B

behavior — Definition

The term *behavior* refers to the way an implementation reacts to a given construct. When a construct conforms to the descriptions within the Standard, then its behavior should be predictable from the Standard's descriptions alone. When a construct does not conform to the descriptions within the Standard, then one of the following four types of abnormal behavior results:

Unspecified behavior

This is behavior produced by a correct construct for which the Standard supplies no description. An example is the order in which a program evaluates the arguments to a function.

Undefined behavior

This is behavior produced by an erroneous construct for which the Standard supplies no description. An example of a construct that generates undefined behavior is attempting to divide by zero.

The Standard does not mandate how a conforming implementation reacts when it detects a construct that will produce undefined behavior: it may pass over it in silence, with unpredictable (and usually unwelcome) results; generate a diagnostic message and continue to translate or execute; or stop translation or execution and produce a diagnostic message.

A portable program, however, should not depend upon undefined behavior performing in any predictable way. Undefined behavior is precisely that: undefined. Whatever happens, happens — from printing an error message to reformatting your hard disk.

Implementation-defined behavior

This is behavior produced by a correct construct that is specific to a given implementation. An example is the number of **register** objects that can actually be loaded into machine registers. The Standard requires that the implementation document all such behaviors.

Locale-specific behavior

This is behavior that depends upon the program's locale. An example is the character that the function **atof** recognizes as marking a decimal point. The Standard requires that an implementation document all such behaviors.

Cross-reference

Standard, §1.6

See Also

compliance, Definitions

Notes

For a program to be maximally portable, it should not rely on any of the above deviants of behavior.

BIOS — Definition

BIOS is an acronym for *basic input/output system*. In most machines, the BIOS consists of a group of routines carried in the read-only memory (ROM). These routines contain basic instructions for accessing the various aspects of the hardware. MS-DOS uses these routines to help protect itself from the peculiarities of the hardware on which it is running.

See Also

Definitions, STDIO

bios.h — Header

Outline ROM BIOS data area

bios.h is a header file to be used with programs that directly access the IBM PC's BIOS data area. It declares a structure that defines the entire BIOS data area, for examination and alteration.

The sample program **biosdata.c**, which is included with **Let's C**, uses **bios.h** to take a "snapshot" of the BIOS data area and print a summary of it.

See Also

BIOS data area, header, peek, poke

bit — Definition

The term *bit* is an abbreviation for *binary digit*. It is the element of storage that can hold either of exactly two values. A contiguous sequence of bits forms a byte. A byte consists of at least eight bits. The macro **CHAR_BIT** specifies the number of bits that constitute a byte for the execution environment.

On most machines a bit cannot be addressed directly; a byte is the smallest unit of storage that can be addressed.

Cross-reference

Standard, §1.6

See Also

bit-field, bitwise operations, byte, Definitions

bit-fields — Definition

A *bit-field* is a member of a structure or **union** that is defined to be a cluster of bits. It provides a way to represent data compactly. For example, in the following structure

```
struct example {
      int member1;
      long member2;
      unsigned int member3 :5;
}
```

member3 is declared to be a bit-field that consists of five bits. A colon ':' precedes the integral constant that indicates the *width*, or the number of bits in the bit-field. Also, the bit-field declarator must include a type, which must be one of **int**, **signed int**, or **unsigned int**. If a bit-field is declared to be in type **int**, the implementation defines whether the highest bit is used to hold the bit-field's sign.

The Standard states, "An implementation may allocate any addressable storage unit large enough to hold a bit-field." This suggests that if a bit-field is defined as holding more bits than are normally held by an **int**, then the implementation may place the bit-field into a larger data object, such as a **long**.

If two bit-fields are declared side-by-side and together are small enough to fit into an **int**, then they must be packed together. However, if together they are too large to fit into an **int**, then the implementation determines whether they are in separate objects or if the second bit-field is partly within the object that holds the first and partly within a second object.

The implementation also defines where the bit-field resides within its object — whether it is built from the low-order bit up, or from the high-order bit down. For example, consider an implementation in which an **int** has 16 bits. If a five-bit bit-field is declared to be part of an **int**, and that bit-field is initialized to all ones, then the **int** may appear like this under one implementation:

```
0000 0000 0001 1111     /* low-order bits set */
```

and like this under another:

```
1111 1000 0000 0000     /* high-order bits set */
```

A bit-field that is not given a name may not be accessed. Such an object is useful as "padding" within an object so that it conforms to a template designed elsewhere.

A bit-field that is unnamed and has a length of zero can be used to force adjacent bit-fields into separate objects. For example, in the following structure

```
struct example {
      int member1;
      int member2 :5;
      int :0;
      int member3 :5;
};
```

the zero-length bit-field forces **member2** and **member3** to be written into separate objects, regardless of the default behavior of the implementation.

Finally, it is not allowed to take the address of a bit-field.

Cross-references

Standard, §3.5.2.1
The C Programming Language, ed. 2, pp

See Also

bit, bit map, byte, Definitions

Notes

Because bit-fields have many implementation-specific properties, they are not considered to be highly portable. Bit-fields use minimal amounts of storage, but the amount of computation needed to manipulate and access them may negate this benefit. Bit-fields must be kept in integral-sized objects because many machines cannot access a quantity of storage smaller than a "word" (a word is generally used to store an **int**).

bit map — Definition

A **bit map** is a string of bits in which each bit has a symbolic, rather than numeric, value.

See Also

bit, byte, Definitions

Notes

C permits the manipulation of bits within a byte through the use of bit field routines. These generate code rather than calls to routines. Bit fields are generally less efficient than masking because they always generate masking and shifting.

block — Definition

A *block* is a set of statements that forms one syntactic unit. It can have its own declarations and initializations.

In C terminology, a block is marked off by braces '{ }'. Block-scoped variables are visible only in the block in which they are declared.

Cross-references

Standard, §3.6.2
The C Programming Language, ed. 2, p. 55

See Also

auto, compound statement, Definitions, scope

Notes

Another term for “block” is *compound statement.*

break — C keyword

Exit unconditionally from loop or switch
break;

break is a statement that causes the program to exit immediately from the smallest enclosing **switch**, **while**, **for**, or **do** statement.

Example

For an example of this statement, see **printf**.

Cross-references

Standard, §3.6.6.3
The C Programming Language, ed. 2, p. 64

See Also

C keywords, continue, goto, statements, return

bsearch() — General utility (libc)

Search an array
#include <stdlib.h>
void *bsearch(const void **item*, **const void** **array*, **size_t** *number*,
size_t *size*, **int (***comparison***)(const void** **arg1*, **const char** **arg2*));

bsearch searches a sorted array for a given item.

item points to the object sought. *array* points to the base of the array; it has *number* elements, each of which is *size* bytes long. Its elements must be sorted into ascending order before it is searched by **bsearch**.

comparison points to the function that compares *item* with an element of *array*. *comparison* must return zero if its arguments match, a number greater than zero if the element pointed to by *arg1* is numerically greater than the element pointed to by *arg2*, and a number less than zero if the element pointed to by *arg1* is numerically less than the element pointed to by *arg2*.

bsearch returns a pointer to the array element that matches *item*. If no element matches *item*, then **bsearch** returns NULL. If more than one element within *array* matches *item*, which element is matched is unspecified.

Example

This example uses **bsearch** to translate English into “bureaucrat-ese”.

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

```
struct syntab {
      char *english, *bureaucratic;
} cdtab[] = {
/* The left column is in alphabetical order */

      "affect",        "impact",
      "after",         "subsequent to",
      "building",      "physical facility",
      "call",          "refer to as",
      "do",            "implement",

      "false",         "inoperative",
      "finish",        "finalize",
      "first",         "initial",
      "full",          "in-depth",
      "help",          "facilitate",

      "lie",           "inoperative statement",
      "order",         "prioritize",
      "talk",          "interpersonal communication",
      "then",          "at that point in time",
      "use",           "utilize"
};

int
comparator(key, item)
char *key;
struct syntab *item;
{
      return(strcmp(key, item->english));
}

main(void)
{
      struct syntab *ans;
      char buf[80];

      for(;;) {
            printf("Enter an English word: ");
            fflush(stdout);

            if(gets(buf) || !strcmp(buf, "quit") == NULL)
                  break;

            if((ans = bsearch(buf, (void *)cdtab,
                        sizeof(cdtab)/ sizeof(struct syntab),
                        sizeof(struct syntab),
                        comparator)) == NULL)
                  printf("%s not found\n");

            else
                  printf("Don’t say \"%s\"; say \"%s\"!\n",
                        ans->english, ans->bureaucratic);
      }

      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.5.1
The C Programming Language, ed. 2, p. 253

See Also

qsort, searching-sorting

byte — Definition

A *byte* is a contiguous set of at least eight bits. It is the unit of storage that is large enough to hold each character within the basic C character set. It is also the smallest unit of storage that a C program can address.

The least significant bit is called the *low-order bit*, and the most significant bit is the *high-order bit*.

In terms of C programming, a byte is synonymous with the data type **char**: a **char** is defined to be equal to one byte's worth of storage. The macro **CHAR_BIT** gives the number of bits in a byte for the execution environment.

Cross-reference

Standard, §1.6

See Also

bit, char, Definitions

byte ordering — Technical information

Describe order of bytes

Byte ordering is the order in which a given machine stores successive bytes of a multibyte data item. Different machines order bytes differently.

The following example displays a few simple examples of byte ordering:

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

main(void)
{
      union
      {
            char b[4];
            int i[2];
            long l;
      } u;
      u.l = 0x12345678L;

      printf("%x %x %x %x\n",
            u.b[0], u.b[1], u.b[2], u.b[3]);
      printf("%x %x\n", u.i[0], u.i[1]);
      printf("%lx\n", u.l);
      return(EXIT_SUCCESS);
}
```

When run on the 68000 or the Z8000, the program gives the following results:

```
      12 34 56 78
      1234 5678
      12345678
```

As you can see, the order of bytes and words from low to high memory is the same as is represented on the screen.

When run on a PDP-11, however, the program gives these results:

```
34 12 78 56
1234 5678
12345678
```

As you can see, the PDP-11 inverts the order of words in memory.

Finally, when the program is run on the i8086 you see these results:

```
78 56 34 12
5678 1234
12345678
```

The i8086 inverts both words and long words.

See Also

Language, technical information