**IMPORTANT KEYWORDS**

**const**

**Makes variable value or pointer parameter unmodifiable.**

When const is used with a variable, it uses the following syntax:

const *variable-name* [ = *value*];

In this case, the const modifier allows you to assign an initial value to a variable that cannot later be changed by the program. For example,

const my\_age = 32;

Any assignments to 'my\_age' will result in a compiler error. However, such declaration is quite different than using

#define my\_age 32

In the first case, the compiler allocates a memory for 'my\_age' and stores the initial value 32 there, but it will not allow any later assignment to this variable. But, in the second case, all occurences of 'my\_age' are simply replaced with 32 by the [preprocessor](http://tigcc.ticalc.org/doc/cpp.html), and no memory will be allocated for it.   
  
Warning: a const variable can be indirectly modified by a pointer, as in the following example:

\*(int\*)&my\_age = 35;

When the const modifier is used with a pointer parameter in a function's parameter list, it uses the following syntax:

*function-name* (const *type* \**var-name*)

Then, the function cannot modify the variable that the pointer points to. For example,

int printf (const char \*format, ...);

Here the printf function is prevented from modifying the format string.

## enum

**Defines a set of constants of type int.**

The syntax for defining constants using enum is

enum [*tag*] {*name* [=*value*], ...};

The set can optionally be given a type tag name with *tag*. *name* is the name of a constant that can optionally be assigned the (constant) value of *value*, etc. For example,

enum Numbers {One = 1, Two = 2, Three = 3, Four = 4, Five = 5};

If *value* is missing, then a value is assumed to be the value of the previous constant in the list + 1. If this is the first constant in the list, the default value is 0.   
  
If you give a type tag name, then you can declare variables of enumerated type using

enum *tag* *variable-names*;

For example,

enum Numbers x, y, z;

declares three variables x, y and z, all of type *Numbers* (they are, in fact, integer variables). More precise, 'enum *tag*' becomes a new type which is equal in rights with any built-in type.

## extern

**Indicates that an identifier is defined elsewhere.**

Keyword extern indicates that the actual storage and initial value of a variable, or body of a function, is defined elsewhere, usually in a separate source code module. So, it may be applied to data definitions and function prototypes:

extern *data-definition*;

extern *function-prototype*;

For example,

extern int \_fmode;

extern void Factorial (int n);

The keyword extern is optional (i.e. default) for a function prototype.

## goto

**Unconditionally transfer control.**

goto may be used for transfering control from one place to another. The syntax is:

goto *identifier*;

Control is unconditionally transferred to the location of a local label specified by *identifier*. For example,

Again:

...

goto Again;

Jumping out of scope (for example out of the body of the [for](http://tigcc.ticalc.org/doc/keywords.html#for) loop) is legal, but jumping into a scope (for example from one function to another) is **not** allowed.   
  
**Note:** The GNU C extends the usage of goto keyword to allow [computed goto](http://tigcc.ticalc.org/doc/gnuexts.html#SEC65). Also, it supports [local labels](http://tigcc.ticalc.org/doc/gnuexts.html#SEC64), useful in macro definitions.

## register

**Tells the compiler to store the variable being declared in a CPU register.**

In standard C dialects, keyword auto uses the following syntax:

register *data-definition*;

The register type modifier tells the compiler to store the variable being declared in a CPU register (if possible), to optimize access. For example,

register int i;

Note that TIGCC will automatically store often used variables in CPU registers when the optimization is turned on, but the keyword register will force storing in registers even if the optimization is turned off. However, the request for storing data in registers may be denied, if the compiler concludes that there is not enough free registers for use at this place.   
  
**Note:** The GNU C extends the usage of register keyword to allow [explicitely choosing of used registers](http://tigcc.ticalc.org/doc/gnuexts.html#SEC97).

## static

**Preserves variable value to survive after its scope ends.**

Keyword static may be applied to both data and function definitions:

static *data-definition*;

static *function-definition*;

For example,

static int i = 10;

static void PrintCR (void) { putc ('\n'); }

static tells that a function or data element is only known within the scope of the current compile. In addition, if you use the static keyword with a variable that is local to a function, it allows the last value of the variable to be preserved between successive calls to that function.   
  
Note that the initialization of automatic and static variables is quite different. Automatic variables (local variables are automatic by default, except you explicitely use static keyword) are initialized during the run-time, so the initialization will be executed whenever it is encountered in the program. Static (and global) variables are initialized during the compile-time, so the initial values will simply be embeded in the executable file itself. If you change them, they will retain changed in the file. By default, the C language proposes that all uninitialized static variables are initialized to zero, but due to some limitations in TIGCC linker, you need to initialize explicitely all static and global variables if you compile the program in "nostub" mode.   
  
The fact that global and static variables are initialized in compile-time and kept in the executable file itself has one serious consequence, which is not present on "standard" computers like PC, Mac, etc. Namely, these computers always reload the executable on each start from an external memory device (disk), but this is not the case on TI. So, if you have the following global (or static) variable

int a = 10;

and if you change its value somewhere in the program to 20 (for example), its initial value will be 20 (not 10) on the next program start! Note that this is true only for global and static variables. To force reinitializing, you must put explicitely something like

a = 10;

at the begining of the main program!   
  
Note, however, that if the program is archived, the initial values will be restored each time you run the program, because archived programs are reloaded from the archive memory to the RAM on each start, similarly like the programs are reloaded from disks on "standard" computers each time when you start them.

## struct

**Groups variables into a single record.**

The syntax for defining records is:

struct [*struct-type-name*]

{

[*type* *variable-names*] ;

...

} [*structure-variables*] ;

A struct, like an [union](http://tigcc.ticalc.org/doc/keywords.html#union), groups variables into a single record. The *struct-type-name* is an optional tag name that refers to the structure type. The *structure-variables* are the data definitions, and are also optional. Though both are optional, one of the two must appear.   
  
Elements in the record are defined by naming a *type*, followed by *variable-names* separated by commas. Different variable types can be separated by a semicolon. For example,

struct my\_struct

{

char name[80], phone\_number[80];

int age, height;

} my\_friend;

declares a record variable *my\_friend* containing two strings (*name* and *phone\_number*) and two integers (*age* and *height*). To declare additional variables of the same type, you use the keyword struct followed by the *struct-type-name*, followed by the variable names. For example,

struct my\_struct my\_friends[100];

declares an array named *my\_friends* which components are records. In fact, 'struct my\_struct' becomes a new type which is equal in rights with any built-in type.   
  
To access elements in a structure, you use a record selector ('.'). For example,

strcpy (my\_friend.name, "Mr. Wizard");

A bit field is an element of a structure that is defined in terms of bits. Using a special type of struct definition, you can declare a structure element that can range from 1 to 16 bits in length. For example,

struct bit\_field

{

int bit\_1 : 1;

int bits\_2\_to\_5 : 4;

int bit\_6 : 1;

int bits\_7\_to\_16 : 10;

} bit\_var;

## typedef

**Creates a new type.**

The syntax for defining a new type is

typedef *type-definition* *identifier*;

This statement assigns the symbol name *identifier* to the data type definition *type-definition*. For example,

typedef unsigned char byte;

typedef char str40[41];

typedef struct {float re, im;} complex;

typedef char \*byteptr;

typedef int (\*fncptr)(int);

After these definition, you can declare

byte m, n;

str40 myStr;

complex z1, z2;

byteptr p;

fncptr myFunc;

with the same meaning as you declare

unsigned char m, n;

char myStr[41];

struct {float re, im;} z1, z2;

char \*p;

int (\*myFunc)(int);

User defined types may be used at any place where the built-in types may be used.

## union

**Groups variables which share the same storage space.**

A union is similar to a [struct](http://tigcc.ticalc.org/doc/keywords.html#struct), except it allows you to define variables that share storage space. The syntax for defining unions is:

union [*union-type-name*]

{

*type* *variable-names*;

...

} [*union-variables*] ;

For example,

union short\_or\_long

{

short i;

long l;

} a\_number;

The compiler will allocate enough storage in a number to accommodate the largest element in the union. Elements of a union are accessed in the same manner as a struct.   
  
Unlike a struct, the variables 'a\_number.i' and 'a\_number.l' occupy the same location in memory. Thus, writing into one will overwrite the other.

## volatile

**Indicates that a variable can be changed by a background routine.**

Keyword volatile is an extreme opposite of const. It indicates that a variable may be changed in a way which is absolutely unpredictable by analysing the normal program flow (for example, a variable which may be changed by an interrupt handler). This keyword uses the following syntax:

volatile *data-definition*;

Every reference to the variable will reload the contents from memory rather than take advantage of situations where a copy can be in a register.