O

object — Definition

An *object* is an area of memory that can contain one or more values. With the exception of a bit-field, an object consists of a byte or a contiguous group of bytes. The significance of each byte's value is defined by the program or the implementation. Objects that are variables are interpreted according to their type.

Cross-references

Standard, §1.6
The C Programming Language, ed. 2, p. 197

See Also

Definitions

object definition — Definition

A *definition* is a declaration that reserves storage for the thing declared. An *object definition* defines an object and makes it available throughout either the translation unit (if it has internal linkage) or throughout the program (if it has external linkage).

The term "tentative definition" refers to a definition to which more information is added by a later re-definition of the same object. The extra information may be a storage-class specifier, or it may initialize the object. The term, although somewhat misleading, is meant to show that every object has only one definition, but that definition can be refined during the course of translation.

Only one definition can contain an initializer. If an object is not initialized by the end of a file, it is initialized to zero.

A tentative definition of a static, incomplete object is disallowed semantically:

```
static int array[];
   . . .
int array[] = {3, 4, 5, 6};   /* Non-portable */
```

Because the Standard does not forbid an implementation to support such code, it may not generate an error message; however, this code is not portable.

The following *is* allowed semantically:

```
int array[];
   . . .
static int array[] = {3, 4, 5, 6};  /* RIGHT */
```

However, it may create a linker conflict in some implementations, such as in one-pass compilers.

To be assured that your code is maximally portable, declare the storage class and size of each object before you use it.

Cross-references

Standard, §3.7.2
The C Programming Language, ed. 2, p. 197

See Also

definition, external definitions, function definition, linkage, object

object format — Definition

An **object format** describes the form of compiled program that contains relocation information. The linker reads files in object format to create executable files.

See Also

Definitions, ld, n.out

object types — Definition

The *object types* are the set of types that describe objects. This set includes the **integral types**, the **floating types**, the **pointer types**, and the **aggregate types**.

Cross-reference

Standard, §3.1.2.5

See Also

function type, incomplete type, pointer, types

obsolescent — Definition

The term *obsolescent* refers to any feature of the C language that is widely used, but that may be withdrawn from future editions of the Standard. For example, consider the practice of first defining a function and then following the definition with a list of parameter declarations:

```
int example(parm1, parm2, parm3)
long parm1;
char *parm2;
int parm3;
{
      . . .
}
```

The Standard regards this as obsolete, and may eventually withdraw recognition of it in favor of the following syntax:

```
int example(long parm1, char *parm2, int parm3)
{
      . . .
}
```

The Standard regards three features of the language as being obsolete. The first is the use of separate lists of parameters identifiers and declaration lists, as described above. The second is the use of function declarators with empty parentheses; if a function takes no arguments, the word **void** should appear between the parentheses. The third is the placing of storage-class specifier at any point other than at the beginning of the declaration specifiers.

Cross-reference

Standard, §1.8, §3.9

See Also

Definitions, function declarators, storage-class specifiers

open() — Extended function (libc)

Open a file
short open(char **file***, short** *type***);**

LEXICON

open prepares a *file* to receive data, or to have its data read. When it can open *file*, **open** returns a file descriptor, which is a small, positive integer that identifies the opened *file* for subsequent calls to **read**, **write**, **close**, **dup**, or **dup2**.

type determines how the file is opened, as follows:

0	Read only
1	Write
2	Read and write

Once *file* is opened, reading or writing begins at byte 0.

open returns -1 if the file is nonexistent, or if a system resource is exhausted.

Example

This example copies the file named in **argv[1]** to the one named in **argv[2]**.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define BUFSIZE (20*512)
char buf[BUFSIZE];

/* prototypes for extended functions */
extern short close(short fd);
extern short open(char *file, short type);
extern short read(short fd, char *buffer, short n);
extern short write(short fd, char *buffer, short n);

void
fatal(char *message)
{
      fprintf(stderr, "copy: %s\n", message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      register short ifd, ofd;
      register unsigned short n;

      /* Check number of arguments */
      if (argc != 3)
            fatal("Usage: copy source destination");

      /* Open files */
      if ((ifd = open(argv[1], 0)) == -1)
            fatal("cannot open input file");
      if ((ofd = creat(argv[2], 0)) == -1)
            fatal("cannot open output file");

      /* Read and write text */
      while ((n = read(ifd, buf, BUFSIZE)) != 0) {
            if (n == -1)
                  fatal("read error");
            if (write(ofd, buf, n) != n)
                  fatal("write error");
      }
```

```
      if (close(ifd) == -1 || close(ofd) == -1)
            fatal("cannot close");
      exit(EXIT_SUCCESS);
}
```

See Also

close, extended miscellaneous, file descriptor, fopen

Notes

open is a low-level call that passes data directly to MS-DOS. It should not be mixed with high-level calls, such as **fread**, **fwrite**, or **fopen**.

open is not described in the ANSI Standard. Any program that uses it does not comply strictly with the Standard, and may not be portable to other compilers or environments.

operating system devices — Overview

Logical devices for system peripherals

MS-DOS gives names to its logical devices. **Let's C** uses these names to access these devices via MS-DOS.

MS-DOS includes the following logical devices:

aux	Auxiliary (serial) port
com1	Serial port
con	Console
lpt1	Parallel port; not always implemented
nul	Null device (the "bit bucket")
prn	Parallel port

You can use the function **fopen** to open these devices, just as if they were files. However, if you attempt to seek on a device, undefined behavior will occur.

See Also

Environment

operators — Overview

An *operator* specifies an operation performed upon one or two operands. The operation yields a value, performs designation, produces a side effect, or performs any combination of these.

The C language uses the following operators:

!	Not
!=	Compare two arithmetic operands for inequality
#	Substitute preprocessor token ("stringize")
##	Token-paste preprocessor tokens
%	Modulus operation on two arithmetic operands
%=	Modulus operation and assign result
&	Bitwise AND operation
&&	Logical AND for two expressions
&=	Bitwise AND operation and assign result
()	Cast operators
*	Multiply two arithmetic operands
***=**	Multiply two arithmetic operands and assign result
+	Add two arithmetic operands
++	Increment a scalar operand
+=	Add two operands and assign result

,	Evaluate an *rvalue*
-	Subtract two scalar operands, unary minus
--	Decrement a scalar operand
-=	Subtract two operands and assign result
->	Offset from structure/**union** pointer
.	Select member from structure/**union**
/	Divide two arithmetic operands
/=	Divide two arithmetic operands and assign result
<	Less than
<<	Bitwise left shift
<<=	Bitwise left shift and assign result
<=	Less than or equality
=	Assignment operator
==	Equality
>	Greater than
>=	Greater than or equal
>>	Bitwise right shift
>>=	Bitwise right shift and assign result
? :	Perform if/else operation
[]	Array subscript
^	Perform bitwise exclusive OR operation
^=	Perform bitwise exclusive OR and assign result
defined	Check if a macro is defined
sizeof	Size of operand in bytes
|	Perform bitwise OR operation
|=	Perform bitwise OR and assign result
||	Logical OR for two expressions
~	One's complement

The term *precedence* refers to the default order in which the operators in an expression are evaluated. The following list gives the default precedence of operators. Precedence is always overridden by the operators **()**, which, by default, enclose a primary expression:

Operator	*Associativity*
`() [] -> . (operand) * & sizeof`	Left to right
`~ ++ -- - * & sizeof`	Right to left
`* / %`	Left to right
`+ -`	Left to right
`<< >>`	Left to right
`< <= > >=`	Left to right
`== !=`	Left to right
`&`	Left to right
`^`	Left to right
`|`	Left to right
`&&`	Left to right
`||`	Left to right
`?:`	Right to left
`= += -= *= /= %=`	Right to left
`,`	Left to right

Cross-references

Standard, §3.1.5
The C Programming Language, ed. 2, pp. 41*ff*

See Also

lexical elements, punctuators

ordinary identifier — Definition

An *ordinary identifier* names all identifiers *except* labels, tags, and members. For example, the expression

```
int example;
```

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

Cross-references

Standard, §3.1.2.6
The C Programming Language, ed. 2, p. 192

See Also

name space

outb() — Extended function (libc)

Write to a port
int outb(int *port*, **int** *data*);

outb provides a C interface to the i8086 machine instruction **out**. It writes the least significant byte of the 16-bit word *data* to *port*, and returns *data*.

Example

For an example of this function, see the entry for **inb**.

See Also

extended miscellaneous, inb, inw, outw