# C Language Keywords

Standard ANSI C recognizes the following keywords:

**auto**
**break**
**case**
**char**
**const**
**continue**
**default**
**do**
**double**
**else**
**enum**
**extern**
**float**
**for**
**goto**
**if**
**int**
**long**
**register**
**return**
**short**
**signed**
**sizeof**
**static**
**struct**
**switch**
**typedef**
**union**
**unsigned**
**void**
**volatile**
**while**

In addition to these standard keywords, TIGCC recognizes some extended keywords which do not exist in ANSI C, like asm, typeof, inline, etc., which are described in details in the section GNU C language extensions. This section also describes extensions to standard keywords, not only new ones.

**Note:** If square brackets '[...]' are used in syntax descriptions, they mean optional arguments (as usual in syntax-describing languages), not square brackets as literals.

---

# auto

### Defines a local variable as having a local lifetime.

Keyword `auto` uses the following syntax:

```
[auto] data-definition;
```

As the local lifetime is the default for local variables, `auto` keyword is extremely rarely used.

**Note:** GNU C extends `auto` keyword to allow forward declaration of nested functions.

---

# break

### Passes control out of the compound statement.

The break statement causes control to pass to the statement following the innermost enclosing while, do, for, or switch statement. The syntax is simply

```
break;
```

---

# 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, 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.

---

# continue

### Passes control to the begining of the loop.

`continue` causes control to pass to the end of the innermost enclosing while, do, or for statement, at which point the loop continuation condition is re-evaluated. The syntax is simply

```
continue;
```

For example,

```
for (i = 0; i < 20; i++)
  {
    if (array[i] == 0)
      continue;
    array[i] = 1/array[i];
  }
```

This example changes each element in the array with its reciprocal, but skips elements which are equal to zero.

---

# do

### Do-while loop.

Keyword `do` is usually used together with while to make another form of repeating statement. Such form of the loop uses the following syntax:

```
do statement while (expression)
```

*statement*, which is usually a compound statement, is executed repeatedly as long as the value of *expression* remains non-zero. The test takes place after each execution of the *statement*. For example,

```
i = 1; n = 1;
do
  {
    n *= i;
    i++;
  } while (i <= factorial);
```

---

# 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.

---

# float, double

**Floating point data types.**

The keyword `float` usually represents a single precision floating point data type, and `double` represents a double precision floating point data type. In TIGCC, both `float` and `double` (and even `long double`) are the same. The TI-89 and TI-92 Plus use a non-IEEE floating point format called SMAP II BCD for floating point values.

These values have a range from 1e-999 to 9.999999999999999e999 in magnitude, with a precision of exactly 16 significant digits. Principally, the exponent range may be as high as 16383, but a lot of math routines do not accept exponents greater than 999.

---

# for

**For loop.**

For-loop is yet another kind of loop. It uses `for` keyword, with the following syntax:

```
for ([expr1]; [expr2]; [expr3]) statement
```

*statement* is executed repeatedly until the value of *expr2* is 0. Before the first iteration, *expr1* is evaluated. This is usually used to initialize variables for the loop. After each iteration of the loop, *expr3* is evaluated. This is usually used to increment a loop counter. In fact, the for-loop is absolutely equivalent to the following sequence of statements:

```
expr1;
while (expr2)
  {
    statement;
    expr3;
  }
```

That's why *expr1* and *expr3* must contain side effects, else they are useless. For example,

```
for (i=0; i<100; i++) sum += x[i];

for (i=0, t=string; i<40 && *t; i++, t++) putch(*t);
putch('\n');

for (i=0, sum=0, sumsq=0, i<100; i++)
  {
    sum += i; sumsq += i*i;
  }
```

All the expressions are optional. If *expr2* is left out, it is assumed to be 1. *statement* may be a compound statement as well.

---

# 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 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. Also, it supports local labels, useful in macro definitions.

---

# if, else

**Conditional statement.**

Keyword `if` is used for conditional execution. The basic form of `if` uses the following syntax:

```
if (expression)
   statement1
```

Alternatively, `if` may be used together with `else`, using the following syntax:

```
if (expression)
   statement1
else
   statement2
```

If *expression* is nonzero when evaluated, then *statement1* is executed. In the second case, *statement2* is executed if the expression is 0.

An optional `else` can follow an `if` statement, but no statements can come between an `if` statement and an `else`. Of course, both *statement1* and *statement2* may be compound statements (i.e. a sequence of statements enclosed in braces). Here will be given some legal examples:

```
if (count < 50) count++;

if (x < y) z = x;
else z = y;

if (x < y)
  {
    printf ("x is smaller");
    return x;
  }
else
  {
    printf ("x is greater")
    return y;
  }
```

The #if and #else preprocessor statements look similar to the `if` and `else` statements, but have very different effects. They control which source file lines are compiled and which are ignored.

---

# int, char

**Basic data types (integer and character).**

Variables of type int are one machine-type word in length. They can be signed (default) or unsigned, which means that in this configuration of the compiler they have by default a range of -32768 to 32767 and 0 to 65535 respectively, but this default may be changed if the compiler option **'-mnoshort'** is given. In this case, the range of type int is -2147483648 to 2147483647 for signed case, or 0 to 4294967295 for unsigned case. See also short and long type modifiers.

Variables of type char are 1 byte in length. They can be signed (this is the default, unless you use the compiler option **'-funsigned-char'**) or unsigned, which means they have a range of -128 to 127 and 0 to 255, respectively.

All data types may be used for defining variables, specifying return types of functions, and specifying types of function arguments. For example,

```
int a, b, c;                    // 'a', 'b', 'c' are integer variables
int func ();                    // 'func' is a function returning int
```

```
char crypt (int key, char value);  // 'crypt' is a function returning char with
                                   // two args: 'key' is int and 'value' is char
```

When function return type is omitted, `int` is assumed.

All data type keywords may be used in combination with asterisks, brackets and parentheses, for making complex data types, like pointer types, array types, function types, or combinations of them, which in the C language may have an arbitrary level of complexity (see asterisk for more info).

---

# 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.

---

# return

**Exits the function.**

`return` exits immediately from the currently executing function to the calling routine, optionally returning a value. The syntax is:

`return [`*`expression`*`];`

For example,

```
int sqr (int x)
{
  return (x*x);
}
```

---

# short, long, signed, unsigned

**Type modifiers.**

A type modifier alters the meaning of the base type to yield a new type. Each of these type modifiers can be applied to the base type int. The modifiers `signed` and `unsigned` can be applied to the base type char. In addition, `long` can be applied to double.

When the base type is omitted from a declaration, `int` is assumed. For example,

```
long x;                 // 'int' is implied
unsigned char ch;
signed int i;           // 'signed' is default
unsigned long int l;    // 'int' is accepted, but not needed
```

In this implementation of the compiler, the valid range of valid data types is as listed in the following table:

```
short int              -32768 to 32767
long int               -2147483648 to 2147483647
signed char            -128 to 127
signed int             -32768 to 32767 (signed is default)
                       [or -2147483648 to 2147483647 if '-mnoshort' is given]
signed short int       -32768 to 32767
signed long int        -2147483648 to 2147483647
unsigned char          0 to 255
unsigned int           0 to 65535
                       [or 0 to 4294967295 if '-mnoshort' is given]
unsigned short int     0 to 65535
unsigned long int      0 to 4294967295
```

**Note:** GNU C extends the `long` keyword to allow double-long integers (64-bit integers in this implementation), so they have range from -9223372036854775808 to 9223372036854775807 if signed, or from 0 to 18446744073709551615 if unsigned.

---

# sizeof

**Returns the size of the expression or type.**

Keyword `sizeof` is, in fact, an operator. It returns the size, in bytes, of the given expression or type (as type size_t). Its argument may be an expression of a type name:

```
sizeof expression
sizeof (type)
```

For example,

```
workspace = calloc (100, sizeof (int));
memset(buff, 0, sizeof buff);
nitems = sizeof (table) / sizeof (table[0]);
```

Note that *type* may be an anonymous type (see asterisk for more info about anonymous types).

---

# 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, 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;
```

---

# switch, case, default

### Branches control.

`switch` causes control to branch to one of a list of possible statements in the block of statements. The syntax is

`switch (`*`expression`*`)` *`statement`*

The statement *statement* is typically a compound statement (i.e. a block of statements enclosed in braces). The branched-to statement is determined by evaluating *expression*, which must return an integral type. The list of possible branch points within *statement* is determined by preceding substatements with

`case` *`constant-expression`* `:`

where *constant-expression* must be an int and must be unique.

Once a value is computed for *expression*, the list of possible *constant-expression* values determined from all case statements is searched for a match. If a match is found, execution continues after the matching case statement and continues until a break statement is encountered or the end of *statement* is reached. If a match is not found and this statement prefix is found within *statement*,

`default :`

execution continues at this point. Otherwise, *statement* is skipped entirely. For example,

```
switch (operand)
  {
    case MULTIPLY:
      x *= y; break;
    case DIVIDE:
      x /= y; break;
    case ADD:
      x += y; break;
    case SUBTRACT:
      x -= y; break;
    case INCREMENT2:
      x++;
    case INCREMENT1:
      x++; break;
    case EXPONENT:
    case ROOT:
    case MOD:
      printf ("Not implemented!\n");
      break;
    default:
      printf("Bug!\n");
      exit(1);
  }
```

See also break.

**Note:** GNU C extends the `case` keyword to allow case ranges.

---

# 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, 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.

---

# void

**Empty data type.**

When used as a function return type, void means that the function does not return a value. For example,

```
void hello (char *name)
{
  printf("Hello, %s.", name);
}
```

When found in a function heading, void means the function does not take any parameters. For example,

```
int init (void)
{
  return 1;
}
```

This is not the same as defining

```
int init ()
{
  return 1;
```

```
}
```

because in the second case the compiler will not check whether the function is really called with no arguments at all; instead, a function call with arbitrary number of arguments will be accepted without any warnings (this is implemented only for the compatibility with the old-style function definition syntax).

Pointers can also be declared as void. They can't be dereferenced without explicit casting. This is because the compiler can't determine the size of the object the pointer points to. For example,

```
int x;
float f;
void *p = &x;    // p points to x
*(int*)p = 2;
p = &r;          // p points to r
*(float*)p = 1.1;
```

---

# 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.

---

# while

**Repeats execution while the condition is true.**

Keyword `while` is the most general loop statemens. It uses the following syntax:

```
while (expression) statement
```

*statement* is executed repeatedly as long as the value of *expression* remains nonzero. The test takes place before each execution of the *statement*. For example,

```
while (*p == ' ') p++;
```

Of course, *statement* may be a compound statement as well.

---

**Return to the main index**