Biyani's Think Tank

Concept based notes

'C' Language

M.C.A

Neha Jain
M.C.A
Lecturer
Deptt. of Information Technology
Biyani Girls College, Jaipur

Published by :
Think Tanks
Biyani Group of Colleges

Concept & Copyright :
©Biyani Shikshan Samiti
Sector-3, Vidhyadhar Nagar,
Jaipur-302 023 (Rajasthan)
Ph : 0141-2338371, 2338591-95 • Fax : 0141-2338007
E-mail : acad@biyanicolleges.org
Website :www.gurukpo.com; www.biyanicolleges.org

First Edition : 2012

Leaser Type Setted by :
Biyani College Printing Department

Preface

I am glad to present this book, especially designed to serve the needs of the students. The book has been written keeping in mind the general weakness in understanding the fundamental concepts of the topics. The book is self-explanatory and adopts the "Teach Yourself" style. It is based on question-answer pattern. The language of book is quite easy and understandable based on scientific approach.

Any further improvement in the contents of the book by making corrections, omission and inclusion is keen to be achieved based on suggestions from the readers for which the author shall be obliged.

I acknowledge special thanks to Mr. Rajeev Biyani, *Chairman* & Dr. Sanjay Biyani, *Director* (*Acad.*) Biyani Group of Colleges, who are the backbones and main concept provider and also have been constant source of motivation throughout this Endeavour. They played an active role in coordinating the various stages of this Endeavour and spearheaded the publishing work.

I look forward to receiving valuable suggestions from professors of various educational institutions, other faculty members and students for improvement of the quality of the book. The reader may feel free to send in their comments and suggestions to the under mentioned address.

Neha Jain

Author

Syllabus

MCA

Problem Solving with Computers: Algorithms, and Flowcharts. Data types, constants, variables, operators, data input and output, assignment statements, conditional statements, string and character handling, data validation examples.
Iteration, arrays, strings processing, defining function, types of functions, function prototype, passing parameters, recursion. Storage class specifiers, pre-processor, header files and standard functions. Pointers: Definition and uses of pointers, pointer arithmetic, pointers and array, pointers and functions, pointer to pointer. Structures, union, pointers to structures, user-defined data types, enumeration.
Data files: Opening, closing, creating, processing and unformatted data files. Introduction to Dynamic Memory Allocation, command line arguments, systems calls.

Contents

Chapter-1
Problem Solving with Computers

Q.1 What is algorithm?

Ans. An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem. In mathematics, an algorithm is a defined set of step-by-step procedures that provides the correct answer to a particular problem.

Or

An algorithm for a given problem is a precise description of steps for a computation that produces a solution to the problem. An algorithm is a representation of a solution to a problem. If a problem can be defined as a difference between a desired situation and the current situation in which one is, then a problem solution is a procedure, or method, for transforming the current situation to the desired one.

Or

An algorithm is procedure consisting of a finite set of unambiguous rules (instructions) which specify a finite sequence of operations that provides the solution to a problem, or to a specific class of problems for any allowable set of input quantities (if there are inputs). In other word, an algorithm is a step-by-step procedure to solve a given problem

Q.2 Explain flowchart?

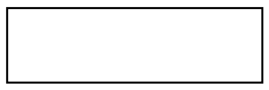
Ans. A flowchart is a diagram made up of boxes, diamonds and other shapes, connected by arrows - each shape represents a step in the process, and the arrows show the order in which they occur. Flowcharting combines symbols and flowlines, to show figuratively the operation of an algorithm.

Flowcharting Symbols

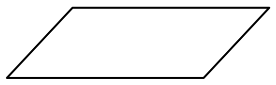
There are 6 basic symbols commonly used in flowcharting of assembly language

Programs:

Terminal,

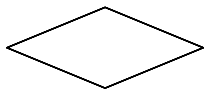


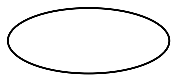
Process,



input/output,

Decision,

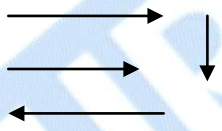




Connector and

Predefined Process.

Flow Lines



This is not a complete list of all the possible flowcharting symbols, it is the ones used most often in the structure of Assembly language programming.

Symbol Name Function

Process Indicates any type of internal operation inside the Processor or Memory

input/output Used for any Input / Output (I/O) operation.

Indicates that the computer is to obtain data or output results

Decision Used to ask a question that can be answered in a binary format (Yes/No, True/False)

Connector Allows the flowchart to be drawn without intersecting lines or without a reverse flow.

Predefined Process Used to invoke a subroutine or an interrupt program.

Terminal Indicates the starting or ending of the program, process, or interrupt program.

Flow Lines Shows direction of flow.

Generally, there are many standard flowcharting symbols.

General Rules for flowcharting

1. All boxes of the flowchart are connected with Arrows. (Not lines)
2. Flowchart symbols have an entry point on the top of the symbol with no other entry points. The exit point for all flowchart symbols is on the bottom except for the Decision symbol.
3. The Decision symbol has two exit points; these can be on the sides or the bottom and one side.
4. Generally a flowchart will flow from top to bottom. However, an upward flow can be shown as long as it does not exceed 3 symbols.
5. Connectors are used to connect breaks in the flowchart. Examples are:

   • From one page to another page.

   • From the bottom of the page to the top of the same page.

   • An upward flow of more then 3 symbols
6. Subroutines and Interrupt programs have their own and independent flowcharts.
7. All flow charts start with a Terminal or Predefined Process (for interrupt programs or subroutines) symbol.
8. All flowcharts end with a terminal or a contentious loop.

   Flowcharting uses symbols that have been in use for a number of years to represent the type of operations and/or processes being performed. The standardized format provides a common method for people to visualize problems together in the same manner. The use of standardized symbols makes the flow charts easier to interpret, however, standardizing symbols is not as important as the sequence of activities that make up the process.

Flowcharting Tips

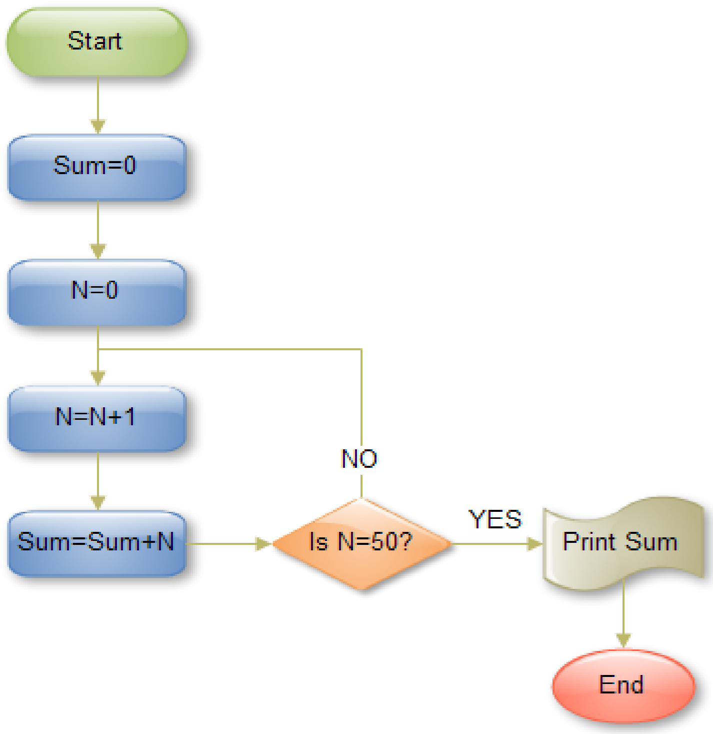
- Chart the process the way it is really occurring. Do not document the way a written process or a manager thinks the process happens.

- People typically modify existing processes to enable a more efficient process. If the desired or theoretical process is charted, problems with the existing process will not be recognised and no improvements can be made.

  Note all circumstances actually dealt with.
- Test the flow chart by trying to follow the chart to perform the process charted. If there is a problem performing the operation as charted, note any differences and modify the chart to correct. A better approach would be to have someone unfamiliar with the process try to follow the flow chart and note questions or problems found.
- Include mental steps in the process such as decisions. These steps are sometimes left out because of familiarity with the process, however, represent sources of problems due to a possible lack of information used to make the decision can be inadequate or incorrect if performed by a different person.Example

Draw a flowchart to find the sum of first 50 natural numbers

The corresponding flowchart is as



Follows:

Chapter-2

Overview of C programming simple program construction

Q.1 Explain Data Types in C?

OR

How many Primitive Data Types are there in C?

Ans.: A Data Type defines a set of values and a set of operations that can be applied on those values. A summary of the basic fundamental data types in , as well as the range of values that can be represented with each one:

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int(short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295

float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)

* The values of the columns Size and Range depend on the system the program is compiled for. The values shown above are those found on most 32-bit systems. But for other systems, the general specification is that int has the natural size suggested by the system architecture (one "word") and the four integer types char, short, int andlong must each one be at least as large as the one preceding it, with char being always one byte in size. The same applies to the floating point types float, double and long double, where each one must provide at least as much precision as the preceding one.

Declaration of variables : In order to use a variable in , we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier. For example:

int a;

float mynumber;

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the identifier mynumber. Once declared, the variables a and mynumber can be used within the rest of their scope in the program.

The integer data types char, short, long and int can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier signed or the specifier unsigned before the type name. For example:

unsigned short int NumberofSisters;

signed int MyAccountBalance;

By default, if we do not specify either signed or unsigned most compiler settingswill assume the type to be signed, therefore instead of the second declaration above we could have written:

int MyAccountBalance;

with exactly the same meaning (with or without the keyword signed) .An exception to this general rule is the char type, which exists by itself and is considered a different fundamental data type from signed char and unsigned char, thought to store characters. You should use either signed or unsigned if you intend to store numerical values in a char-sized variable. short and long can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: short is equivalent to short int and long is equivalent to long int. The following two variable declarations are equivalent:

short Year;

short int Year;

Finally, signed and unsigned may also be used as standalone type specifiers, meaning the same as signed int and unsigned int respectively. The following two declarations are equivalent:

unsigned NextYear;

unsigned int NextYear;

```
// operating with variables
#include <stdio.h>
int main () {
// declaring variables:
int a, b,result;
// process:
a = 5;

b = 2; a = a + 1;
    result = a - b;
```

```
  // print out the result:
printf("%d",result);
  // terminate the program:
return 0;
}
```

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.

Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

The scope of local variables is limited to the block enclosed in braces ({}) where they are declared.

Initialization of variables :-When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in :

The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

type identifier = initial_value ;

For example, if we want to declare an int variable called a initialized with a value

of 0 at the moment in which it is declared, we could write:

int a = 0;

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses (()):

type identifier (initial_value) ;

Constants: Constants are expressions with a fixed value.

Literals: Literals are used to express particular values within the source code of a program. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out

Q.2 Explain the different types of operators in C?

Ans.: Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result.

1. Assisgnment(=):-The assignment operator assigns a value to a variable.

a = 5;

This statement assigns the integer value 5 to the variable a. The part at the left of the assignment operator (=) is known as the lvalue (left value) and the right one as the rvalue (right value). The lvalue has to be a variable whereas the rvalue can be either a constant, a variable, the result of an operation or any combination of these.

The most important rule when assigning is the right-to-left rule: The assignment operation always takes place from right to left, and never the other way:
a = b;

This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost.

A property that has over other programming languages is that the assignment operation can be used as the value (or part of an rvalue) for another assignment operation. For example: a = 2 + (b = 5);
is equivalent to: b = 5; a = 2 + b; that means: first assign 5 to variable b and then assign to a the value 2 plus the result of the previous assignment of b (i.e. 5), leaving a with a final value of 7.

The following expression is also valid in :
a = b = c = 5;
It assigns 5 to the all the three variables: a, b and c.

2. Arithmetic operators (+, -, *, /, %):- The five arithmetical operations supported by the language are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is modulo; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

a = 11 % 3;

the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

3. Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=,^=, |=):-When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators.

expression	is equivalent to
value += increase	value = value + increase;

and the same for all other operators

4. Increase and decrease (++, --):-the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

;

c+=1;

c=c+1;

are all equivalent in its functionality: the three of them increase by one the value of c. A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++).

5. **Relational and equality operators (==, !=, >, <, >=, <=):-** In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.
We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in :

==	Equal to
!=	Not equal to
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to

The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it are equal to each other.

6. **Logical operators (!, &&, | |):-** The Operator ! is the operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:
!(5 == 5) // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4) // evaluates to true because (6 <= 4) would be false.
!true // evaluates to false
!false // evaluates to true.

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The
operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands
are true, and false otherwise.

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves.

7. **Conditional operator (?):-** The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:
condition ? result1 : result2
If condition is true the expression will return result1, if it is not it will return result2.
7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3 // returns 4, since 7 is equal to 5+2.
5>3 ? a : b // returns the value of a, since 5 is greater than 3.
a>b ? a : b // returns whichever is greater, a or b.

8. **Comma operator (,):-** The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

Q.3 Explain control structures available in ;

Ans. Conditional structure: if and else
The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:
if (condition) statement
Where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

For example, the following code fragment prints x is 100 only if the value stored in the x variable is indeed 100:

```
if (x == 100)
printf( "x is 100");
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:

```
if (x == 100)
{
printf( "x is ");
printf( “%d”,x);
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword else. Its form used in conjunction with if is:

if (condition) statement1 else statement2

For example:

```
if (x == 100)
printf( "x is 100");
else
printf( "x is not 100");
```

prints on the screen x is 100 if indeed x has a value of 100, but if it has not - and only if not- it prints out x is not 100.

The if + else structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
printf( "x is positive");
else if (x < 0)
printf( "x is negative");
else
printf( "x is 0");
```

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

Iteration structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

while (expression) statement

and its functionality is simply to repeat statement while the condition set in expression is true.

For example, we are going to make a program to countdown using a while-loop:

```
// custom countdown using while
#include <stdio.h>
int main ()
{
int n;
printf( "Enter the starting number ) ";
scanf("%d",;
while (n>0) {
printf("%d", n);
--n;
}
printf( "FIRE");
return 0;
}
```

Output: Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition n>0 (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition (n>0) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. User assigns a value to n

2. The while condition is checked (n>0). At this point there are two posibilities:
   * condition is true: statement is executed (to step 3)
   * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   printf("%d", n);
   --n;
   (prints the value of n on the screen and decreases n by 1)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included --n; that decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition (n>0) to become false after a certain number of loop iterations: to be more specific, when n becomes 0, that is where our while-loop and our countdown end.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

The do-while loop
Its format is:
do statement while (condition);
Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

```
// number echoer
#include <stdio.h>

int main ()
```

```
{
unsigned long n;
do {
printf( "Enter number (0 to end): ");
scanf("%d",& n);
printf( "You entered: %d" , n);
} while (n != 0);
return 0;
}
```

Output

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value 0 in the previous example you can be prompted for more numbers forever.

The for loop

Its format is:

for (initialization; condition; increase) statement;

and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.

2. condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```
// countdown using a for loop
#include <stdio.h>
int main ()
{
for (int n=10; n>0; n--) {
printf("%d", n);
}
printf( "FIRE");
return 0;
}
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
// whatever here...
}
```

This loop will execute for 50 times if neither n or i are modified within the loop:

n starts with a value of 0, and i with 100, the condition is n!=i (that n is not equal to i). Because n is increased by one and i decreased by one, the loop's condition will become false after the 50th loop, when both n and i will be equal to 50.

Jump statements.

The break statement

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite
loop, or to force it to end before its natural end. For example, we are going to stop the count down before its
natural end (maybe because of an engine check failure?):

```
// break loop example
#include <stdio.h>
int main ()
{
int n;
for (n=10; n>0; n--)
{
printf("%d", n)";
if (n==3)
{
printf( "countdown aborted!");

break;
}
}
return 0;
}
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

```
// continue loop example
```

```
#include <stdio.h>
int main ()
{
for (int n=10; n>0; n--) {
if (n==5) continue;
printf("%d",n);
}
printf( "FIRE!");
return 0;
}
10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!
```

The goto statement

goto allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.

The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:

```
// goto loop example
#include <stdio.h>
int main ()
{
int n=10;
loop:
printf("%d", n);
n--;
if (n>0) goto loop;
printf( "FIRE");
return 0;
}
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The exit function

exit is a function defined in the cstdlib library.

The purpose of exit is to terminate the current program with a specific exit code. Its prototype is:

void exit (int exitcode);

The exitcode is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

The selective structure: switch.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several if and else if instructions. Its form is the following:

```
switch (expression)
{
case constant1:
group of statements 1;
break;
case constant2:
group of statements 2;
break;
.
.
.
default:
default group of statements
}
```

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:
switch example if-else equivalent

```
switch (x) {
case 1:
printf( "x is 1");
break;
case 2:
printf( "x is 2");
break;
default:
printf( "value of x unknown");
}
if (x == 1) {
printf( "x is 1");
}
else if (x == 2) {
printf( "x is 2");
}
else {
printf("value of x unknown");
}
```

The switch statement is a bit peculiar within the language because it uses labels instead of blocks. This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached.

For example, if we did not include a break statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch selective block. This makes unnecessary to include braces { } surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:

```
switch (x) {
case 1:
case 2:
case 3:
printf("x is 1, 2 or 3");
break;
default:
printf( "x is not 1, 2 nor 3");
}
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example case n: where n is a variable) or ranges (case (1..3):) because they are not valid constants.

If you need to check ranges or values that are not constants, use a concatenation of if and else if statements.

Chapter-3

Functions

Q.1 Explain function in C ?

Ans. A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

type name (parameter1, parameter2, ...) { statements }

where:

- type is the data type specifier of the data returned by the function.
- name is the identifier by which it will be possible to call the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces { }.

Here you have the first function example:

```
// function example
# include <stdio.h>

int addition (int a, int b)
{
int r;
r=a+b;
return (r);
}
int main ()
{
int z;
z = addition (5,3);
printf("%d",z);
return 0;
}
```

a program always begins its execution by the main function. the main function begins by declaring the variable z of type int. Right after that, we see a call to a function called addition.

Q.2 Explain call-by-value and call by reference?

Ans. When calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

```
int x=5, y=3, z;

z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves. when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

When we need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following

Chapter-4

Arrays

Q.1 How Arrays are declared and initialized in C?

OR

What is an Array and how it is declared and initialized?

Ans.: An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. That means that, for example, we can store 5 values of type int in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, int for example, with a unique identifier.

These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length. Like a regular variable, an array must be declared before it is used. A typical declaration for an array in is:

type name [elements];

where type is a valid type (like int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies how many of these elements the array has to contain.

The elements field within brackets [] which represents the number of elements the array is going to hold, must be a constant value, since arrays are blocks of non-dynamic memory whose size must be determined before execution.

Initializing arrays:-When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros. In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }. For example:

int billy [5] = { 16, 2, 77, 40, 12071 };

This declaration would have created an array like this: 55

The amount of values between braces { } must not be larger than the number of elements that we declare for the array between square brackets [ ]. For example, in the example of array billy we have declared that it has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element.

When an initialization of values is provided for an array, allows the possibility of leaving the square brackets empty [ ]. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }:

int billy [] = { 16, 2, 77, 40, 12071 };

After this declaration, array billy would be 5 ints long, since we have provided 5 initialization values.

Accessing the values of an array.

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

name[index]

Following the previous examples in which billy had 5 elements and each of those elements was of type int, the name which we can use to refer to each element is the following:

For example, to store the value 75 in the third element of billy, we could write the following statement:

billy[2] = 75;

and, for example, to pass the value of the third element of billy to a variable called a, we could write:

a = billy[2];

Therefore, the expression billy[2] is for all purposes like a variable of type int.

Notice that the third element of billy is specified billy[2], since the first one is billy[0], the second one is billy[1], and therefore, the third one is billy[2]. By this same reason, its last element is billy[4]. Therefore, if

we write billy[5], we would be accessing the sixth element of billy and therefore exceeding the size of the array.

In it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors. The reason why this is allowed will be seen further ahead when we begin to use pointers.

At this point it is important to be able to clearly distinguish between the two uses that brackets [ ] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets [ ] with arrays. T

```
int billy[5];       // declaration of a new array
billy[2] = 75;      // access to an element of the array.
```

If you read carefully, you will see that a type specifier always precedes a variable or array declaration, while it never precedes an access.

Some other valid operations with arrays:

```
billy[0] = a;
billy[a] = 75;
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
// arrays example
#include <stdio.h>
```

```
int billy [] = {16, 2, 77, 40, 12071};
int n, result=0;
int main ()
{
 for ( n=0 ; n<5 ; n++ )
 {
   result += billy[n];
 }
 printf( “%d”,result);
 return 0;
}
```

12206

Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a bidimensional table made of elements, all of them of a same uniform data type. jimmy represents a bidimensional array of 3 per 5 elements of type int. The way to declare this array in would be:

int jimmy [3][5];

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

jimmy[1][3]T

57

(remember that array indices always begin by zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as

needed. But be careful! The amount of memory needed for an array rapidly increases with each dimension. For

example:

char century [100][365][24][60][60];

declares an array with a char element for each second in a century, that is more than 3 billion chars. So this declaration would consume more than 3 gigabytes of memory!

Multidimensional arrays are just an abstraction for programmers, since we can obtain the same results with a simple array just by putting a factor between its indices:

int jimmy [3][5]; // is equivalent to

int jimmy [15]; // (3 * 5 = 15)

With the only difference that with multidimensional arrays the compiler remembers the depth of each imaginary dimension for us. Take as example these two pieces of code, with both exactly the same result. One uses a bidimensional array and the other one uses a simple array: multidimensional array pseudo-multidimensional array

```
#define WIDTH 5
#define HEIGHT 3
int jimmy [HEIGHT][WIDTH];
int n,m;
int main ()
{
 for (n=0;n<HEIGHT;n++)
  for (m=0;m<WIDTH;m++)
  {
  jimmy[n][m]=(n+1)*(m+1);
 }
 return 0;
}
#define WIDTH 5
#define HEIGHT 3
```

```
int jimmy [HEIGHT * WIDTH];
int n,m;
int main ()
{
for (n=0;n<HEIGHT;n++)
  for (m=0;m<WIDTH;m++)
 {
   jimmy[n*WIDTH+m]=(n+1)*(m+1);
   }
 return 0;
}
```

None of the two source codes above produce any output on the screen, but both assign values to the memory block called jimmy in the following way: T

58

We have used "defined constants" (#define) to simplify possible future modifications of the program. For example, in case that we decided to enlarge the array to a height of 4 instead of 3 it could be done simply by changing the line:

```
#define HEIGHT 3
```

to:

```
#define HEIGHT 4
```

with no need to make any other modifications to the program.

Arrays as parameters

At some moment we may need to pass an array to a function as a parameter. In it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address.

In practice this has almost the same effect and it is a much faster and more efficient operation.

In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets []. For example, the following function:

```
void procedure (int arg[])
```

accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters
#include <iostream>
using namespace std;
void printarray (int arg[], int length) {
  for (int n=0; n<length; n++)
 cout << arg[n] << " ";
 cout << "\n";
}
int main ()
{
  int firstarray[] = {5, 10, 15};
int secondarray[] = {2, 4, 6, 8, 10};
printarray (firstarray,3);
 printarray (secondarray,5);
return 0;
```

}

Output: 5 10 15

2 4 6 8 10

As you can see, the first parameter (int arg[]) accepts any array whose elements are of type int, whatever its length. For that reason we have included a second parameter that tells the function the length of each array that T 59

we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the passed array without going out of range.

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

base_type[][depth][depth]

for example, a function with a multidimensional array as argument could be:

void procedure (int myarray[][3][4])

Notice that the first brackets [] are left blank while the following ones are not. This is so because the compiler

must be able to determine within the function which is the depth of each additional dimension.

Arrays, both simple or multidimensional, passed as function parameters are a quite common source of errors for novice programmers. I recommend the reading of the chapter about Pointers for a better understanding on how arrays operate. T

Character Sequences

As you may already know, the Standard Library implements a powerful string class, which is very useful to

handle and manipulate strings of characters. However, because strings are in fact sequences of characters, we can represent them also as plain arrays of char elements.

For example, the following array:

char jenny [20];

is an array that can store up to 20 elements of type char. It can be represented as:

Therefore, in this array, in theory, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences. For example, jenny could store at some point in a program either the sequence "Hello"

or the sequence "Merry christmas", since both are shorter than 20 characters.

Therefore, since the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence: the null character, whose literal constant can be written as '\0'

(backslash, zero).

Our array of 20 elements of type char, called jenny, can be represented storing the characters sequences "Hello"and "Merry Christmas" as:

Notice how after the valid content a null character ('\0') has been included in order to indicate the end of the sequence. The panels in gray color represent char elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays they follow all their same rules. For example, if we want to initialize an array of characters with some predetermined sequence of characters we can do it just like any other array:

char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };

In this case we would have declared an array of 6 elements of type char initialized with the characters that form the word "Hello" plus a null character '\0' at the end.

But arrays of char elements have an additional method to initialize their values: using string literals.

In the expressions we have used in some examples in previous chapters, constants that represent entire strings of characters have already showed up several times. These are specified enclosing the text to become a string literal between double quotes ("). For example: "the result is: "T

61

is a constant string literal that we have probably used already.

Double quoted strings (") are literal constants whose type is in fact a null-terminated array of characters. So string literals enclosed between double quotes always have a null character ('\0') automatically appended at the end.

Therefore we can initialize the array of char elements called myword with a null-terminated sequence of characters by either one of these two methods:

char myword [] = { 'H', 'e', 'l', 'l', 'o', '\0' };

char myword [] = "Hello";

In both cases the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello" plus a final null character ('\0') which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Please notice that we are talking about initializing an array of characters in the moment it is being declared, and not about assigning values to them once they have already been declared. In fact because this type of nullterminated arrays of characters are regular arrays we have the same restrictions that we have with any other array, so we are not able to copy blocks of data with an assignment operation.

Assuming mystext is a char[] variable, expressions within a source code like:

mystext = "Hello";

mystext[] = "Hello";

would not be valid, like neither would be:

mystext = { 'H', 'e', 'l', 'l', 'o', '\0' };

The reason for this may become more comprehensible once you know a bit more about pointers, since then it will be clarified that an array is in fact a constant pointer pointing to a block of memory.

Using null-terminated sequences of characters

Null-terminated sequences of characters are the natural way of treating strings in , so they can be used as such in many procedures. In fact, regular string literals have this type (char[]) and can also be used in most

cases.

For example, cin and cout support null-terminated sequences as valid containers for sequences of characters, so they can be used directly to extract strings of characters from cin or to insert them into cout. For example: T

```
// null-terminated sequences of characters
#include <iostream>
using namespace std;
int main ()
{
char question[] = "Please, enter your first
name: ";
char greeting[] = "Hello, ";
 char yourname [80];
  cout << question;
cin >> yourname;
  cout << greeting << yourname << "!";
return 0;
}
```

```
Please, enter your first name: John
Hello, John!
```

As you can see, we have declared three arrays of char elements. The first two were initialized with string literal constants, while the third one was left uninitialized. In any case, we have to speficify the size of the array: in the first two (question and greeting) the size was implicitly defined by the length of the literal constant they were initialized to. While for yourname we have explicitly specified that it has a size of 80 chars.

Finally, sequences of characters stored in char arrays can easily be converted into string objects just by using the

assignment operator:

string mystring;

char myntcs[]="some text";

mystring = myntcs; T

We have already seen how variables are seen as memory cells that can be accessed using their identifiers. This way we did not have to care about the physical location of our data within memory, we simply used its identifier whenever we wanted to refer to our variable.

The memory of your computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as

Chapter-5

Storage Classes

Q.1 Explain Storage Classes in C.

Ans. The storage class determines the part of memory where storage is allocated for an object (particularly variables and functions) and how long the storage allocation continues to exist.

Or

A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.

Or

It is possible in C to have a partial control over the way variables are allocated. This is done with the <class> field defining the storage class. This information will also control the scope of the variable, meaning the locations in the program where this variable is known and then accessible.

Q.2 How many storage class specifiers does 'c' language provide?Explain all.

Ans. C provides the following storage-class specifiers:

1. auto- Storage Class-:

- They are declared at the start of a program's block such as the curly braces ({ }). Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block.

- The scope of automatic variables is local to the block in which they are declared, including any blocks nested within that block. For these reasons, they are also called local variables.

- No block outside the defining block may have direct access to automatic variables (by variable name) but, they may be accessed indirectly by other blocks and/or functions using pointers.

- Automatic variables may be specified upon declaration to be of storage class auto. However, it is not required to use the keyword auto because by default, storage class within a block is auto.
- Automatic variables declared with initializers are initialized every time the block in which they are declared is entered or accessed.

2. register- Storage Class-:

- Automatic variables are allocated storage in the main memory of the computer; however, for most computers, accessing data in memory is considerably slower than processing directly in the CPU.
- Registers are memory located within the CPU itself where data can be stored and accessed quickly. Normally, the compiler determines what data is to be stored in the registers of the CPU at what times.
- However, the C language provides the storage class register so that the programmer can suggest to the compiler that particular automatic variables should be allocated to CPU registers, if possible and it is not an obligation for the CPU.
- Thus, register variables provide a certain control over efficiency of program execution.
- Variables which are used repeatedly or whose access times are critical may be declared to be of storage class register.
- Variables can be declared register as follows:
  register int var;

4.extern-storage class:-

- All variables we have seen so far have had limited scope (the block in which they are declared) and limited lifetimes (as for automatic variables).
- However, in some applications it may be useful to have data which is accessible from within any block and/or which remains in existence for the entire execution of the program. Such variables are called global variables, and the C language provides storage classes which can meet these requirements; namely, the external (extern) and static (static) classes.
- Declaration for external variable is as follows:
  extern int var;
- External variables may be declared outside any function block in a source code file the same way any other variable is declared; by specifying its type and name (extern keyword may be omitted).
- Typically if declared and defined at the beginning of a source file, the extern keyword can be omitted. If the program is in several source files, and a

variable is defined in let say file1.c and used in file2.c and file3.c then the extern keyword must be used in file2.c and file3.c.

- But, usual practice is to collect extern declarations of variables and functions in a separate header file (.h file) then included by using #include.
- Memory for such variables is allocated when the program begins execution, and remains allocated until the program terminates. For most C implementations, every byte of memory allocated for an external variable is initialized to zero.
- The scope of external variables is global, i.e. the entire source code in the file following the declarations. All functions following the declaration may access the external variable by using its name. However, if a local variable having the same name is declared within a function, references to the name will access the local variable cell.

4. Static-Storage Class:-

- As we have seen, external variables have global scope across the entire program (provided extern declarations are used in files other than where the variable is defined), and have a lifetime over the entire program run.
- Similarly, static storage class provides a lifetime over the entire program, however; it provides a way to limit the scope of such variables, and static storage class is declared with the keyword static as the class specifier when the variable is defined.
- These variables are automatically initialized to zero upon memory allocation just as external variables are. Static storage class can be specified for automatic as well as external variables such as:
  static extern varx;
- Static automatic variables continue to exist even after the block in which they are defined terminates. Thus, the value of a static variable in a function is retained between repeated function calls to the same function.
- The scope of static automatic variables is identical to that of automatic variables, i.e. it is local to the block in which it is defined; however, the storage allocated becomes permanent for the duration of the program.
- Static variables may be initialized in their declarations; however, the initializers must be constant expressions, and initialization is done only once at compile time when memory is allocated for the static variable.

Chapter-6
Pointers, Structure and File handling

Q.1 What is a pointer?

Ans. A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.
The unary or monadic operator & gives the ``address of a variable''.
The indirection or dereference operator * gives the ``contents of an object pointed to by a pointer''. To declare a pointer to a variable do:

int *pointer;

Q.2 What is Indirection?

Ans. If we declare a variable, its name is a direct reference to its value. If we have a pointer to a variable, or any other object in memory, you have an indirect reference to its value. If p is a pointer, the value of p is the address of the object. *p means "apply the indirection operator to p"; its value is the value of the object that p points to. (Some people would read it as "Go indirect on p.")
*p is an lvalue; like a variable, it can go on the left side of an assignment operator, to change the value. If p is a pointer to a constant, *p is not a modifiable lvalue; it can't go on the left side of an assignment.
Consider the following program. It shows that when p points to i, *p can appear wherever i can.

```
#include <stdio.h>
int main()
{
  int i;
  int *p;
  i = 5;
   p = & i;    /* now *p == i */
   printf("i=%d, p=%P, *p=%d\n", i, p, *p);
    *p = 6;     /* same as i = 6 */
   printf("i=%d, p=%P, *p=%d\n", i, p, *p);

   return 0;
}
```

After p points to i (p = &i), you can print i or *p and get the same thing. You can even assign to *p, and the result is the same as if you had assigned to i.

Q.3 What is Null Pointer?

Ans There are times when it's necessary to have a pointer that doesn't point to anything. The macro *NULL*, defined in *<stddef.h>*, has a value that's guaranteed to be different from any valid pointer. *NULL* is a literal zero, possibly cast to void* or char*. Some people, notably C++ programmers, prefer to use *0* rather than *NULL*.

You can't use an integer when a pointer is required. The exception is that a literal zero value can be used as the *null* pointer. (It doesn't have to be a literal zero, but that's the only useful case. Any expression that can be evaluated at compile time, and that is zero, will do. It's not good enough to have an integer variable that might be zero at runtime.)

Q.4 When is a Null Pointer Used?

Ans. The null pointer is used in three ways:

1. To stop indirection in a recursive data structure.

2. As an error value.

3. As a sentinel value.

1. Using a Null Pointer to Stop Indirection or Recursion

Recursion is when one thing is defined in terms of itself. A recursive function calls itself. The following factorial function calls itself and therefore is considered recursive:

```
/* Dumb implementation; should use a loop */
unsigned factorial( unsigned i )
{
    if ( i == 0 || i == 1 )
    {
            return 1;
        }
     else
        {
           return i * factorial( i - 1 );
        }
   }
```

A recursive data structure is defined in terms of itself. The simplest and most common case is a (singularly) linked list. Each element of the list has some value, and a pointer to the next element in the list:

```
   struct string_list
   {
        char    *str;   /* string (in this case) */
     struct string_list      *next;
   };
```

There are also doubly linked lists (which also have a pointer to the preceding element) and trees and hash tables and lots of other neat stuff. You'll find them described in any good book on data structures. You refer to a linked list with a pointer to its first element. That's where the list starts; where does it stop? This is where the null pointer comes in. In the last element in the list, the next field is set to NULL when there is no following element. To visit all the elements in a list, start at the beginning and go indirect on the next pointer as long as it's not null:

```
   while ( p != NULL )
   {
      /* do something with p->str */
    p = p->next;
   }
```

Notice that this technique works even if p starts as the null pointer.

2. Using a Null Pointer As an Error Value

The second way the null pointer can be used is as an error value. Many C functions return a pointer to some object. If so, the common convention is to return a null pointer as an error code:

```
if ( setlocale( cat, loc_p ) == NULL )
{
   /* setlocale() failed; do something */
  /* ... */
}
```

This can be a little confusing. Functions that return pointers almost always return a valid pointer (one that doesn't compare equal to zero) on success, and a null pointer (one that compares equal to zero) pointer on failure. Other functions return an int to show success or failure; typically, zero is success and nonzero is failure. That way, a "true" return value means "do some error handling":

```
if ( raise( sig ) != 0 ) {
        /* raise() failed; do something */
        /* ... */
}
```

The success and failure return values make sense one way for functions that return ints, and another for functions that return pointers. Other functions might return a count on success, and either zero or some negative value on failure. As with taking medicine, you should read the instructions first.

Using a Null Pointer As a Sentinel Value

The third way a null pointer can be used is as a "sentinel" value. A sentinel value is a special value that marks the end of something. For example, in main(), argv is an array of pointers. The last element in the array (argv[argc]) is always a null pointer. That's a good way to run quickly through all the elements:

```
  /* A simple program that prints all its arguments.
It doesn't use argc ("argument count"); instead,  it takes advantage of the fact that
the last value
  in argv ("argument vector") is a null pointer. */
  #include <stdio.h>
  #include <assert.h>
  int
  main( int argc, char **argv)
  {
       int i;
       printf("program name = \"%s\"\n", argv[0]);
  for (i=1; argv[i] != NULL; ++i)
 printf("argv[%d] = \"%s\"\n", i, argv[i]);
   assert(i == argc);
  return 0;
}
```

Q 5. What is a void pointer?

Ans. A void pointer is a C convention for "a raw address." The compiler has no idea what type of object a void pointer "really points to." If you write
int *ip; ip points to an int. If you write
void *p; p doesn't point to a void!
In C and C++, any time you need a void pointer, you can use another pointer type. For example, if you have a char*, you can pass it to a function that expects a void*. You don't even need to cast it. In C (but not in C++), you can use a void* any time you need any kind of pointer, without casting. (In C++, you need to cast it.)

Q6. When is a void pointer used?

Ans. A void pointer is used for working with raw memory or for passing a pointer to an unspecified type.
Some C code operates on raw memory. When C was first invented, character pointers (char *) were used for that. Then people started getting confused about when a character pointer was a string, when it was a character array, and when it was raw memory.
For example, strcpy() is used to copy data from one string to another, and strncpy() is used to copy at most a certain length string to another:
char *strcpy(char *str1, const char *str2);
char *strncpy(char *str1, const char *str2, size_t n);
memcpy() is used to move data from one location to another:
void *memcpy(void *addr1, void *addr2, size_t n);
void pointers are used to mean that this is raw memory being copied. NUL characters (zero bytes) aren't significant, and just about anything can be copied. Consider the following code:

```
#include "thingie.h"    /* defines struct thingie */
struct thingie  *p_src, *p_dest;
/* ... */
memcpy( p_dest, p_src, sizeof( struct thingie) * numThingies );
```

This program is manipulating some sort of object stored in a struct thingie. p1 and p2 point to arrays, or parts of arrays, of struct thingies. The program wants to copy numThingies of these, starting at the one pointed to by p_src, to the part of the array beginning at the element pointed to by p_dest. memcpy() treats p_src and p_dest as pointers to raw memory; sizeof(struct thingie) * numThingies is the number of bytes to be copied.

Q7. Can we subtract pointers from each other? Why?

Ans. If you have two pointers into the same array, you can subtract them. The answer is the number of elements between the two elements.

Consider the street address analogy presented in the introduction of this chapter. Say that I live at 118 Fifth Avenue and that my neighbor lives at 124 Fifth Avenue. The "size of a house" is two (on my side of the street, sequential even numbers are used), so my neighbor is (124-118)/2 (or 3) houses up from me. (There are two houses between us, 120 and 122; my neighbor is the third.) You might do this subtraction if you're going back and forth between indices and pointers.

You might also do it if you're doing a binary search. If p points to an element that's before what you're looking for, and q points to an element that's after it, then (q-p)/2+p points to an element between p and q. If that element is before what you want, look between it and q. If it's after what you want, look between p and it.

(If it's what you're looking for, stop looking.)

You can't subtract arbitrary pointers and get meaningful answers. Someone might live at 110 Main Street, but I can't subtract 110 Main from 118 Fifth (and divide by 2) and say that he or she is four houses away!

If each block starts a new hundred, I can't even subtract 120 Fifth Avenue from 204 Fifth Avenue. They're on the same street, but in different blocks of houses (different arrays).

C won't stop you from subtracting pointers inappropriately. It won't cut you any slack, though, if you use the meaningless answer in a way that might get you into trouble.

When you subtract pointers, you get a value of some integer type. The ANSI C standard defines a typedef, ptrdiff_t, for this type. (It's in <stddef.h>.) Different compilers might use different types (int or long or whatever), but they all define ptrdiff_t appropriately.

Below is a simple program that demonstrates this point. The program has an array of structures, each 16 bytes long. The difference between array[0] and array[8] is 8 when you subtract struct stuff pointers, but 128 (hex 0x80) when you cast the pointers to raw addresses and then subtract.

If you subtract 8 from a pointer to array[8], you don't get something 8 bytes earlier; you get something 8 elements earlier.

```
#include <stdio.h>
#include <stddef.h>
struct stuff {
        char    name[16];
        /* other stuff could go here, too */
};
struct stuff array[] = {
        { "The" },
        { "quick" },
        { "brown" },
        { "fox" },
        { "jumped" },
        { "over" },
        { "the" },
        { "lazy" },
        { "dog." },
        { "" }
};
int main()
{
        struct stuff    *p0 = & array[0];
        struct stuff    *p8 = & array[8];
        ptrdiff_t       diff = p8 - p0;
        ptrdiff_t       addr_diff = (char*) p8 - (char*) p0;
        /* cast the struct stuff pointers to void* */
        printf("& array[0] = p0 = %P\n", (void*) p0);
        printf("& array[8] = p8 = %P\n", (void*) p8);
        /* cast the ptrdiff_t's to long's
        (which we know printf() can handle) */
        printf("The difference of pointers is %ld\n",
          (long) diff);
        printf("The difference of addresses is %ld\n",
          (long) addr_diff);
        printf("p8 - 8 = %P\n", (void*) (p8 - 8));

        printf("p0 + 8 = %P (same as p8)\n", (void*) (p0 + 8));
        return 0;
}
```

Q.8. Is NULL always defined as 0(zero)?

Ans. NULL is defined as either 0 or (void*)0. These values are almost identical; either a literal zero or a void pointer is converted automatically to any kind of pointer, as necessary, whenever a pointer is needed (although the compiler can't always tell when a pointer is needed).

Q9. Is NULL always equal to 0(zero)?

Ans The answer depends on what you mean by "equal to." If you mean "compares equal to," such as

```
if ( /* ... */ )
{
     p = NULL;
}
else
{
     p = /* something else */;
}
/* ... */
if ( p == 0 )
```

then yes, *NULL* is always equal to *0*. That's the whole point of the definition of a *null* pointer.

If you mean "is stored the same way as an integer zero," the answer is no, not necessarily. That's the most common way to store a null pointer. On some machines, a different representation is used.

The only way you're likely to tell that a null pointer isn't stored the same way as zero is by displaying a pointer in a debugger, or printing it. (If you cast a null pointer to an integer type, that might also show a nonzero value.)

Q10. What does it mean when a pointer is used in an if statement?

Ans. Any time a pointer is used as a condition, it means "Is this a non-null pointer?" A pointer can be used in an if, *while*, *for*, or *do*/*while* statement, or in a conditional expression. It sounds a little complicated, but it's not.

Take this simple case:

```
if ( p )
{
     /* do something */
}
else
{
     /* do something else */
}
```

An *if* statement does the "then" (first) part when its expression compares unequal to zero. That is,

if (/ something */)*

is always exactly the same as this:

if (/ something */ != 0)*

That means the previous simple example is the same thing as this:

```
if ( p != 0 )
{
     /* do something (not a null pointer) */
}
else
{
 /* do something else (a null pointer) */
}
```

This style of coding is a little obscure. It's very common in existing C code; you don't have to write code that way, but you need to recognize such code when you see it.

Q. 11. Can you add pointers together? Why would you?

Ans No, you can't add pointers together. If you live at 1332 Lakeview Drive, and your neighbor lives at 1364 Lakeview, what's 1332+1364? It's a number, but it doesn't mean anything. If you try to perform this type of calculation with pointers in a C program, your compiler will complain.

The only time the addition of pointers might come up is if you try to add a pointer and the difference of two pointers:

p = p + p2 - p1;

which is the same thing as this:

p = (p + p2) - p1.

Here's a correct way of saying this:

p = p + (p2 - p1);

Or even better in this case would be this example:

p += p2 - p1;

Q.12. How do you use a pointer to a function?

Ans. The hardest part about using a pointer-to-function is declaring it. Consider an example. You want to create a pointer, pf, that points to the strcmp() function. The strcmp() function is declared in this way:

```
int strcmp( const char *, const char * )
```

To set up pf to point to the strcmp() function, you want a declaration that looks just like the strcmp() function's declaration, but that has *pf rather than strcmp:

```
int (*pf)( const char *, const char * );
```

Notice that you need to put parentheses around *pf. If you don't include parentheses, as in

```
int *pf( const char *, const char * ); /* wrong */
```

you'll get the same thing as this:

```
(int *) pf( const char *, const char * ); /* wrong */
```

That is, you'll have a declaration of a function that returns int*.

After you've gotten the declaration of pf, you can #include <string.h> and assign the address of strcmp() to pf:

```
pf = strcmp;
```

or

```
pf = & strcmp; /* redundant & */
```

You don't need to go indirect on pf to call it:

```
if ( pf( str1, str2 ) > 0 ) /* ... */
```