15 Traits of Turbo C

"Everyone who asks will receive."

In the First Chapter itself I told you that Turbo C++3.0 is the IDE that is used throughout this book. If you've got Turbo C 2.0 or latter version of Turbo C, please get version 3.0. Why I prefer Version 3.0 is, it is being helpful to explain DOS programming than any other versions.

15.1 Features of TC++3.0

- Syntax highlighting
- Supports C++'s single line comment (//) even for C codes
- More options
- Can execute inline assembly without any overhead.

15.2 Configure your TC++3.0

If you change the default configuration (color, tab etc) of TC++3.0, it is enough to delete the file `TCCONFIG.TC` that is found on the TC directory to get back default configuration.

- Set the default extension to C by `Options > Editor > Extension > C`
- Set tab size to 8 by `Options > Editor > Tab > 8`

15.3 IDE basics

Tool	Input	Output
Compiler	.c	.asm
Assembler	.asm	.obj or .lib
Linker	.obj & .lib	.exe or .com

IDE is nothing but Integrated Development Environment. IDE has got so many components. The most important components among them are Editor, Compiler, Assembler & linker.

First of all we should know the difference between Editor, Compiler, Assembler & linker. Editor is the one in which we create, read & edit our texts. Compiler is the one, which converts C files (.c) to Assembly (.asm) files. Compiler is very often treated as language converter. Assembler is the one, which converts assembly (.asm) files into object (.obj) files or (.lib) files. Linker is the one that links object (.obj) files and library (.lib) files and thus creates an executable file (.exe or .com).

Compiler, Assembler & Linker are usually command line executable files, which requires filename(s) and other information as parameters. What IDE does is, it saves our time by invoking the proper utilities with proper parameters within the Editor.

15.4 Useful Utilities

You have many useful utilities to use with TC++3.0. These useful utilities are rarely known in India. Please try to use them for better programming! I will just introduce the utilities. For more explanations about those utilities, see the documentation (found on TC directory).

15.4.1 BASM

BASM is Built-in inline Assembler. It is used to assemble the inline assembly to the C file.

15.4.2 TASM

BASM is not much efficient. It can handle only x286 instructions. TASM (Turbo Assembler) can handle x386 instructions. x386 instructions are efficient compared to x286 instructions. So real programmers use TASM than BASM.

In the beginning of the program you have to add the following line to invoke TASM.

```
#pragma inline
```

Otherwise the default BASM will be called.

Note
Even in TASM, the default instruction sets are x286. To call x386 instruction, you have to add .386. We will see this later!

15.4.3 TLINK

TLINK is used to link object files and library files and produces the executable file.

15.4.4 TLIB

Turbo library or TLIB is useful to manage, create library files.

15.4.5 MAKE

MAKE file seems to be like a batch file. Real programmers very often use this useful utility.

15.4.6 TCC

TCC is a command line compiler. It is an integrated compiler. Using this you can create assembly files, object files, and you can also create executable files directly.

15.5 main( )

In contradict to ANSI C, Turbo C supports three arguments: `argc, argv & env`. `argc` holds number of arguments passed in command line. `argv` is the array of pointer to the string in command line. Under 3.X versions of DOS, `argv[0]` points to the full path name of the program (e.g., `C:\WAR\CHKMAIN.EXE`). Under versions of DOS before 3.0, `argv[0]` points to null string. `argv[1]` points to first string typed on command line after the program name. `argv[argc]` contains `NULL`. `env` is an array of pointers to the string of environment variables.

Let's see an example:

```
/* chkmain.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[], char *env[])
{
   int i;
   printf("argc = %d \n", argc);
   for( i=0; i<=argc; ++i)
      printf("argv["%d"] = %s \n", i, argc[i]);
   for( i=0; env[i] != NULL; ++i)
      printf("env["%d"] = %s \n", i, env[i]);
   return(0);
}
```

Input & Output
C:\WAR>CHKMAIN argument1 "second argument" 3 "last argument"

See `argv[2]` and `arg[4]`. In order to embed blanks we have put it in double quotes. Turbo C sends all the three arguments (`argc, argv, env`) to its programs. But using the third argument `env` is not a standard way. For standard programming use `environ`.

15.5.1 int `main( )` or void `main( )` ?

Turbo C accepts both `int` and `void main( )` and Turbo C programmers use both `int` and `void main( )` in their programs. But in my opinion, `void main( )` is not a standard usage. The reason is, whenever a program gets executed it returns an integer to the operating system. If it returns '0' means, the program is executed successfully. Otherwise it means the program has been terminated with error.

Using a sample program, I have found that `void main( )` returns 20 *even* after successful completion of program (which means it returns wrong status to the operating system!).

```
/* intmain0.c */
int main( void )
{
   printf( "int main returns 0 \n" );
```

```
   return(0);
} /*--main( )-----*/


/* intmain5.c */
int main( void )
{
   printf( "int main returns 5 \n" );
   return(5);
} /*--main( )------*/


/* voidmain.c */
void main( void )
{
   printf( "void main returns? \n" );
} /*--main( )----*/


@ECHO OFF
REM *** Batch file to check return code (Testmain.bat) ***
   CLS
   intmain0.exe
   ECHO %errorlevel%
   intmain5.exe
   ECHO %errorlevel%
   voidmain.exe
   ECHO %errorlevel%
REM *** end ***
@ECHO ON
```

Note

As I am working on Windows NT, I used %errrorlevel% in a batch file. In other platforms, it may not work. You may have to try different techniques to display the "errorlevel".

After compiling all the C files to exe files, test the return values with `TESTMAIN.BAT`. It shows the error value or status.

Thus we have found that `int main( )` is the appropriate usage.

Note

However void main( ) will be useful in certain circumstances like programming for embedded systems & real time operating system, because there is no place to return the status value. We will see those things later!

We can also get status of `main( )` by using the menu option `COMPILE>Information…` from IDE without using BATCH file.

15.6 Preprocessor

Preprocessor performs macro substitutions, conditional compilation and inclusion of named files. All these are done with controls like: `#define, #if, #ifdef, #ifndef,`

`#elif, #else, #line, #error, #pragma, #include`. We've got several predefined identifiers and macros that expand to produce special information (`__LINE__, __FILE__, __DATE__, __TINY__, etc`)

15.7 Header file

The *costly mistake* very often performed by Indian Programmers is to write all functions in the header (.h) file and to include it in main. Actually header files are those that contain `#defines` and function prototype declarations.

The following demonstration explains why writing functions in header and including it in the main program is wrong.

```
/*  Badhead.h   */
static void PrintHello( void )
{
   printf( "Hello! \n" );
} /*--PrintHello( )-----*/

/* chkhead.c */
#include "badhead.h"
int main( void )
{
   PrintHello( );
   return(0);
} /*--main( )-----*/
```

Input & Output

```
C:>CHKHEAD
Hello!
```

When we include the `Badhead.h` file in `chkhead.c`, file gets expanded. And so it prints the message "Hello!", which is wrong according to the definition of static functions. K&R page-83 says, "*If a function is declared static, however, its name is invisible outside of the file in which it is declared*".

Now let's see the right declaration of a header file.

```
/*   Head.h  */
#ifndef __HEAD_H  /* OR if !define(__HEAD_H) */
#define TRUE  ( 1 )
#define FALSE ( 0 )
typedef int BOOLEAN;

void PrintHello1( void );
void PrintHello2( void );

#endif
```

If `head.h` file is included in our program, the compile time variable `__HEAD_H` will be created. We can use it as a flag to check whether the file is already included or not.

The `#ifndef __HEAD_H` or `#if !defined(__HEAD_H)` helps us to avoid multiple inclusion error. That is, if we don't use the above preprocessor control line and if we include head.h more than one time in our program, we will get error. Now you would ask me where to write the function `PrintHello1( )` and `PrintHello2( )`. Yes, you have to write them in a separate file and you have to create a library file or object file.

15.8 Pragma

`#pragma` is used to control the compiler.

15.8.1 Example 1

```
#pragma inline
```

tells the compiler that the C file contains inline assembly and the compiler will use TASM to assemble the inline codes.

15.8.2 Example 2

Sometimes we write code that will be specific to memory models. In such a case our code must be compiled in that memory model only (We have 6 different memory models: Tiny, Small, Medium, Compact, Large and Huge). So programmers use conditional compilation method.

That is,

```
#ifndef __SMALL__       /* or #if !defined(__SMALL__) */
   #error compile with small memory model
#elif
   :
   :
   /* Program Codes */
#endif
```

There is of course a simple method to do this. That is to use pragma and to force the compiler to compile in specified memory model.

That is,

```
#pragma -ms    /* forces compiler to compile in small memory
model */
   :
   :
   /* Program Codes */
   :
```

15.9 Creating library file

Creating a library(.lib) file is the easiest one. Let's see one example.

```
/* chklib.c */
void PrintHello1( void )
{
   printf( "Hello1" );
} /*--PrintHello1( )------*/

void PrintHello2( void )
{
   printf( "Hello2" );
} /*--PrintHello2( )------*/
```

Now choose `OPTIONS>Applications...>Library`. Then Press F9 to compiler. Now you will get `chklib.lib`.

Creating library file is a good way to organize your program. You can put all the interrelated functions (say mouse functions) in a library file and then you can link the library file whenever necessary.

(e.g.) `tcc mylib.lib foo.c`

Attention! you cannot link the library file that is created in one memory model with another file that is created in another model. So it is advisable to create library file for each memory model.

(e.g.) `mouset.lib` (for Tiny), `mouses.lib` (for Small)

If you write a effective library file, you can sell it without the source code! (Only a narrow-minded people do that!)

15.10 Creating a project file

I already pointed out that it is enough to have OBJ or LIB file to create an EXE file. Project file allows you to organize these files.

Let's see how to create project file. Choose `PROJECT>OPEN` and enter the project name. Now you will get a project window. Press [`Insert`] to add file. Add the respective OBJ, LIB and C files. Now click [`Done`] and press F9 to compile the project file. You will get the EXE file. When you create project file, you should note that more than one file should not have `main( )`.

The applications of these ideas are dealt in forthcoming chapters.

15.11 Turbo C keywords

Along with ANSI C keywords, Turbo C got the following keywords:

```
near    far       huge         cdecl
asm     passed    interrupt
_es     _ds       _cs          _ss
```

When you set the compiler to ANSI standard, you can use the above keywords as *identifiers*.

15.12 Bugs & Remedy

15.12.1 system( )

People who use `system( )` function may have noticed that it won't work when run from IDE. The reason is IDE reserves memory for its own use and there won't be enough memory. But when you run the corresponding EXE file in command line it will work properly. Let's see it with a real program.

```
int main( void )
{
    int err;
    err = system( "DIR" );
    if ( err == -1 )
            perror( "Error: " );
    return(0);
} /*--main( )------*/
```

If you run the above program from IDE, you will get the following message:

```
Error: Not enough memory
```

So running only the EXE file of respective program in DOS Box will be the remedy.

15.12.2 delay( )

The `delay( )` function found in dos.h is processor dependent. And it won't work on all systems. The reason is the delay function is implemented with clock speed.

15.12.2.1 Solution with BIOS tick

An easy solution for this is to implement our own delay with the help of BIOS tick as:

```
/* PC bios data area pointer to incrementing unsigned long int */
#define BIOSTICK  (*(volatile unsigned long far *)(0x0040006CL))
```

The BIOSTICK get incremented for every 18.2 times per second. But this is not much preferred by the professional programmers.

15.12.2.2 Solution with int 8 handler

You might have noticed that all DOS games work fine on all systems. The reason is game programmers' use the techniques of installing this int8 handler for delay as:

```
/* Author: Alexander J. Russel */
volatile unsigned long fast_tick, slow_tick;
static void interrupt (far *oldtimer)(void);   /* BIOS timer handler */

void deinit_timer(void);

/*-----------------------------------------------
    new_timer
      Logic:
   You don't have to call the old timer, but if you don't
   you have to write some code to cleanup in de-init that
   fixes DOS's internal clock.

   Its also considered 'good form' to call the old int.
   If everyone does, then everything that other TSR's etc...
   may have installed will also work.

   If you skip the little chunk of ASM code- the out 20-
   you WILL LOCKUP all interrupts, and your computer

   Anyways, this test replacement just increments a couple of
   long ints.                                  */

static void interrupt new_timer(void)
{
   asm cli
   fast_tick++;

   if ( !(fast_tick & 3) ) // call old timer ever 4th new tick
    {
      oldtimer( );  // not the best way to chain
      slow_tick++;
    }
   else
     {
      // reset PIC
      asm {
          mov al, 20h
          out 20h, al
          }
     }
   asm sti
}
```

Note

Here we come across inline assembly. The clear description can be found on next chapter.

```
/*-----------------------------------------
    init_timer
      Logic:
      see that 1st line of inline asm!
        to set whatever clock speed you want load
        bx with 1193180/x where x is the
            clock speed you want in Hz. */

void init_timer(void)
{
   slow_tick=fast_tick=0l;
   oldtimer=getvect(8); // save old timer

   asm cli

   // speed up clock
   asm {
            mov     bx,  19886 /* set the clock speed to
                                  60Hz (1193180/60) */
            mov     al,  00110110b
            out     43h, al
            mov     al,  bl
            out     40h, al
            mov     al,  bh
            out     40h, al
      }

   setvect(8, new_timer);

   asm sti
}

/*--------------------------------
      deinit_timer             */

void deinit_timer(void)
{
   asm cli

   // slow down clock   1193180 / 65536 = 18.2, but we use zero

   asm {
            xor bx,  bx        // min rate 18.2 Hz when set to zero
            mov al,  00110110b
            out 43h, al
            mov al,  bl
            out 40h, al
```

```
            mov al,  bh
            out 40h, al
      }

   setvect(8, oldtimer);  // restore oldtimer

   asm sti
}
```

Then we can use the following code in `main( )` to get a machine independent delay.

```
next_time=fast_tick + 3;  /* fast tick is incremented by
                             the int8 ISR (global)*/
while( next_time>=fast_tick )
      ;     /* wait */
```

15.12.3 Floating point formats not linked

You will get this error when the TC does some optimizing techniques. TC's optimizing techniques prevent the floating point to be linked unless our program needs. But in certain cases, the compiler's decision would be wrong and even though we use floating formats, it doesn't link it. Normally it would happen when we don't call any floating point functions but we use %f in `scanf( )` or `printf( )`. In such a case we must take effort explicitly to link floating formats.

```
struct foo
{
    float a;
    int b;
};

int main( void )
{
    int i;
    struct foo s[2];
    for ( i=0; i<2; ++i )
     {
      printf( "Enter a: " );
      scanf( "%f", &s[i].a );
      printf( "Enter b: " );
      scanf( "%d", &s[i].b );
      printf( "a=%f, b=%d \n", s[i].a, s[i].b );
      }
    getch( );
    return(0);
} /*--main( )------*/
```

The above program will result in runtime error as:

```
Enter a:  scanf : floating point formats not linked
Abnormal program termination
```

15.12.3.1 Solution with pragma directive

One of the remedies for floating point formats link error is to include a pragma directive in our file as per Borland's suggestion:

```
extern unsigned _floatconvert;
#pragma extref _floatconvert
```

15.12.3.2 Another solution

Another remedy for floating point formats link error is to use our own code to force floating point formats to be linked.

```
void Force2LinkFloat( void )
{
   float a, *f=&a;
   *f = 0000;  /* dummy value */
}
```

Just include the above piece of code in your file. You don't need to call the above function. If the above function gets linked, with your code, it would automatically force floating point formats to be linked.

15.12.4 Null pointer assignment

You will get this message when you assign a value through a pointer without first assigning a value to the pointer. Normally it would happen if you use `strcpy( )` or `memcpy( )` with a pointer as its first argument.

Your program may look as if it runs correctly, but if you get this message, bug will be somewhere inside. The actual reason for the cause is you might have written, via a Null or uninitialized pointer, to location 0000. Whenever TC finds `exit( )` or returns from `main( )`, it would check whether the location 0000 in your data segment contains different values from what you started with. If so, you might have used an uninitialized pointer. That is, you may get the error message irrespective of where the error actually occurred.

The remedy for this problem is to watch the following expressions with **Add Watch** (Ctrl+F7):

```
*(char *)0,4m
(char *)4
```

If the values at these locations get changed, it means that the line just executed is the one causing the problem.