35. A BASIC INTRODUCTION TO ‘C’

35.1 WHY USE ‘C’?

- ‘C’ is commonly used to produce operating systems and commercial software. Some examples of these are UNIX, Lotus-123, dBase, and some ‘C’ compilers.

- *Machine Portable*, which means that it requires only small changes to run on other computers.

- *Very Fast*, almost as fast as assembler.

- Emphasizes *structured programming*, by focusing on functions and subroutines.

- You may easily customize ’C’ to your own needs.

- Suited to Large and Complex Programs.

• Very Flexible, allows you to create your own functions.

35.2 BACKGROUND

• Developed at Bell Laboratories in the Early 70’s, and commercial compilers became available in the Late 70’s.Recnetly has become more popular because of its ties to UNIX. Over 90% of UNIX is written in ‘C’. AT&T originally developed ‘C’ with the intention of making it an in-house standard.

35.3 PROGRAM PARTS

• /* is the start of a comment.

• */ is the end of comment.

• The main program is treated like a function, and thus it has the name *main().*

• lower/UPPER case is crucial, and can never be ignored.

• Statements are separated by semi-colons <u>‘;’</u>

• Statements consist of one operation, or a set of statements between curly brackets *<u>{, }</u>*

• There are no line numbers.

```
Program to Add two Numbers:
/* A simple program to add two numbers and print the results */

main()
{
int x, y = 2, z; /* define three variables and give one a value */
x = 3; /* give another variable a value */
z = x + y; /* add the two variables */
printf(“%d + %d = %d\n”, x, y, z); /*print the results */
}

Results (output):
```

• lines may be of any length.

• A very common function in ‘C’ is *<u>printf()</u>*. This function will do a formatted print. The format is the first thing which appears between the brackets. In this case the format says print an integer *<u>%d</u>* followed by a space then a ‘+’ then another space, another integer, another space, ‘=’, and another space, another integer, then a line feed ‘*<u>\n</u>*’. All variables that follow the format statement are those to be printed. *<u>x, y</u>*, and *<u>z</u>* are the three integers to be printed, in their respective orders.

• Major Data Types for variables and functions are (for IBM PC):

int (2 byte integer),
short (1 byte integer),
long (4 byte integer),
char (1 byte integer),
float (4 byte IEEE floating point standard),
double (8 byte IEEE floating point standard).

•*int, short, long, char* can be modified by the addition of *unsigned*, and *register*. An *unsigned integer* will not use 1 bit for number sign. A *register* variable will use a data register in the microprocessor, if possible, and it will speed things up (this is only available for integers).

Example of Defining Different Data Types:

```
main()
{
unsigned int i;
register j;
short k;
char l;
double m;
etc
```

• A *function* consists of a sub-routine or program, which has been assigned a name. This function is capable of accepting an argument list, and returning a single value. The function must be defined before it is called from within the program. (e.g. *sin()* and *read()*).

Program to add numbers with a function:

```
/* A simple program to add two numbers and print the results */

int add(); /* Declare a integer function called 'add' */

main()
{
int x = 3, y = 2, z; /* define three variables and give values */
z = add(x, y); /* pass the two values to 'add' and get the sum*/
printf("%d + %d = %d\n", x, y, z); /*print the results */
}

int add(a, b) /* define function and variable list */
int a, b; /* describe types of variable lists */
{
int c; /* define a work integer */
c = a + b; /* add the numbers */
return(c); /* Return the number to the calling program */
```

• Every variable has a *scope*. This determines which functions are able to use that variable. If a variable is *global*, then it may be used by any function. These can be modified by the addition of *static*, *extern* and *auto*. If a variable is defined in a function, then it will be local to that function, and is not used by any other function. If the variable needs to be initialized every time the subroutine is called, this is an *auto* type. *static* variables can be used for a variable that must keep the value it had the last time the function was called. Using *extern* will allow the variable types from other parts of the program to be used in a function.

```
Program example using global variables:
/* A simple program to add two numbers and print the results */

int x = 3, /* Define global x and y values */
y = 2,
add(); /* Declare an integer function called 'add' */

main()
{
printf("%d + %d = %d\n", x, y, add()); /*print the results */
}

int add() /* define function */
{
return(x + y); /* Return the sumto the calling program */
```

• Other variable types of variables are ***union, enum, struct***, etc.

• Some basic control flow statements are ***while(), do-while(), for(), switch(),*** and ***if()***. A couple of example programs are given below which demonstrate all the 'C' flow statements.

```
Program example with a for loop:
/* A simple program toprint numbers from 1 to 5*/

main()
{
int i;
for(i = 1; i <= 5; i = i + 1){
printf("number %d \n", i); /*print the number */
}
}
```

or example with a while loop:

```
main()
{
int i = 1;
while(i <= 5){
printf("number %d \n", i);
i = i + 1;
}
}
```

or example with a do while loop

```
main()
{
int i = 1;
do{
printf("number %d \n", i);
i = i + 1;
}while(i <= 5)
}
```

or example with a do until loop:

```
main()
{
int i = 1;
do{
printf("number %d \n", i);
i = i + 1;
}until(i > 5)
}
```

Example Program with an if else:

```
main()
{
int x = 2, y = 3;
if(x > y){
printf("Maximum is %d \n", x);
} else if(y > x){
printf("Maximum is %d \n", y);
} else {
printf("Both values are %d \n", x);
}
}
```

Example Program using switch-case:

```
main()
{
int x = 3; /* Number of People in Family */
switch(x){ /* choose the numerical switch */
case 0: /* Nobody */
printf("There is no family \n");
break;
case 1: /* Only one person, but a start */
printf("There is one parent\n");
break;
case 2: /* You need two to start something */
printf("There are two parents\n");
break;
default: /* critical mass */
printf("There are two parents and %d kids\n", x-2);
break;
}
}
```

• *#include <filename.h>* will insert the file named *filename.h* into the program. The **.h* extension is used to indicate a header file which contains 'C' code to define functions and constants. This almost always includes "*stdio.h*". As we saw before, a function must be defined (as with the 'add' function). We did not define *printf()* before we used it, this is normally done by using *#include <stdio.h>* at the top of your programs. "*stdio.h*" contains a line which says '*int printf();*'. If we needed to use a math function like *y = sin(x)* we would have to also use *#include <math.h>*, or else the compiler would not know what type of value that *sin()* is supposed to return.

•*#define CONSTANT TEXT* will do a direct replacement of *CONSTANT* in the program with *TEXT*, before compilation. *#undef CONSTANT* will undefine the *CONSTANT*.

A Sample Program to Print Some sin() values
(using defined constatnts)

```
#include "stdio.h"
#include "math.h"
#define TWO_PI 6.283185307
#define STEPS 5

main()
{
double x; /* Current x value*/

for(x = 0.0; x <= TWO_PI; x = x + (TWO_PI / STEPS)){
printf("%f = sin(%f) \n", sin(x), x);
}
}
```

• *#ifdef, #ifndef, #if, #else* and *#else* can be used to conditionally include parts of a program. This is use for including and eliminating debugging lines in a program.

• *#define, #include, #ifdef, #ifndef, #if, #else, /** and **/* are all handled by the Preprocessor, before the compiler touches the program.

• Matrices are defined as shown in the example. In ‘C’ there are no limits to the matrix size, or dimensions. Arrays may be any data type. Strings are stored as arrays of characters.

• *i++* is the same as *i = i + 1.*

A Sample Program to Get a String
Then Print its ASCII Values (with matrix)

```
#include “stdio.h”
#define STRING_LENGTH 5

main()
{
int i;
char string[STRING_LENGTH]; /* character array */
gets(string); /* Input string from keyboard */
for(i = 0; i < STRING_LENGTH; i++){
printf(“pos %d, char %c, ASCII %d \n”, i, string[i], string[i]);
}
}
```

```
INPUT:
HUGH<return>

OUTPUT:
pos 0, char H, ASCII 72
pos 0, char U, ASCII 85
pos 0, char G, ASCII 71
pos 0, char H, ASCII 72
pos 0, char , ASCII 0
```

• Pointers are a very unique feature of ‘C’. First recall that each variable uses a real location in memory. The computer remembers where the location of that variable is, this memory of location is called a pointer. This pointer is always hidden from the programmer, and uses it only in the background. In ‘C’, the pointer to a variable may be used. We may use some of the operations of ‘C’ to get the variable that the pointer, points to. This allows us to deal with variables in a very powerful way.

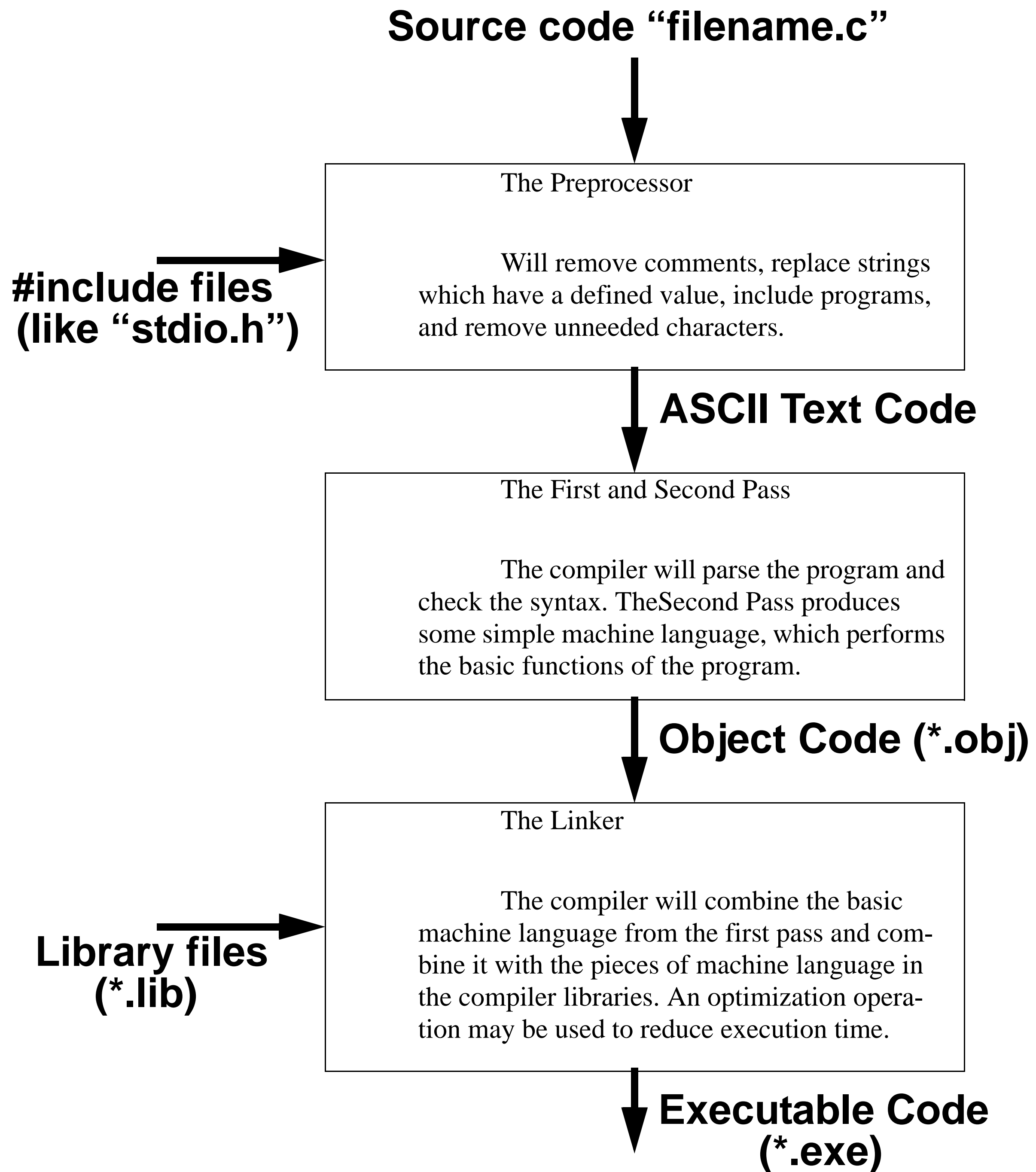
A Sample Program to Get a String
Then Print its ASCII Values (with pointers):

```
#include “stdio.h”

main()
{
int i;
char *string; /* character pointer */
gets(string); /* Input string from keyboard */
for(i = 0; string[i] != 0; i++){
printf(“ pos %d, char %c, ASCII %d \n”, i, string[i], string[i]);
}
}

INPUT:
HUGH<return>

OUTPUT:
pos 0, char H, ASCII 72
pos 0, char U, ASCII 85
pos 0, char G, ASCII 71
pos 0, char H, ASCII 72
```

35.4 HOW A ‘C’ COMPILER WORKS

• A ‘C’ compiler has three basic components: Preprocessor, First and Second Pass

Compiler, and Linker.



35.5 STRUCTURED 'C' CODE

- A key to well designed and understandable programs.

- Use indents, spaces and blank lines, to make the program look less cluttered, and give it a block style.

- Comments are essential to clarify various program parts.

- Descriptive variable names, and defined constants make the purpose of the variable obvious.

- All declarations for the program should be made at the top of the program listing.

A Sample of a Bad Program Structure:

```
main(){int i;for(;i<10;i++)printf("age:%d\n",i);}
```

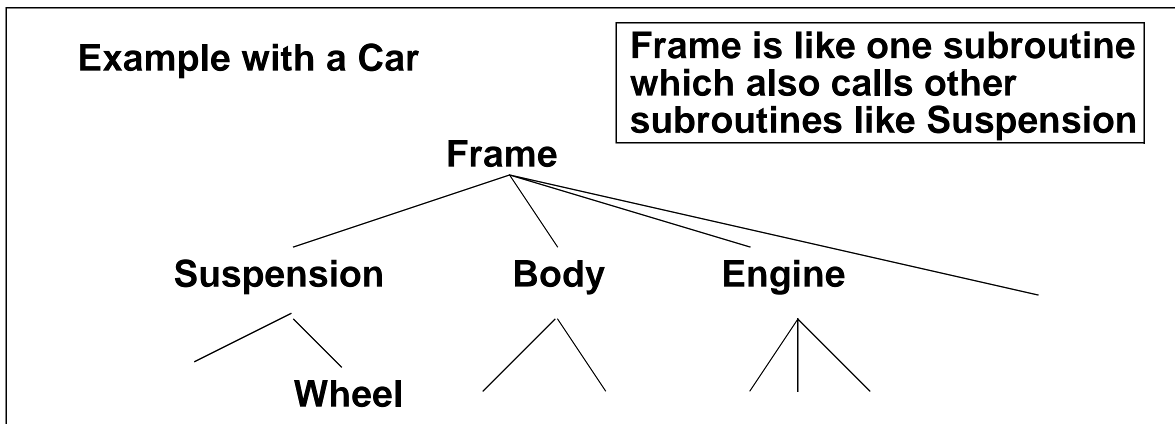
A Good Example of the same Program:

```
#include <stdio.h>
#define COUNT 10 /* Number of counts in loop */

main()
{
int i; /* counter */
        for(i = 0; i < COUNT; i++){ /* loop to print numbers */
        printf("age:%d\n", i);
        }
        exit(0);
}
```

35.6 ARCHITECTURE OF 'C' PROGRAMS (TOP-DOWN)

35.6.1 How?

• A program should be broken into fundamental parts (using functions for each part) and then assembled using functions. Each function consists of programs written using the previous simpler functions.



• A Clear division should be maintained between program levels.

• Never use goto's, they are a major source of logic errors. Functions are much easier to use, once written.

• Try to isolate machine specific commands (like graphics) into a few functions.

35.6.2 Why?

• A top-down design allows modules to be tested as they are completed. It is much easier to find an error in a few lines of code, than in a complete program.

• When programs are complete, errors tend to be associated with modules, and are thus much easier to locate.

• Updates to programs are much easier, when we only need to change one function.

• It is just as easy to change the overall flow of a program, as it is to change a function.

Application of ‘C’ to a CAD Program

35.7 CREATING TOP DOWN PROGRAMS

1. *Define Objectives* - Make a written description of what the program is expected to do.

2. *Define Problem* - Write out the relevant theory. This description should include variables, calculations and figures, which are necessary for a complete solution to the problem. From this we make a list of required data (inputs) and necessary results (output).

3. *Design User Interface* - The layout of the screen(s) must be done on paper. The method of data entry must also be considered. User options and help are also considered here. *(There are numerous factors to be considered at this stage, as outlined in the course notes.)*

4. *Write Flow Program* - This is the main code that decides when general operations occur. This is the most abstract part of the program, and is written calling dummy ‘program stubs’.

5. *Expand Program* - The dummy ‘stubs’ are now individually written as functions. These functions will call another set of dummy ‘program stubs’. This continues until all of the stubs are completed. *After the completion of any new function, the program is compiled, tested and debugged.*

6. *Testing and Debugging*- The program operation is tested, and checked to make sure that it meets the objectives. If any bugs are encountered, then the program is revised, and then retested.

7. *Document* - At this stage, the operation of the program is formally described. For Programmers, a top-down diagram can be drawn, and a written description of functions should also be given.

Golden Rule: If you are unsure how to proceed when writing a program, then work out the problem on paper, before you commit yourself to your programmed solution.

Note: Always consider the basic elements of Software Engineering, as outlined in the ES488 course notes.

35.8 HOW THE BEAMCAD PROGRAM WAS DESIGNED

35.8.1 Objectives:

• The program is expected to aid the design of beams by taking basic information about beam geometry and material, and then providing immediate feedback. The beam will be simply supported, and be under a single point load. The program should also provide a printed report on the beam.

35.8.2 Problem Definition:

• The basic theory for beam design is available in any good mechanical design textbook. In this example it will not be given.

• The inputs were determined to be few in number: Beam Type, Beam Material, Beam Thickness, Beam Width, Beam Height, Beam Length, Load Position, Load Force.

• The possible outputs are Cross Section Area, Weight, Axial Stiffness, Bending Stiffness, and Beam Deflection, a visual display of Beam Geometry, a display of Beam Deflection.

35.8.3 User Interface:

35.8.3.1 - Screen Layout (also see figure):

• The small number of inputs and outputs could all be displayed, and updated, on a single screen.

• The left side of the screen was for inputs, the right side for outputs.

• The screen is divided into regions for input(2), input display and prompts(1), Beam Cross section(3), Numerical Results(4), and Beam Deflection(5).

35.8.3.2 - Input:

• Current Inputs were indicated by placing a box around the item on the display(1).

• In a separate Prompt Box(2), this input could be made.

• The cursor keys could be used to cursor the input selector up or down.

• Single keystroke operation.

• Keys required: UP/DOWN Cursors, F1, F2, F4, numbers from ‘0’ to ‘9’, ‘.’, ‘-’,

and <RETURN>. In the spirit of robustness it was decided to screen all other keys.

35.8.3.3 - Output:

• Equations, calculations, material types, and other relevant information were obtained from a text.

• Proper textual descriptions were used to ensure clarity for the user.

• For a printed report, screen information would be printed to a printer, with the prompt area replaced with the date and time.

35.8.3.4 - Help:

• A special set of help information was needed. It was decided to ensure that the screen always displays all information necessary(2).

35.8.3.5 - Error Checking:

• Reject any input which violates the input limits.

• A default design was given, which the user could modify.

- An error checking program was created, which gives error messages.

35.8.3.6 - Miscellaneous:

- The screen was expressed in normalized coordinates by most sub-routines.

- Colors were used to draw attention, and highlight areas.

35.8.4 Flow Program:

```
main()
/*
 * EXECUTIVE CONTROL LEVEL
 *
 * This is the main terminal point between the
 * various stages of setup, input, revision
 * and termination.
 *
 * January 29th, 1989.
 */
{
        static int error;

        if((error = setup()) != ERROR) {
                        screen(NEW);
                        screen(UPDATE);
                        while((error = input()) != DONE) {
                                if(error == REVISED) {

screen(NEW);

screen(UPDATE);
                                }
                        }
                        error = NO_ERROR;
        }
        kill();
        if(error == ERROR) {
                        printf("EGA Graphics Driver Not Installed");


                }
```

35.8.5 Expand Program:

• The routines were written in a top down fashion, in a time of about 30 hours. These routines are listed below.

Routines Used In Package:

• **main()** - to be used as the main program junction.

• **setup()** - to set up graphics mode and printer.

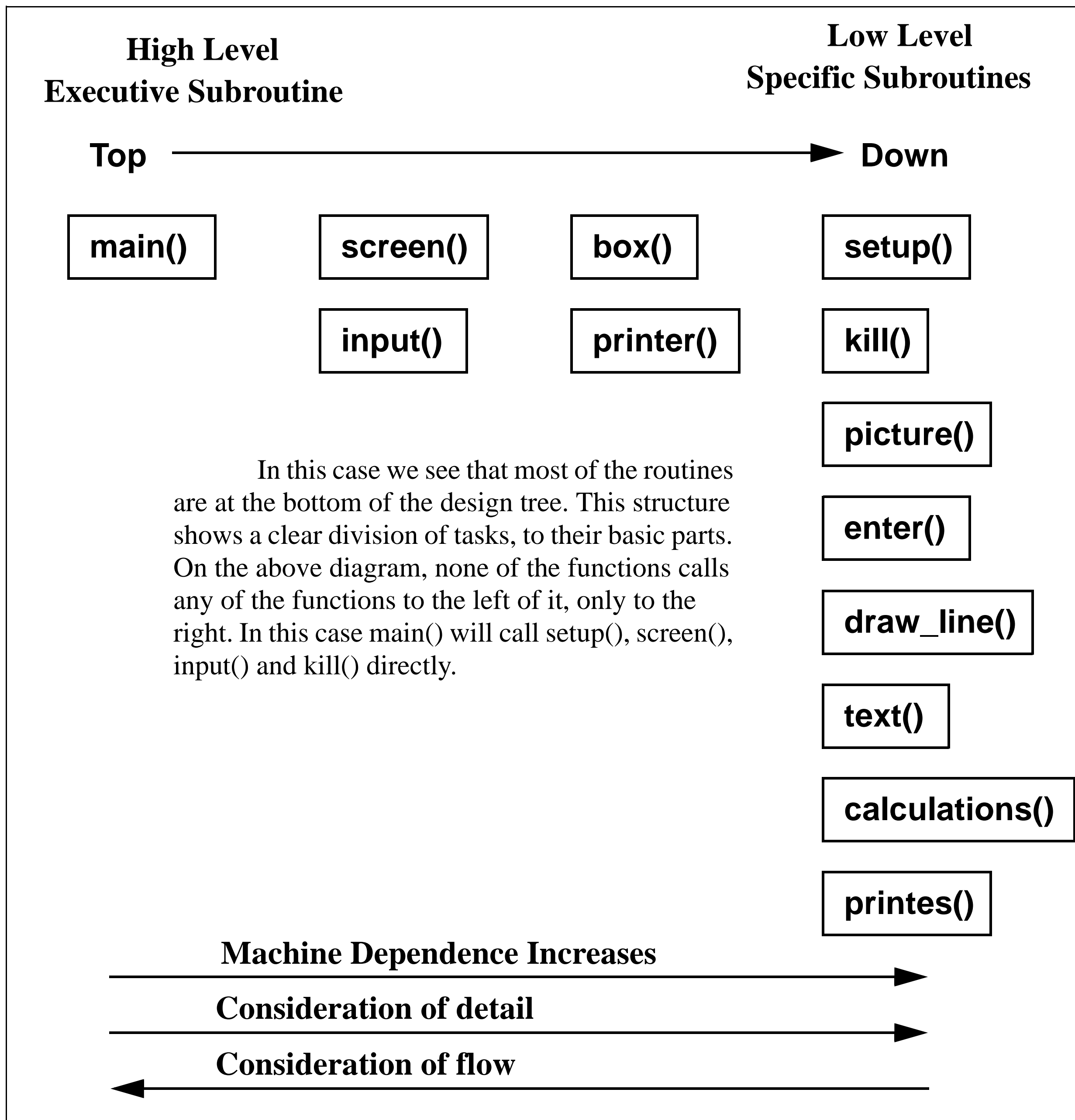
• **screen()** - A function to draw, or refresh part of the screen. In the interest of program speed, this function uses some low level commands.

• **calculations()** - perform the calculations of outputs from the inputs

• **picture()** - draws the beam cross section and deflection of beam. For the sake of speed, this section will use low level commands.

• **input()** - A function which controls the main input loop for numbers, con-

• Condition and error flags were used to skip unnecessary operations, and thus speed up response. A response of more than 0.5 seconds will result in loss of attention by the user.



In this case we see that most of the routines are at the bottom of the design tree. This structure shows a clear division of tasks, to their basic parts. On the above diagram, none of the functions calls any of the functions to the left of it, only to the right. In this case main() will call setup(), screen(), input() and kill() directly.

35.8.6 Testing and Debugging:

• The testing and debugging was very fast, with only realignment of graphics being required. This took a couple of hours.

35.8.7 Documentation

35.8.7.1 - Users Manual:

• The documentation included an Executive Summary of what the Program does.

• The Objectives of the program were described.

• The theory for beam design was given for the reference of any program user, who wanted to verify the theory, and possible use it.

• A manual was given which described key layouts, screen layout, basic sequence of operations, inputs and outputs.

• Program Specifications were also given.

• A walk through manual was given. This allowed the user to follow an example which displayed all aspects of the program.

35.8.7.2 - Programmers Manual:

- Design Strategy was outlined and given.

- A complete program listing was given (with complete comments).

- Complete production of this Documentation took about 6 hours.

35.8.8 Listing of BeamCAD Program.

- Written for turbo ‘C’

35.9 PRACTICE PROBLEMS

1. What are the basic components of a ‘C’ compiler, and what do they do?

2. You have been asked to design a CAD program which will choose a bolt and a nut to hold two pieces of sheet metal together. Each piece of sheet metal will have a hole drilled in it that is the size of the screw. You are required to consider that the two pieces are experiencing a single force. State your assumptions about the problem, then describe how you would produce this program with a Top Down design.

3. What are some reasons for using ‘C’ as a programming language?

4. Describe some of the reasons for Using Top-Down Design, and how to do it.