N

name space — Definition

The term *name space* refers to the "list" where the translator records an identifier. Each name space holds a different set of identifiers. If two identifiers are spelled exactly the same and appear within the same scope but are not in the same name space, they are *not* considered to be identical.

The four varieties of name space, as follows:

Label names
: The translator treats every identifier followed by a colon ':' or that follows a **goto** statement as a label.

Tags
: A tag is the name that follows the keywords **struct**, **union**, or **enum**. It names the type of object so declared.

Members
: A member names a field within a structure or a **union**. A member can be accessed via the operators '.' or '->'. Each structure or **union** type has a separate name space for its members.

Ordinary identifiers
: These name ordinary functions and variables. For example, the expression

```
int example;
```

: declares the ordinary identifier **example** to name an object of type **int**.

The Standard reserves external identifiers with leading underscores to the implementor. To reduce "name-space pollution," the implementor should not reserve anything that is not explicitly defined in the Standard (macros, **typedef**s, etc.) and that does not begin with a leading underscore.

Example

The following program illustrates the concept of name space. It shows how the identifier **foo** can be used numerous times within the same scope yet still be distinguished. This is extremely poor programming style. Please do not write programs like this.

```
#include <stdio.h>
#include <stdlib.h>

/* structure tag */
struct foo {
      /* structure member */
      struct foo *foo;
      int bar;
};

main(void)
{
      /* ordinary identifier */
      struct foo *foo;
      int i = 0;

      foo = (struct foo *)malloc(sizeof(foo));
      foo->bar = ++i;
      foo->foo = NULL;
```

```
/* label */
foo:  printf("Chain, chain, chain -- chain of \"foo\"s.\n");
      if (foo->foo == NULL) {
            foo->foo = (struct foo *)malloc(sizeof(foo));
            foo->foo->foo = NULL;
            foo->foo->bar = ++i;
            goto foo;
      }

      printf("The foo loop executed %d times\n", foo->foo->bar);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §3.1.2.3

See Also

identifiers, linkage, scope

Notes

Pre-ANSI implementations disagree on the name spaces of structure/**union** members. The Standard adopted the "Berkeley" rules, which state that every unique structure/**union** type has its own name space for its members. It rejected the rules of the first edition of *The C Programming Language*, which state that the members of all structures/**union**s reside in a common name space.

nested comments — Definition

Both *The C Programming Language*, ed. 2 and the draft ANSI standard declare that comments cannot be nested. Earlier versions of **Let's C** included a switch, called **-VCNEST**, that allowed a programmer to nest comments. This switch has been removed. Current and future versions of **Let's C** abort compilation when they detect nested comments.

See Also

Definitions, Language

nm — Command

Print a program's symbol table
nm [-**adgnopru**] *file* ...

nm prints the symbol table of each *file*. Each *file* argument must be a **Let's C** object module.

The first argument selects one of several options. It is optional; if present, it must begin with '-'. The options are as follows:

-a Print all symbols. Normally, **nm** prints names that are in C-style format and ignores symbols with names inaccessible from C programs.

-d Print only defined symbol.

-g Print only global symbols.

-n Sort numerically rather than alphabetically. **nm** uses unsigned compares when sorting symbols with this option.

-o Append the file name to the beginning of each output line.

-p Print symbols in the order in which they appear within the symbol table.

-r Sort in reverse-alphabetical order.

-u Print only undefined symbols.

By default, **nm** sorts symbol names alphabetically. Each symbol is followed by its value and its OMF segment.

See Also

cc, commands, ld, size, strip

nondigit — Definition

In the context of identifiers, a *nondigit* is any one of the following characters:

```
_     a     b     c     d     e     f     g     h
i     j     k     l     m     n     o     p     q
r     s     t     u     v     w     x     y     z
A     B     C     D     E     F     G     H     I
J     K     L     M     N     O     P     Q     R
S     T     U     V     W     X     Y     Z
```

Cross-reference

Standard, §3.1.2

See Also

digit, identifiers

non-local jumps — Overview

At times, exceptional conditions arise in a program that make it desirable to jump to a previous point within the program. **goto** can jump from one point to another within the same function, but it does not permit a jump from one function to another. The **setjmp/longjmp** mechanism was created to allow a program to jump immediately from one function to another, i.e., to perform a non-local jump.

The macro **setjmp** reads the machine environment and stores the environment in the array **jmp_buf**, which must be an array. The "machine environment" consists of the elements that determine the behavior of the machine, e.g., the contents of machine registers. What constitutes the machine environment will vary greatly from machine to machine. It may be impossible on some machines to save such elements of the machine environment as register variables and the contents of the stack or to restore the machine environment from within an extraordinarily complex computation.

For example, consider the following:

```
{
      int status[3][3][3], fn();
      jmp_buf buf;
      status[fn(1)][fn(2)][fn(3)] = setjmp(buf);
}
```

Here, the translator is trying to store the return value of **setjmp** into an array element with extremely complex index computations. It cannot be guaranteed that on every machine, the proper array element will be overwritten on reentry. For this reason, the Standard states that **setjmp** can be expected to save the machine environment only if used in a simple expression, such as in an **if** or **switch** statement.

The function **longjmp** jumps back to the point marked by the earlier invocation of **setjmp**. It restores the machine environment that **setjmp** had saved. This allows **longjmp** to perform a non-local jump.

LEXICON

A non-local jump can be dangerous. For example, many user-level routines cannot be interrupted and reentered safely. Thus, improper use of **longjmp** and **setjmp** with them will create mysterious and irreproducible bugs.

The Standard mandates that **longjmp** work correctly "in the contexts of interrupts, signals and any of their associated functions." Experience has shown, however, that **longjmp** should not be used within an exception handler that interrupts STDIO routines.

longjmp must not restore the machine environment of a routine that has already returned.

Cross-references

Standard, §4.6
The C Programming Language, ed. 2, p. 254

See Also

jmp_buf, Library, setjmp.h

Notes

longjmp's behavior is undefined if it is invoked from within a function that is called by a signal that is received during the handling of another signal. See **signal handling** for more information on signals.

notmem() — Extended function (libc)

Check if memory is allocated
int notmem(char ******ptr***);**

notmem checks if a memory block has been allocated by **calloc**, **malloc**, or **realloc**. *ptr* points to the block to be checked.

notmem searches the arena for *ptr*. It returns one if *ptr* is not a memory block obtained from **malloc**, **calloc**, or **realloc**, and zero if it is.

See Also

arena, calloc, extended miscellaneous, free, malloc, realloc, setbuf

null directive — Definition

Directive that does nothing

A *null directive* is a preprocessing directive that consists only of a '#' followed by **<newline>**. It does nothing.

Cross-reference

Standard, §3.8.7

See Also

preprocessing

null pointer constant — Definition

A *null pointer constant* is an integral constant expression with the value of zero, or such a constant that has been cast to type **void** *****. When the null pointer constant is compared with a pointer for equality, it is converted to the same type as the pointer before they are compared.

The null pointer constant always compares unequal to a pointer to an object or function. Two null pointers will always compare equal, regardless of any casts.

Cross-references

Standard, §3.2.2.3
The C Programming Language, ed. 2, p. 102

See Also

conversions, NULL

null statement — Definition

A *null statement* is one that consists only of a semicolon ';'. Its syntax is as follows:

null statement:
;

A null statement performs no operations.

Cross-references

Standard, §3.6.3
The C Programming Language, ed. 2, p. 222

See Also

Definitions, statements

numerical limits — Overview

The Standard describes numerical limits for every arithmetic type. For integral types, it sets the largest and smallest values that can be held in the given environment. For floating types, it also gives values for the manner in which a floating-point number is encoded.

These limits are recorded in two groups of macros: one for integral types, and the other for floating types. The groups of macros are kept, respectively, in the headers **limits.h** and **float.h**. The Lexicon entries for these headers lists the Standard's numerical limits.

Cross-references

Standard, §2.2.4.2
The C Programming Language, ed. 2, p. 257

See Also

Environment

Notes

The ANSI Committee has tried to keep its numerical limits compatible with those given in IEEE document 754, which describes a floating-point standard for binary number systems.

nybble — Definition

A **nybble** is four bits, or half of an eight-bit byte. The term is generally used to refer to the low four bits or the high four bits of a byte. Thus, a byte may be said to have a "low nybble" and a "high nybble". One nybble encodes one hexadecimal digit.

See Also

bit, byte, Definitions