of connections, and choose the one that is best for you.

CHAPTER – 4

File

File is nothing but an Electronic document. The contents can be ordinary Text or it can be an executable program. Each file is given a file name to identify it. The File name is in the form

File Name . Extension

Filename can consist of Alphabets or combinations of alphabets, numerals and special characters. Extension indicates the type of file.

Example : XY.Doc

Here File name is XY

Extension name is .DOC which indicates Document file.

Folder

Folder contains a group of files. Folder is otherwise called as directory. Folder may have a set of files under it. It may have other folders under it also. This files and folders can be arranged in hierarchical manner or a tree like structure.

File organization

The arrangement of records in a file is known as File Organisation. File Organisation deals with the arrangement of data items in the secondary storage devices like magnetic disk. That is, the file organisation deals with how the logical tuples (rows) of tables (relations) are organised on the physical storage medium.

For organising records efficiently in the form of a computer file, following three things are important:

(a) A logical method should be selected to organise records in a file.

(b) File structure should be so designed that it would allow quick access to needed data items.

(c) Means of adding or deleting data items or records from files must be present.

Depending on the above considerations, a file may be organised as:

(a) Sequential file

(b) Direct or random access file

(c) Indexed-sequential file

Sequential file

A sequential file is a file in which the records are stored in some order, say the student file contains records of students in the ascending order of roll number of students. It is not necessary that all the records of a sequential file should be in physical adjacent positions. On a magnetic tape, the records are written one after the other along the length of the tape. In case of magnetic disks, the records of a sequential file may not be in contiguous locations. The sequential order may be given with the help of pointers on each record.

Sequential files are preferable when they are to be stored on sequential access devices such as Magnetic tapes.

1001 Rec.1	1002 Rec.2	1003 Rec.3	1004 Rec.4	1005 Rec.5	1006 Rec.6

A sequential file on tape

1001 Rec.1	1002 Rec.2		1003 Rec.3	1004 Rec.4

1015 Rec.1	1017 Rec.2

A sequential file on disk

The main advantages of sequential file organisation are:

(a) File design is simple.

(b) Location or records requires only the record key.

(c) When the activity rate is high, simplicity of the accessing method makes processing efficient.

(d) Low-cost file media such as magnetic tapes can be used for storing data.

The main drawbacks of sequential file organisation are.

(a) Updating requires that all transaction records are sorted in the record key sequence.

(b) A new master file, physically separate and exclusive, is always created as a result of sequential updating.

(c) Addition and deletion of records is not simple.

Direct Access File.

A sequential file is not suitable for on-line enquiry. Suppose a customer at a bank wishes to know the balance amount in his savings account. If the customer file Ls organised sequentially, the record of this customer has to be obtained by searching sequentially from the beginning. There is no way of picking out the particular record wit'hout traversing the file from the beginning and this mpy take a long time. Hence, in such situations, random access or direct access file organisation provides a means of accessing records speedily.

In random access or direct access method of file organisation, each record has its own address on the file. With the help of this physical address, the record can be directly accessed for reading or writing. The records need not be in any sequence within the file and also need not be in adjacent locations on the storage medium. Such a file cannot be created on a magnetic tape medium. Random (or direct) files are created only on magnetic disks. Since every record can be independently accessed, every transaction can be manipulated individually.

Random access file organisation is best suited for on-line processing systems where current information is the one that is always required.

The advantages of Direct Access file organisation are:

(a) Immediate access to records is possible.

(b) Up-to-date information will always be available on the file.

(c) Several files can be simultaneously updated.

(d) Addition and deletion or records is not very complex.

(e) No new master file is created for updating a random access file.

The disadvantages of Direct Access file organisation are:

(a) Less efficient in the use of storage space.

(b) Uses a relatively expensive medium.

(c) Not well suited for batch processing.

(d) Data security is less due to direct access facility.

<u>Indexed Sequential File</u>

Some files may be required to support both batch processing and on-line processing. For example, an inventory or stock file may be updated periodically by batch processing and at the same time may have to provide current information about stock availability on-line. They can be thus, organised as indexed sequential files. Indexed Sequential file combines the advantages of sequential and direct access file organisations.

An indexed sequential file is basically a sequential file organised serially on key fields. In addition, an index is maintained which speedsúp the access of isolated records. Just as you may se indexes to locate information in book, similarly an index is provided for the file. The file is divided into a number of blocks and the highest key in each block is indexed.

Key	**Starting address of the block**
125	12
860	19
1420	24
1600	42
1829	49
2225	159
2890	165
3200	807

Indexed sequential files are also known as Indexed Sequential Access Method (ISAM) files.

Within each block, the record is searched sequentially. This method is much faster than searching the entire file sequentially. It is also possible to have more than one level of indexing to make the search process faster.

The main advantage of indexed sequential file organisation is that it is suitable for both sequential and on-line or direct access processing.

The main disadvantages of this organisation are:

(a) Less efficient in the use of storage space.

(b) Additions and deletions of records are more complex as they effect both the index and the record number in the file.

Deciding on a File Organisation

The major factors to be considered while deciding which file organisation should be chosen are the following:

(a) Percentage of actual records processed in a day. If large number of records are accessed at a time, direct access file organisation should be used. If very few records are accessed, sequential file organisation will be more suitable and cheaper.

(b) Files that are frequently updated must be stored on a direct access storage device, such as disk.

(c) Selection of file organisation also depend on the mode of processing i.e. where the system requires an online processing or batch processing.

DATA PROCESSING

Data Processing means, processing the input data to produce some meaningful and purposeful information. Computer has the capability of processing high-volume of data in less time with higher accuracy. Hence the data processing performed by computer is sometimes called Electronic Data Processing (EDP). Data processing involves 5 distinct steps.

- Data capturing
- Data validation
- Processing / Execution
- Data storage
- Data Retrieval/Out generation

Data capturing encompasses the activities of inputing data to the computer. Before giving input to the system the required data are to be first identified and put in the defined format called source data layout. The aim of this layout is to have faster data entry. To reduce the volume of data and also have better organisation and easy access to data those can suitably be coded. After data are ready they can be entered to the computer through keyboard. This is sometimes called data capture through intelligent terminal. The other form of data capture is through scanners or optical devices. In this type of data capturing data are not entered rather data are captured from the source document or paper as it is Photographs, fingerprints, signatures, objective multiple type answers in answer papers etc. are captured through this method. Another form of data capturing is through some interfacing devices from where data can be transferred directly to the computer. Example of this is Electronic cash Registers used in shops and cash counters.

After data capturing, the data is validated. Data validation involves checking of input data to fit to requirements or specifications. For example price of a book can be numeric only. If by mistake Alphabetic data are entered then it is checked and error is shown to revalidate the data. This prevents unwanted and unspecified data to enter into the system and causing errors.

The valid input data are stored in file or database and processed as per the instructions. The instructions are put in programs or software. This software or program, when executed does the processing of input data and produces the output. Again outputs are stored in files in memory.

After processing of data, the outputs are produced. The outputs may be formed in different ways depending on the requirements and specifications. The same set of data can be printed in tabular form or in form of graphs. There are variety of ways for presenting data. The printed output is sometimes called hardcopy. There can be provision of answering to queries of user where the answer is displayed on the monitor screen itself. So it depends on the requirements of user.

An important step of Data Processing is maintaining Database. Database is nothing but collection of data which is controlled centrally with many provisions of data security. This is where, normally data are stored for reference. The input data and output data are stored in database. Even after processing of data and producing of data is over, database is maintained properly with safety for future needs and reference. The important tool of data processing is file. File is nothing but an electronic document where data can be stored. Depending on the type of file structure and organisation the data access speed varies.

CHAPTER - 5

Algorithm & Flowchart

Algorithm is defined as the step by step solution of problem in user's language.

It is considered as an effective procedure for solving a problem in finite number of steps. The characteristics of Algorithm are

- Precise
- Unambiguous
- Finite termination
- Unique solution

Once algorithm is written, it can be coded into a program using any programming language. Algorithm uses 3 different constructs

- Sequence
- Branching or Decision making
- Repetition

Sequence says that instructions are to be executed in what order or sequence. Branching involves testing of condition and based on the outcome of the condition testing different instructions are executed. Repetition means one or more instructions shall be repeated for a number of times. This is otherwise called as loop. There are different types of loops such as

While- do , do-while , for

Example:

1. Algorithm to find out sum of two numbers to be taken as input.

```
Step-1  Read the 1st number  x
Step-2  Read the 2nd number y
Step-3  Sum=x+y
```

```
Step-4  Print Sum
```

This is an example where only sequence is exhibited

2. Algorithm to find out larger between numbers to be taken as input.

```
Step-1  Read the 1st number  x
Step-2  Read the 2nd number y
Step-3  If x > y
            Then Print x
        Else if  x< y
            Then  Print y
        Else Print " Both are Equal "
```

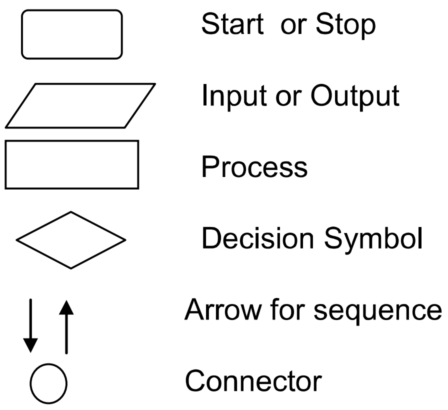
This is an example where Branching is exhibited

3. Algorithm to find out sum of first 10 natural numbers.

```
Step-1  i=1, Sum=0
Step-2  Repeat step 3 and 4 while i<>10
Step-3  Sum= Sum+i
Step-4  i=i+1
Step-5  Print Sum
```

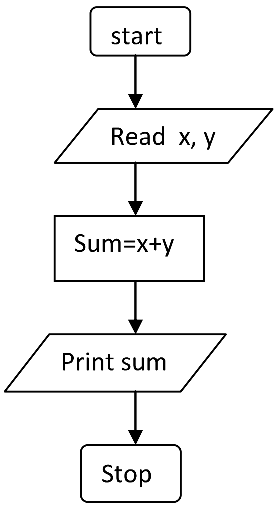
This is an example where Repetition is exhibited

Flowchart is a graphical or symbolic representation of the process of solution to a problem or algorithm. It helps to visualize the complex logic of the solution of the problem in a simplified manner through diagrammatic representation. Each step of the algorithm is presented using a symbol and a short description. The different symbols used for the flowchart are

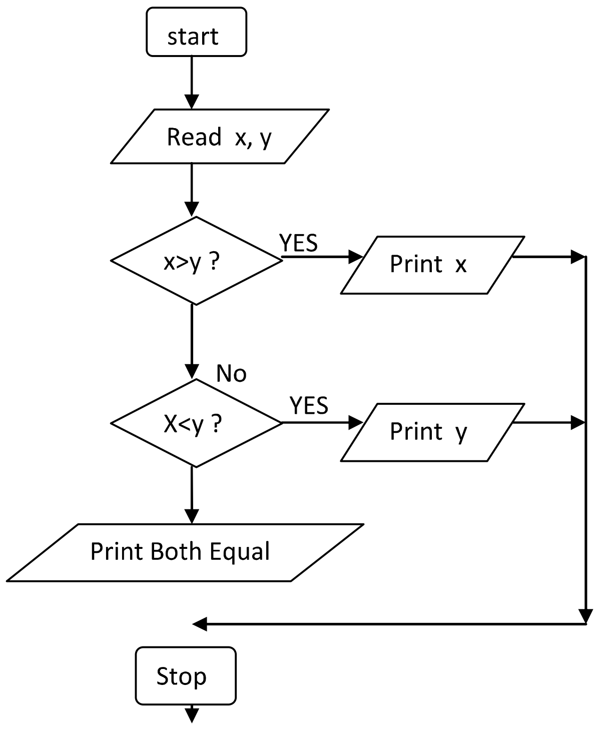


Example

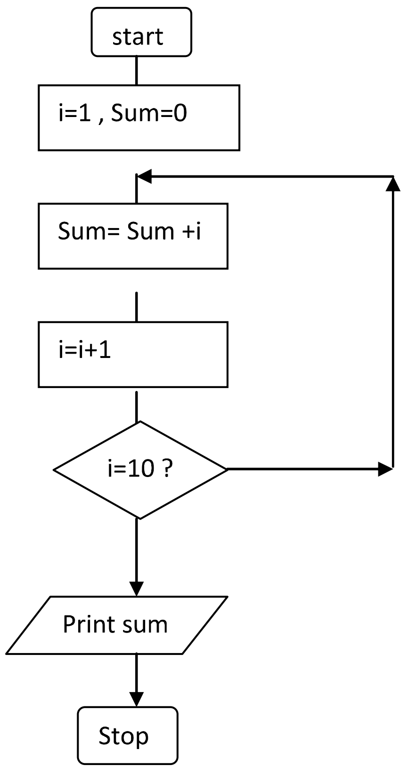
1. Flowchart to find out sum of two numbers to be taken as input



2. Flowchart to find out larger between two numbers to be taken as input



3. Flowchart to find out sum of first 10 natural numbers



Pseuodocode

It is a concise description algorithm in English language that uses programming language constructs. It contains outlines of the program that can be easily converted to program. It focuses on the logic of the algorithm without giving stress on the syntax of programming language. This is meant for understanding the logic of the program easily. Flowchart can be considered as an alternative to pseudocode. Several constructs/key words of programming language can be used in the algorithm to write the pseudocode. Some of them are

If ... Endif

Do while ... enddo

While ... endwhile

Repeat ... until

For ... endfor

Case .... endcase

Call

Return

Programming Language

Programming language is a tool to express the logic or instructions for understanding of the computer. Any programming language has two components:

- Syntax
- Semantics

Syntax refers to the rules to be followed for writing valid program statements. Compiler can detect errors in syntax while compiling the program.

Semantics is associated with logic of the program. Compiler can not detect the semantic error. The user of programmer can diagnose semantic error.

There are good number of High level languages, each meant for specific area of data processing. Commonly known languages are BASIC,FORTRAN, COBOL, Pascal, C, C++ etc. While FORTRAN is good for Numerical and scientific calculation, COBOL is good for Business applications involving large amount of data handling.

Generations of Programming Language

The Programming languages can be classified into 4 generations

1st Generation:	Machine Language
2nd Generation:	Assembly Language
3rd Generation:	High Level Language
4th Generation:	Very High Level Language

Machine Level language contains instructions in binary form i.e. in 0s and 1s. Thus writing instruction was very difficult and needs heavy expertise. This was used in early days computers.

Assembly level language instructions were written using symbolic codes known as mnemonics. In comparison to Machine language it is relatively easier to write program, but still it requires lot of expertise. A translator called assembler is used to translate assembly language program to machine level language.

High level language contains instructions in English like words so that user will feel easier to formulate and write the logical statements of the program. Here the logic may spread over multiple statements as against a single statement in assembly language. It uses a translator called compiler for translation of High level language program to machine level language program. There are many High level languages used for programming such as BASIC, FORTRAN, COBOL, PASCAL, C, C++ etc.

Very High Level language other wise called as 4GL uses nonprocedural logical statements. A typical example of 4GL is the query language such as SQL.

Structured Programming Language

Structured Programming is also known as Modular Programming. In this type of programming technique, the program shall be broken into several modules. This helps in managing memory efficiently as the required module of the program will be loaded into the memory only and not the entire program. This will also enhance code reuse. Writing, understanding, debugging and modifying the individual module of the program is also easier.

CHAPTER – 6

The C Language

C is a professional programmer's language. It was designed to get in one's way as little as possible. Kernighan and Ritchie wrote the original language definition in their book, *The C Programming Language* (below), as part of their research at AT&T. Unix and C++ emerged from the same labs. For several years I used AT&T as my long distance carrier in appreciation of all that CS research, but hearing "thank you for using AT&T" for the millionth time has used up that good will.

The C Language

C is a professional programmer's language. It was designed to get in one's way as little as possible. Kernighan and Ritchie wrote the original language definition in their book, *The C Programming Language* (below), as part of their research at AT&T. Unix and C++ emerged from the same labs. For several years I used AT&T as my long distance carrier in appreciation of all that CS research, but hearing "thank you for using AT&T" for the millionth time has used up that good will.

Some languages are forgiving. The programmer needs only a basic sense of how things work. Errors in the code are flagged by the compile-time or run-time system, and the programmer can muddle through and eventually fix things up to work correctly. The C language is not like that.The C programming model is that the programmer knows exactly what they want to do and how to use the language constructs to achieve that goal. The language lets the expert programmer express what they want in the minimum time by staying out of their way. C is "simple" in that the number of components in the language is small-- If two language features accomplish more-or-less the same thing, C will include only one. C's syntax is terse and the language does not restrict what is "allowed" -- the programmer can pretty much do whatever they want C's type system and error checks exist only at compile-time. The compiled code runs in a stripped down run-time model with no safety checks for bad type casts, bad array indices, or bad pointers. There is no garbage collector to manage memory. Instead the programmer manages heap memory manually. All this makes C fast but fragile.

Analysis -- Where C Fits

Because of the above features, C is hard for beginners. A feature can work fine in one context, but crash in another. The programmer needs to understand how the features work and use them correctly. On the other hand, the number of features is pretty small. Like most programmers, I have had some moments of real loathing for the C language. Itcan be irritatingly obedient -- you type something incorrectly, and it has a way of compiling fine and just doing something you don't expect at run-time. However, as I have become a more experienced C programmer, I have grown to appreciate C's straight-to-the point style. I have learned not to fall into its little traps, and I appreciate its simplicity.

Perhaps the best advice is just to be careful. Don't type things in you don't understand. Debugging takes too much time. Have a mental picture (or a real drawing) of how your C code is using memory. That's good advice in any language, but in C it's critical.

Perl and Java are more "portable" than C (you can run them on different computers without a recompile). Java and C++ are more structured than C. Structure is useful for large projects. C works best for small projects where performance is important and the progammers have the time and skill to make it work in C. In any case, C is a very popular and influential language. This is mainly because of C's clean (if minimal) style, it's lack of annoying or regrettable constructs, and the relative ease of writing a C compiler.

Other Resources

• *The C Programming Language*, 2nd ed., by Kernighan and Ritchie. The thin book which for years was the bible for all C programmers. Written by the original designers of the language. The explanations are pretty short, so this book is better as a reference than for beginners.

• http://cslibrary.stanford.edu/102/ Pointers and Memory -- Much more detail about local memory, pointers, reference parameters, and heap memory than in this article, and memory is really the hardest part of C and C++.

• http://cslibrary.stanford.edu//103/ Linked List Basics -- Once you understand the basics of pointers and C, these problems are a good way to get more practice.

Basic Types and Operators

C provides a standard, minimal set of basic data types. Sometimes these are called “primitive" types. More complex data structures can be built up from these basic types.

Integer Types

The "integral" types in C form a family of integer types. They all behave like integers and can be mixed together and used in similar ways. The differences are due to the different number of bits ("widths") used to implement each type -- the wider types can store a greater ranges of values.

char ASCII character -- at least 8 bits. Pronounced "car". As a practical matter char is basically always a byte which is 8 bits which is enough to store a single ASCII character. 8 bits provides a signed range of -128..127 or an unsigned range is 0..255. char is also required to be the "smallest addressable unit" for the machine -- each byte in memory has its own address.

short Small integer -- at least 16 bits which provides a signed range of -32768..32767. Typical size is 16 bits. Not used so much.

int Default integer -- at least 16 bits, with 32 bits being typical. Defined to be the "most comfortable" size for the computer. If you do not really care about the range for an integer variable, declare it int since that is likely to be an appropriate size (16 or 32 bit) which works well for that machine.

Long Large integer -- at least 32 bits. Typical size is 32 bits which gives a signed range of about -2 billion..+2 billion. Some compilers support "long long" for 64 bit ints.

The integer types can be preceded by the qualifier unsigned which disallows representing negative numbers, but doubles the largest positive number representable. For example, a 16 bit implementation of short can store numbers in the range -32768..32767, while unsigned short can store 0..65535. You can think of pointers as being a form of unsigned long on a machine with 4 byte pointers. In my opinion, it's best to avoid using unsigned unless you really need to. It tends to cause more misunderstandings and problems than it is worth.

Extra: Portability Problems

Instead of defining the exact sizes of the integer types, C defines lower bounds. This makes it easier to implement C compilers on a wide range of hardware. Unfortunately it occasionally leads to bugs where a program runs differently on a 16-bit-int machine than it runs on a 32-bit-int machine. In particular, if you are designing a function that will be implemented on several different machines, it is a good idea to use typedefs to set up types like Int32 for 32 bit int and Int16 for 16 bit int. That way you can prototype a function Foo(Int32) and be confident that the typedefs for each machine will be set so that the function really takes exactly a 32 bit int. That way the code will behave the same on all the different machines.

char Constants

A char constant is written with single quotes (') like 'A' or 'z'. The char constant 'A' is really just a synonym for the ordinary integer value 65 which is the ASCII value for uppercase 'A'. There are special case char constants, such as '\t' for tab, for characters which are not convenient to type on a keyboard.

'A' uppercase 'A' character

'\n' newline character

'\t' tab character

'\0' the "null" character -- integer value 0 (different from the char digit '0')

'\012' the character with value 12 in octal, which is decimal 10 int Constants

Numbers in the source code such as 234 default to type int. They may be followed by an 'L' (upper or lower case) to designate that the constant should be a long such as 42L. An integer constant can be written with a leading 0x to indicate that it is expressed in hexadecimal -- 0x10 is way of expressing the number 16. Similarly, a constant may be written in octal by preceding it with "0" -- 012 is a way of expressing the number 10.

This page lists C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied.

Operator	**Description**	**Associativity**
() [] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (change *type*) Dereference Address Determine size in bytes	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right

== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Type Combination and Promotion

The integral types may be mixed together in arithmetic expressions since they are all basically just integers with variation in their width. For example, char and int can be combined in arithmetic expressions such as ('b' + 5). How does the compiler deal with the different widths present in such an expression? In such a case, the compiler "promotes" the smaller type (char) to be the same size as the larger type (int) before combining the values. Promotions are determined at compile time based purely on the types of the values in the expressions. Promotions do not lose information -- they always convert from a type to compatible, larger type to avoid losing information.

Pitfall -- int Overflow

A piece of code which tried to compute the number of bytes in a buffer with the expression (k * 1024) where k was an int representing the number of kilobytes wanted. Unfortunately this was on a machine where int happened to be 16 bits. Since k and 1024 were both int, there was no promotion. For values of k >= 32, the product was too big to fit in the 16 bit int resulting in an overflow. The compiler can do whatever it wants in overflow situations -- typically the high order bits just vanish. One way to fix the code was to rewrite it as (k * 1024L) -- the long constant forced the promotion of the int. This was not a fun bug to track down -- the expression sure looked reasonable in the source code. Only stepping past the key line in the debugger showed the overflow problem. "Professional Programmer's Language." This example also demonstrate s the way that C only promotes based on the types in an expression. The compiler does not consider the values 32 or 1024 to realize that the operation will overflow (in general, the values don't exist until run time anyway). The compiler just looks at the compile time types, int and int in this case, and thinks everything is fine.

Floating point Types

float Single precision floating point number typical size: 32 bits double Double precision floating point number typical size: 64 bits long double Possibly even bigger floating point number (somewhat obscure) Constants in the source code such as 3.14 default to type double unless the are suffixed with an 'f' (float) or 'l' (long double). Single precision equates to about 6 digits of precision and double is about 15 digits of precision. Most C programs use double for their computations. The main reason to use float is to save memory if many numbers need to be stored. The main thing to remember about floating point numbers is that they are inexact. For example, what is the value of the following double expression?

(1.0/3.0 + 1.0/3.0 + 1.0/3.0) // is this equal to 1.0 exactly?

The sum may or may not be 1.0 exactly, and it may vary from one type of machine to another. For this reason, you should never compare floating numbers to eachother for equality (==) -- use inequality (<) comparisons instead. Realize that a correct C program run on different computers may produce slightly different outputs in the rightmost digits of its floating point computations.

Comments

Comments in C are enclosed by slash/star pairs: /* .. comments .. */ which may cross multiple lines. C++ introduced a form of comment started by two slashes and extending to the end of the line: // comment until the line end

The // comment form is so handy that many C compilers now also support it, although it is not technically part of the C language.

Along with well-chosen function names, comments are an important part of well written code. Comments should not just repeat what the code says. Comments should describe what the code accomplishes which is much more interesting than a translation of what each statement does. Comments should also narrate what is tricky or non-obvious about a section of code.

Variables

As in most languages, a variable declaration reserves and names an area in memory at run time to hold a value of particular type. Syntactically, C puts the type first followed by the name of the variable. The following declares an int variable named "num" and the 2nd line stores the value 42 into num.

```
int num;

num = 42;

num 42
```

A variable corresponds to an area of memory which can store a value of the given type.

Making a drawing is an excellent way to think about the variables in a program. Draw each variable as box with the current value inside the box. This may seem like a "beginner" technique, but when I'm buried in some horribly complex programming problem, invariably resort to making a drawing to help think the problem through.

Variables, such as num, do not have their memory cleared or set in any way when they are allocated at run time. Variables start with random values, and it is up to the program to set them to something sensible before depending on their values.

Names in C are case sensitive so "x" and "X" refer to different variables. Names can contain digits and underscores (_), but may not begin with a digit. Multiple variables can be declared after the type by separating them with commas. C is a classical "compiletime" language -- the names of the variables, their types, and their implementations are all flushed out by the compiler at compile time (as opposed to figuring such details out at run time like an interpreter).

```
float x, y, z, X;
```

Assignment Operator =

The assignment operator is the single equals sign (=).

```
i = 6;

i = i + 1;
```

The assignment operator copies the value from its right hand side to the variable on its left hand side. The assignment also acts as an expression which returns the newly assigned value. Some programmers will use that feature to write things like the following.

```
y = (x = 2 * x); // double x, and also put x's new value in y
```

Truncation

The opposite of promotion, truncation moves a value from a type to a smaller type. In that case, the compiler just drops the extra bits. It may or may not generate a compile time warning of the loss of information. Assigning from an integer to a smaller integer (e.g.. long to int, or int to char) drops the most significant bits. Assigning from a floating point type to an integer drops the fractional part of the number.

```
char ch;

int i;

i = 321;

ch = i; // truncation of an int value to fit in a char

// ch is now 65
```

The assignment will drop the upper bits of the int 321. The lower 8 bits of the number 321 represents the number 65 (321 - 256). So the value of ch will be (char)65 which happens to be 'A'.

The assignment of a floating point type to an integer type will drop the fractional part of the number. The following code will set i to the value 3. This happens when assigning a floating point number to an integer or passing a floating point number to a function which takes an integer.

```
double pi;

int i;

pi = 3.14159;

i = pi; // truncation of a double to fit in an int

// i is now 3
```

Pitfall -- int vs. float Arithmetic

Here's an example of the sort of code where int vs. float arithmetic can cause problems. Suppose the following code is supposed to scale a homework score in the range 0..20 to be in the range 0..100.

```
{

int score;

...// suppose score gets set in the range 0..20 somehow

7

score = (score / 20) * 100; // NO -- score/20 truncates to 0

...
```

Unfortunately, score will almost always be set to 0 for this code because the integer division in the expression (score/20) will be 0 for every value of score less than 20.

The fix is to force the quotient to be computed as a floating point number...

```
score = ((double)score / 20) * 100; // OK -- floating point division from cast

score = (score / 20.0) * 100; // OK -- floating point division from 20.0
```

score = (int)(score / 20.0) * 100; // NO -- the (int) truncates the floating

// quotient back to 0

No Boolean -- Use int

C does not have a distinct boolean type-- int is used instead. The language treats integer

0 as false and all non-zero values as true. So the statement...

i = 0;

while (i - 10) {

...

will execute until the variable i takes on the value 10 at which time the expression (i - 10) will become false (i.e. 0). (we'll see the while() statement a bit later)

Mathematical Operators

C includes the usual binary and unary arithmetic operators. See the appendix for the table of precedence. Personally, I just use parenthesis liberally to avoid any bugs due to a misunderstanding of precedence. The operators are sensitive to the type of the operands.

So division (/) with two integer arguments will do integer division. If either argument is a float, it does floating point division. So (6/4) evaluates to 1 while (6/4.0)

evaluates to 1.5 -- the 6 is promoted to 6.0 before the division.

+ Addition

- Subtraction

/ Division

* Multiplication

% Remainder (mod)

Unary Increment Operators: ++ --

The unary ++ and -- operators increment or decrement the value in a variable. There are "pre" and "post" variants for both operators which do slightly different things (explained below)

var++ increment "post" variant

++*var* increment "pre" variant

8

var-- decrement "post" variant

--*var* decrement "pre" variant

int i = 42;

i++; // increment on i

// i is now 43

i--; // decrement on i

// i is now 42

Pre and Post Variations

The Pre/Post variation has to do with nesting a variable with the increment or decrement operator inside an expression -- should the entire expression represent the value of the variable before or after the change? I never use the operators in this way (see below), but an example looks like...

int i = 42;

int j;

j = (i++ + 10);

// i is now 43

// j is now 52 (NOT 53)

j = (++i + 10)

// i is now 44

// j is now 54

Relational Operators

These operate on integer or floating point values and return a 0 or 1 boolean value.

== Equal

!= Not Equal

> Greater Than

< Less Than

>= Greater or Equal

<= Less or Equal

To see if x equals three, write something like:

if (x == 3) ...

Pitfall = ==

An absolutely classic pitfall is to write assignment (=) when you mean comparison (==).

This would not be such a problem, except the incorrect assignment version compiles fine because the compiler assumes you mean to use the value returned by the assignment. This is rarely what you want

if (x = 3) ...

This does not test if x is 3. This sets x to the value 3, and then returns the 3 to the if for testing. 3 is not 0, so it counts as "true" every time. This is probably the single most common error made by beginning C programmers. The problem is that the compiler is no help -- it thinks both forms are fine, so the only defense is extreme vigilance when coding. Or write "= □□==" in big letters on the back of your hand before coding. This mistake is an absolute classic and it's a bear to debug. Watch Out! And need I say:

"Professional Programmer's Language."

Logical Operators

The value 0 is false, anything else is true. The operators evaluate left to right and stop as soon as the truth or falsity of the expression can be deduced. (Such operators are called "short circuiting") In ANSI C, these are furthermore guaranteed to use 1 to represent true, and not just some random non-zero bit pattern. However, there are many C programs out there which use values other than 1 for true (non-zero pointers for example), so when programming, do not assume that a true boolean is necessarily 1 exactly.

! Boolean not (unary)

&& Boolean and

|| Boolean or

Bitwise Operators

C includes operators to manipulate memory at the bit level. This is useful for writing low level hardware or operating system code where the ordinary abstractions of numbers, characters, pointers, etc... are insufficient -- an increasingly rare need. Bit manipulation code tends to be less "portable". Code is "portable" if with no programmer intervention it compiles and runs correctly on different types of computers. The bitwise operations are typically used with unsigned types. In particular, the shift operations are guaranteed to shift 0 bits into the newly vacated positions when used on unsigned values.

~ Bitwise Negation (unary) – flip 0 to 1 and 1 to 0 throughout

& Bitwise And

| Bitwise Or

^ Bitwise Exclusive Or

>> Right Shift by right hand side (RHS) (divide by power of 2)

<< Left Shift by RHS (multiply by power of 2)

Do not confuse the Bitwise operators with the logical operators. The bitwise connectives are one character wide (&, |) while the boolean connectives are two characters wide (&&, ||). The bitwise operators have higher precedence than the boolean operators. The compiler will never help you out with a type error if you use &

when you meant &&. As far as the type checker is concerned, they are identical-- they both take and produce integers since there is no distinct boolean type.

Other Assignment Operators

In addition to the plain = operator, C includes many shorthand operators which represents variations on the basic =. For example "+=" adds the right hand side to the left hand side.

x = x + 10; can be reduced to x += 10;. This is most useful if x is a long expression such as the following, and in some cases it may run a little faster. person->relatives.mom.numChildren += 2; // increase children by 2

Here's the list of assignment shorthand operators...

+=, -= Increment or decrement by RHS

*=, /= Multiply or divide by RHS

%= Mod by RHS

>>= Bitwise right shift by RHS (divide by power of 2)

<<= Bitwise left shift RHS (multiply by power of 2)

&=, |=, ^= Bitwise and, or, xor by RHS

Control Structures

C uses curly braces ({}) to group multiple statements together. The statements execute in order. Some languages let you declare variables on any line (C++). Other languages insist that variables are declared only at the beginning of functions (Pascal). C takes the middle road -- variables may be declared within the body of a function, but they must follow a '{'. More modern languages like Java and C++ allow you to declare variables on any line, which is handy.

If Statement

Both an if and an if-else are available in C. The <*expression*> can be any valid expression. The parentheses around the expression are required, even if it is just a single variable.

if (<expression>) <statement> // simple form with no {}'s or else clause if (<expression>) { // simple form with {}'s to group statements

<statement>

<statement>

}

if (<expression>) { // full then/else form

<statement>

}

else {

<statement>

}

Conditional Expression -or- The Ternary Operator

The conditional expression can be used as a shorthand for some if-else statements. The general syntax of the conditional operator is:

<expression1> ? <expression2> : <expression3>

This is an expression, not a statement, so it represents a value. The operator works by evaluating expression1. If it is true (non-zero), it evaluates and returns expression2 .

Otherwise, it evaluates and returns expression3.

The classic example of the ternary operator is to return the smaller of two variables. Every once in a while, the following form is just what you needed. Instead of...

if (x < y) {

min = x;

}

else {

min = y;

```
}
12
```

You just say...

```
min = (x < y) ? x : y;
```

Switch Statement

The switch statement is a sort of specialized form of if used to efficiently separate different blocks of code based on the value of an integer. The switch expression is evaluated, and then the flow of control jumps to the matching const-expression case. The case expressions are typically int or char constants. The switch statement is probably the single most syntactically awkward and error-prone features of the C language.

```
switch (<expression>) {
case <const-expression-1>:
<statement>
break;
case <const-expression-2>:
<statement>
break;
case <const-expression-3>: // here we combine case 3 and 4
case <const-expression-4>:
<statement>
break;
default: // optional
<statement>
}
```

Each constant needs its own case keyword and a trailing colon (:). Once execution has jumped to particular case, the program will keep running through all the cases from that point down -- this so called "fall through" operation is used in the above example so that expression-3 and expression-4 run the same statements. The explicit break statements are necessary to exit the switch. Omitting the break statements is a common error – it compiles, but leads to inadvertent fall-through behavior.

Why does the switch statement fall-through behavior work the way it does? The best explanation I can think of is that originally C was developed for an audience of assembly language programmers. The assembly language programmers were used to the idea of a jump table with fall-through behavior, so that's the way C does it (it's also relatively easy to implement it this way.) Unfortunately, the audience for C is now quite different, and the fall-through behavior is widely regarded as a terrible part of the language.

While Loop

The while loop evaluates the test expression before every loop, so it can execute zero times if the condition is initially false. It requires the parenthesis like the if.

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

Do-While Loop

Like a while, but with the test condition at the bottom of the loop. The loop body will always execute at least once. The do-while is an unpopular area of the language, most everyone tries to use the straight while if at all possible.

```
do {
<statement>
} while (<expression>)
```

For Loop

The for loop in C is the most general looping construct. The loop header contains three parts: an initialization, a continuation condition, and an action.

```
for (<initialization>; <continuation>; <action>) {
<statement>
}
```

The initialization is executed once before the body of the loop is entered. The loop continues to run as long as the continuation condition remains true (like a while). After every execution of the loop, the action is executed. The following example executes 10 times by counting 0..9. Many loops look very much like the following...

```
for (i = 0; i < 10; i++) {
<statement>
}
```

C programs often have series of the form 0..(some_number-1). It's idiomatic in C for the above type loop to start at 0 and use < in the test so the series runs up to but not equal to the upper bound. In other languages you might start at 1 and use <= in the test. Each of the three parts of the for loop can be made up of multiple expressions separated by commas. Expressions separated by commas are executed in order, left to right, and represent the value of the last expression. (See the string-reverse example below for a demonstration of a complex for loop.)

Break

The break statement will move control outside a loop or switch statement. Stylistically speaking, break has the potential to be a bit vulgar. It's preferable to use a straight while with a single test at the top if possible. Sometimes you are forced to use a break because the test can occur only somewhere in the midst of the statements in the loop body. To keep the code readable, be sure to make the break obvious -- forgetting to account for the action of a break is a traditional source of bugs in loop behavior.

```
while (<expression>) {
<statement>
<statement>
if (<condition which can only be evaluated here>)
break;
<statement>
<statement>
}
// control jumps down here on the break
```

The break does not work with if. It only works in loops and switches. Thinking that a break refers to an if when it really refers to the enclosing while has created some high quality bugs. When using a break, it's nice to write the enclosing loop to iterate in the most straightforward, obvious, normal way, and then use the break to explicitly catch the exceptional, weird cases.

Continue

The continue statement causes control to jump to the bottom of the loop, effectively skipping over any code below the continue. As with break, this has a reputation as being vulgar, so use it sparingly. You can almost always get the effect more clearly using an if inside your loop.

```
while (<expression>) {
...
if (<condition>)
continue;
...
...
// control jumps here on the continue
}
```

CHAPTER - 7

Complex Data Types

C has the usual facilities for grouping things together to form composite types—arrays and records (which are called "structures"). The following definition declares a type called "struct fraction" that has two integer sub fields named "numerator" and "denominator". If you forget the semicolon it tends to produce a syntax error in whatever thing follows the struct declaration.

```
struct fraction {

int numerator;

int denominator;

}; // Don't forget the semicolon!
```

This declaration introduces the type struct fraction (both words are required) as a new type. C uses the period (.) to access the fields in a record. You can copy two records of the same type using a single assignment statement, however == does not work on structs.

```
struct fraction f1, f2; // declare two fractions

f1.numerator = 22;

f1.denominator = 7;

f2 = f1; // this copies over the whole struct
```

Arrays

The simplest type of array in C is one which is declared and used in one place. There are more complex uses of arrays which I will address later along with pointers. The following declares an array called scores to hold 100 integers and sets the first and last elements. C arrays are always indexed from 0. So the first int in scores array is scores[0] and the last is scores[99].

int scores[100];

scores[0] = 13; // set first element

scores[99] = 42; // set last element

0

scores

Index 1 2 99

Someone else's memory off either end of the array — do not read or write this memory. There is space for each int element in the scores array — this element is referred to as scores[0].

-5673 22541 42

These elements have random values because the code has not yet initialized them to anything. The name of the array refers to the whole array. (implementation) it works by representing a pointer to the start of the array.

It's a very common error to try to refer to non-existent scores[100] element. C does not do any run time or compile time bounds checking in arrays. At run time the code will just access or mangle whatever memory it happens to hit and crash or misbehave in some unpredictable way thereafter. "Professional programmer's language." The convention of numbering things 0..(number of things - 1) pervades the language. To best integrate with C and other C programmers, you should use that sort of numbering in your own data structures as well.

Multidimensional Arrays

The following declares a two-dimensional 10 by 10 array of integers and sets the first and last elements to be 13.

```
int board [10][10];

board[0][0] = 13;

board[9][9] = 13;
```

The implementation of the array stores all the elements in a single contiguous block of memory. The other possible implementation would be a combination of several distinct one dimensional arrays -- that's not how C does it. In memory, the array is arranged with the elements of the rightmost index next to each other. In other words, board[1][8] comes right before board[1][9] in memory. (highly optional efficiency point) It's typically efficient to access memory which is near other recently accessed memory. This means that the most efficient way to read through a chunk of the array is to vary the rightmost index the most frequently since that will access elements that are near each other in memory.

Array of Structs

The following declares an array named "numbers" which holds 1000 struct fraction's.

```
struct fraction numbers[1000];

numbers[0].numerator = 22; /* set the 0th struct fraction */

numbers[0].denominator = 7;
```

Here's a general trick for unraveling C variable declarations: look at the right hand side and imagine that it is an expression. The type of that expression is the left hand side. For the above declarations, an expression which looks like the right hand side (numbers[1000], or really anything of the form numbers[...]) will be the type on the left hand side (struct fraction).

Pointers

A pointer is a value which represents a reference to another value sometimes known as the pointer's "pointee". Hopefully you have learned about pointers somewhere else, since the preceding sentence is probably inadequate explanation. This discussion will concentrate on the syntax of pointers in C -- for a much more complete discussion of pointers and their use see http://cslibrary.stanford.edu/102/, Pointers and Memory.

Syntax

Syntactically C uses the asterisk or "star" (*) to indicate a pointer. C defines pointer types based on the type pointee. A char* is type of pointer which refers to a single char.

A struct fraction* is type of pointer which refers to a struct fraction.

```
int* intPtr; // declare an integer pointer variable intPtr

char* charPtr; // declares a character pointer --

// a very common type of pointer

// Declare two struct fraction pointers

// (when declaring multiple variables on one line, the *

// should go on the right with the variable)

struct fraction *f1, *f2;
```

The Floating "*"

In the syntax, the star is allowed to be anywhere between the base type and the variable name. Programmer's have their own conventions-- I generally stick the * on the left with the type. So the above declaration of intPtr could be written equivalently...

```
int *intPtr; // these are all the same

int * intPtr;

int* intPtr;
```

Pointer Dereferencing

We'll see shortly how a pointer is set to point to something -- for now just assume the pointer points to memory of the appropriate type. In an expression, the unary * to the left of a pointer dereferences it to retrieve the value it points to.

There's an alternate, more readable syntax available for dereferencing a pointer to a struct. A "->" at the right of the pointer can access any of the fields in the struct. So the reference to the numerator field could be written f1->numerator.

Here are some more complex declarations...

struct fraction** fp; // a pointer to a pointer to a struct fraction

struct fraction fract_array[20]; // an array of 20 struct fractions

struct fraction* fract_ptr_array[20]; // an array of 20 pointers to

// struct fractions

One nice thing about the C type syntax is that it avoids the circular definition problems which come up when a pointer structure needs to refer to itself. The following definition defines a node in a linked list. Note that no preparatory declaration of the node pointer type is necessary.

struct node {

int data;

struct node* next;

};

The & Operator

The & operator is one of the ways that pointers are set to point to things. The & operator computes a pointer to the argument to its right. The argument can be any variable which takes up space in the stack or heap (known as an "LValue" technically). So &i and &(f1->numerator) are ok, but &6 is not. Use & when you have some memory, and you want a pointer to that memory.

```
void foo() {
int* p; // p is a pointer to an integer
int i; // i is an integer
p = &i; // Set p to point to i
*p = 13; // Change what p points to -- in this case i -- to 13
// At this point i is 13. So is *p. In fact *p is i.
}
p
i 13
```

When using a pointer to an object created with &, it is important to only use the pointer so long as the object exists. A local variable exists only as long as the function where it is declared is still executing (we'll see functions shortly). In the above example, i exists only as long as foo() is executing. Therefore any pointers which were initialized with &i are valid only as long as foo() is executing. This "lifetime" constraint of local memory is standard in many languages, and is something you need to take into account when using the & operator.

NULL

A pointer can be assigned the value 0 to explicitly represent that it does not currently have a pointee. Having a standard representation for "no current pointee" turns out to be very handy when using pointers. The constant NULL is defined to be 0 and is typically used when setting a pointer to NULL. Since it is just 0, a NULL pointer will behave like a boolean false when used in a boolean context. Dereferencing a NULL pointer is an error which, if you are lucky, the computer will detect at runtime -- whether the computer detects this depends on the operating system.

Pitfall -- Uninitialized Pointers

When using pointers, there are two entities to keep track of. The pointer and the memory it is pointing to, sometimes called the "pointee". There are three things which must be done for a pointer/pointee relationship to work...

(1) The pointer must be declared and allocated

(2) The pointee must be declared and allocated

(3) The pointer (1) must be initialized so that it points to the pointee (2)

The most common pointer related error of all time is the following: Declare and allocate the pointer (step 1). Forget step 2 and/or 3. Start using the pointer as if it has been setup to point to something. Code with this error frequently compiles fine, but the runtime results are disastrous. Unfortunately the pointer does not point anywhere good unless (2) and (3) are done, so the run time dereference operations on the pointer with * will misuse and trample memory leading to a random crash at some point.

```
{
int* p;
*p = 13; // NO NO NO p does not point to an int yet
// this just overwrites a random area in memory
}
-14346
p
i
```

Of course your code won't be so trivial, but the bug has the same basic form: declare a pointer, but forget to set it up to point to a particular pointee.

Using Pointers

Declaring a pointer allocates space for the pointer itself, **but it does not allocate space for the pointee**. The pointer must be set to point to something before you can dereference it.

Here's some code which doesn't do anything useful, but which does demonstrate (1) (2)

(3) for pointer use correctly...

```
int* p;                 // (1) allocate the pointer

int i;                  // (2) allocate pointee

struct fraction f1;     // (2) allocate pointee

p = &i;                     // (3) setup p to point to i

*p = 42;                    // ok to use p since it's setup

p = &(f1.numerator);        // (3) setup p to point to a different int

*p = 22;

p = &(f1.denominator);      // (3)

*p = 7;
```

So far we have just used the & operator to create pointers to simple variables such as i. Later, we'll see other ways of getting pointers with arrays and other techniques.

C Strings

C has minimal support of character strings. For the most part, strings operate as ordinary arrays of characters. Their maintenance is up to the programmer using the standard facilities available for arrays and pointers. C does include a standard library of functions which perform common string operations, but the programmer is responsible for the managing the string memory and calling the right functions. Unfortunately computations involving strings are very common, so becoming a good C programmer often requires becoming adept at writing code which manages strings which means managing pointers and arrays.

A C string is just an array of char with the one additional convention that a "null" character ('\0') is stored after the last real character in the array to mark the end of the string. The compiler represents string constants in the source code such as "binky" as arrays which follow this convention. The string library functions (see the appendix for a partial list) operate on strings stored in this way. The most useful library function is strcpy(char dest[], const char source[]); which copies the bytes of one string over to another. The order of the arguments to strcpy() mimics the arguments in of '=' -- the right is assigned to the left. Another useful string function is strlen(const char string[]); which returns the number of characters in C string not counting the trailing '\0'.

Note that the regular assignment operator (=) does **not** do string copying which is why strcpy() is necessary. See Section 6, Advanced Pointers and Arrays, for more detail on how arrays and pointers work.

The following code allocates a 10 char array and uses strcpy() to copy the bytes of the string constant "binky" into that local array.

```
{

char localString[10];

strcpy(localString, "binky");

}

b i n k y 0 x x x x

0 1 2 ...

localString
```

The memory drawing shows the local variable localString with the string "binky" copied into it. The letters take up the first 5 characters and the '\0' char marks the end of the string after the 'y'. The x's represent characters which have not been set to any particular value. If the code instead tried to store the string "I enjoy languages whichh have good string support" into localString, the code would just crash at run time since the 10 character array can contain at most a 9 character string. The large string will be written passed the right hand side of localString, overwriting whatever was stored there.

String Code Example

Here's a moderately complex for loop which reverses a string stored in a local array. It demonstrates calling the standard library functions strcpy() and strlen() and demonstrates that a string really is just an array of characters with a '\0' to mark the effective end of the string. Test your C knowledge of arrays and for loops by making a drawing of the memory for this code and tracing through its execution to see how it works.

```
{
char string[1000]; // string is a local 1000 char array
int len;
strcpy(string, "binky");
len = strlen(string);
/*
Reverse the chars in the string:
i starts at the beginning and goes up
j starts at the end and goes down
i/j exchange their chars as they go until they meet
*/
int i, j;
char temp;
for (i = 0, j = len - 1; i < j; i++, j--) {
temp = string[i];
string[i] = string[j];

string[j] = temp;

}

// at this point the local string should be "yknib"

}
```

"Large Enough" Strings

The convention with C strings is that the owner of the string is responsible for allocating array space which is "large enough" to store whatever the string will need to store. Most routines do not check that size of the string memory they operate on, they just assume its big enough and blast away. Many, many programs contain declarations like the following...

```
{

char localString[1000];

...

}
```

The program works fine so long as the strings stored are 999 characters or shorter. Someday when the program needs to store a string which is 1000 characters or longer, then it crashes. Such array-not-quite-big-enough problems are a common source of bugs, and are also the source of so called "buffer overflow" security problems. This scheme has the additional disadvantage that most of the time when the array is storing short strings, 95% of the memory reserved is actually being wasted. A better solution allocates the string dynamically in the heap, so it has just the right size. To avoid buffer overflow attacks, production code should check the size of the data first, to make sure it fits in the destination string. See the strlcpy() function in Appendix A.

char*

Because of the way C handles the types of arrays, the type of the variable localString above is essentially char*. C programs very often manipulate strings using variables of type char* which point to arrays of characters. Manipulating the actual chars in a string requires code which manipulates the underlying array, or the use of library functions such as strcpy() which manipulate the array for you. See Section 6 for more detail on pointers and arrays.

TypeDef

A typedef statement introduces a shorthand name for a type. The syntax is...

```
typedef <type><name>;
```

The following defines Fraction type to be the type (struct fraction). C is case sensitive, so fraction is different from Fraction. It's convenient to use typedef to create types with upper case names and use the lower-case version of the same word as a variable.

```
typedef struct fraction Fraction;

Fraction fraction; // Declare the variable "fraction" of type "Fraction"

// which is really just a synonym for "struct fraction".
```

The following typedef defines the name Tree as a standard pointer to a binary tree node where each node contains some data and "smaller" and "larger" subtree pointers.

```
typedef struct treenode* Tree;

struct treenode {

int data;

Tree smaller, larger; // equivalently, this line could say
}; // "struct treenode *smaller, *larger"
```

Functions

All languages have a construct to separate and package blocks of code. C uses the "function" to package blocks of code. This article concentrates on the syntax and peculiarities of C functions. The motivation and design for dividing a computation into separate blocks is an entire discipline in its own.

A function has a name, a list of arguments which it takes when called, and the block of code it executes when called. C functions are defined in a text file and the names of all the functions in a C program are lumped together in a single, flat namespace. The special function called "main" is where program execution begins. Some programmers like to begin their function names with Upper case, using lower case for variables and parameters, Here is a simple C function declaration. This declares a function named Twice which takes a single int argument named num. The body of the function computes the value which is twice the num argument and returns that value to the caller.

```
/*
Computes double of a number.
Works by tripling the number, and then subtracting to get back to double.
*/
static int Twice(int num) {
int result = num * 3;
result = result - num;
return(result);
}
```

Syntax

The keyword "static" defines that the function will only be available to callers in the file where it is declared. If a function needs to be called from another file, the function cannot be static and will require a prototype -- see prototypes below. The static form is convenient for utility functions which will only be used in the file where they are declared. Next , the "int" in the function above is the type of its return value. Next comes name of the function and its list of parameters. When referring to a function by name in documentation or other prose, it's a convention to keep the parenthesis () suffix, so in this case I refer to the function as "Twice()". The parameters are listed with their types and names, just like variables. Inside the function, the parameter num and the local variable result are "local" to the function -- they get their own memory and exist only so long as the function is executing. This independence of "local" memory is a standard feature of most languages.

The "caller" code which calls Twice() looks like...

```
int num = 13;

int a = 1;

int b = 2;

a = Twice(a); // call Twice() passing the value of a

b = Twice(b + num); // call Twice() passing the value b+num

// a == 2

// b == 30

// num == 13 (this num is totally independent of the "num" local to Twice()
```

Call by Value vs. Call by Reference

C passes parameters "by value" which means that the actual parameter values are copied into local storage. The caller and callee functions do not share any memory --

they each have their own copy. This scheme is fine for many purposes, but it has twodisadvantages.

1) Because the callee has its own copy, modifications to that memory are not communicated back to the caller. Therefore, value parameters do not allow the callee to communicate back to the caller. The function's return value can communicate some information back to the caller, but not all problems can be solved with the single return value.

2) Sometimes it is undesirable to copy the value from the caller to the callee because the value is large and so copying it is expensive, or because at a conceptual level copying the value is undesirable.

The alternative is to pass the arguments "by reference". Instead of passing a copy of a value from the caller to the callee, pass a pointer to the value. In this way there is only one copy of the value at any time, and the caller and callee both access that one value through pointers.

Some languages support reference parameters automatically. C does not do this – the programmer must implement reference parameters manually using the existing pointer constructs in the language.

Swap Example

The classic example of wanting to modify the caller's memory is a swap() function which exchanges two values. Because C uses call by value, the following version of Swap will not work...

```
void Swap(int x, int y) { // NO does not work

int temp;

temp = x;

x = y; // these operations just change the local x,y,temp
y = temp; // -- nothing connects them back to the caller's a,b

}

// Some caller code which calls Swap()...

int a = 1;

int b = 2;

Swap(a, b);
```

Swap() does not affect the arguments a and b in the caller. The function above only operates on the copies of a and b local to Swap() itself. This is a good example of how "local" memory such as (x, y, temp) behaves -- it exists independent of everything else only while its owning function is running. When the owning function exits, its local memory disappears.

Reference Parameter Technique

To pass an object X as a reference parameter, the programmer must pass a pointer to X instead of X itself. The formal parameter will be a pointer to the value of interest. The caller will need to use & or other operators to compute the correct pointer actual parameter. The callee will need to dereference the pointer with * where appropriate to access the value of interest. Here is an example of a correct Swap() function.

```
static void Swap(int* x, int* y) { // params are int* instead of int

int temp;

temp = *x; // use * to follow the pointer back to the caller's memory

*x = *y;

*y = temp;

}


// Some caller code which calls Swap()...
```

```
int a = 1;

int b = 2;

Swap(&a, &b);
```

Things to notice...

• The formal parameters are int* instead of int.

• The caller uses & to compute pointers to its local memory (a,b).

• The callee uses * to dereference the formal parameter pointers back to get the caller's memory.

Since the operator & produces the address of a variable -- &a is a pointer to a. In Swap() itself, the formal parameters are declared to be pointers, and the values of interest (a,b) are accessed through them. There is no special relationship between the **names** used for the actual and formal parameters. The function call matches up the actual and formal parameters by their order -- the first actual parameter is assigned to the first formal parameter, and so on. I deliberately used different names (a,b vs x,y) to emphasize that the names do not matter.

const

The qualifier const can be added to the left of a variable or parameter type to declare that the code using the variable will not change the variable. As a practical matter, use of const is very sporadic in the C programming community. It does have one very handy use, which is to clarify the role of a parameter in a function prototype...

void foo(**const** struct fraction* fract);

In the foo() prototype, the const declares that foo() does not intend to change the struct fraction pointee which is passed to it. Since the fraction is passed by pointer, we could not know otherwise if foo() intended to change our memory or not. Using the const, foo() makes its intentions clear. Declaring this extra bit of information helps to clarify the role of the function to its implementor and caller.