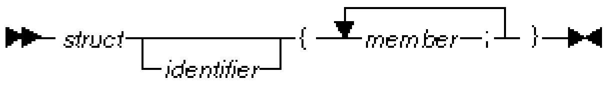
struct (Structures)

A *structure* contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have varied types. Each data object within a structure is called a *member* or *field*.

Use structures to group logically-related objects. For example, to allocate storage for the components of one address, define the following variables:

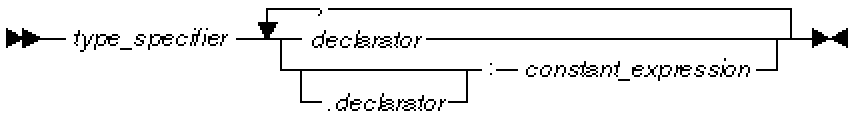
```
int street_no;
char *street_name;
char *city;
char *prov;
char *postal_code;
```

Declaring a Structure



identifier Provides a tag name for the structure. If specified, subsequent declarations (in the same scope) of variables using the structure can be made by referring to the tag name. If not specified, you must place all variable definitions that refer to the structure within the declaration of the data type.

member The list of members provides the data type with a description of the values that can be stored in the structure.



A member that does not represent a bit field can be of any data type and can have the **volatile** or **const** qualifier.

If a `:` (colon) and a constant expression follow the member declarator, the member represents a *bit field*. Bit fields are described in Declaring and Using Bit Fields in Structures.

A structure type declaration describes the members that are part of the structure.

Identifiers used as structure or member names can be redefined to represent different objects in the same scope without conflicting. You cannot use the name of a member more than once in a structure type, but you can use the same member name in another structure type that is defined within the same scope.

You cannot declare a structure type that contains itself as a member, but you can declare a structure type that contains a pointer to itself as a member.

Defining a Structure Variable

A structure variable definition contains an optional storage class keyword, the **struct** keyword, a structure tag, a declarator, and an optional identifier. The structure tag indicates the data type of the structure variable.

C++ The keyword **struct** is optional in C++.

You can declare structures having any storage class. Most compilers, however, treat structures declared with the **register** storage class specifier as automatic structures.

Initializing Structures
The initializer contains an = (equal sign) followed by a brace-enclosed comma-separated list of values. You do not have to initialize all members of a structure.

The following definition shows a completely initialized structure:

```
struct address {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
               };
static struct address perm_address =
               { 3, "Savona Dr.", "Dundas", "Ontario", "L4B 2A1"};
```

The values of `perm_address` are:

Member	Value
`perm_address.street_no`	`3`
`perm_address.street_name`	address of string `"Savona Dr."`
`perm_address.city`	address of string `"Dundas"`
`perm_address.prov`	address of string `"Ontario"`
`perm_address.postal_code`	address of string `"L4B 2A1"`

The following definition shows a partially initialized structure:

```
struct address {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
               };
struct address temp_address =
               { 44, "Knyvet Ave.", "Hamilton", "Ontario" };
```

The values of `temp_address` are:

Member	Value
`temp_address.street_no`	`44`
`temp_address.street_name`	address of string `"Knyvet Ave."`
`temp_address.city`	address of string `"Hamilton"`
`temp_address.prov`	address of string `"Ontario"`
`temp_address.postal_code`	value depends on the storage class.

Note: The initial value of uninitialized structure members like `temp_address.postal_code` depends on the storage class associated with the member.

Alignment of Structures
Structures are aligned according to the setting of the

- AIX align
- OS/2 WIN /Sp

compiler option, which specifies the alignment rules the compiler uses when laying out memory storage for structures and unions. The mapping of a structure is based on the alignment setting in effect at the beginning of the structure definition.

Structures and unions with identical members, but using different alignment, are not type compatible and cannot be assigned to each other. Use the **extchk** compiler option in AIX to check for alignment mismatches, and refer to the attribute section of the compiler listing to find the variables that have different alignment settings.

Your code should not depend on the offset or alignment of members within a structure. Use the **offsetof** macro, defined in the **stddef.h** header file, to determine the offset of members in a macro.

Declaring Structure Types and Variables
To define a structure type and a structure variable in one statement, put a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the statement.

For example:

```
static struct {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
              } perm_address, temp_address;
```

Because this example does not name the structure data type, `perm_address` and `temp_address` are the only structure variables that will have this data type. Putting an identifier after **struct**, lets you make additional variable definitions of this data type later in the program.

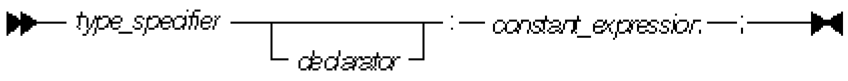
The structure type (or tag) cannot have the **volatile** qualifier, but a member or a structure variable can be defined as having the **volatile** qualifier.

For example:

```
static struct class1 {
                        char descript[20];
                        volatile long code;
                        short complete;
                     } volatile file1, file2;
struct class1 subfile;
```

This example qualifies the structures `file1` and `file2`, and the structure member `subfile.code` as **volatile**.

Declaring and Using Bit Fields in Structures
A structure can contain *bit fields* that allow you to access individual bits. You can use bit fields for data that requires just a few bits of storage. A bit field declaration contains a type specifier followed by an optional declarator, a colon, a constant expression, and a semicolon.



The *constant expression* specifies how many bits the field reserves.

Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized. A zero-width bit field causes the next field to be aligned on the next container boundary, where the container is the same size as the underlying type as the bit field.

The maximum bit-field length is implementation dependent. The maximum bit field length for the compiler is 32 bits (4 bytes, or 1 word).

For portability, do not use bit fields greater than 32 bits in size.

The following restrictions apply to bit fields. You cannot:

- Define an array of bit fields
- Take the address of a bit field
- Have a pointer to a bit field
- Have a reference to a bit field (C++ only)

In C, you can declare a bit field as type **int**, **signed int**, or **unsigned int**. Bit fields of the type **int** are equivalent to those of type **unsigned int**.

The default integer type for a bit field is **unsigned**. Use the **bitfields=signed** option to change this default.

In extended mode C, bit fields can be any integral type. For example,

```
struct S {
   short x : 4;
   long  y : 10;
   char  z : 7;
} s;
```

Non-integral bit fields in extended mode C are converted to type **unsigned int** and a warning is issued. In other modes, the use of non-integral bit fields results in an error.

In **ansi** mode C, bit fields of type **unsigned char** or **unsigned short** are changed to **unsigned int**. An **unsigned short** bit field occupies 32 bits.

A bit field cannot have the **volatile** or **const** qualifier.

The following structure has three bit-field members `kingdom, phylum`, and `genus`, occupying 12, 6, and 2 bits respectively:

```
struct taxonomy {
     int kingdom : 12;
     int phylum : 6;
     int genus : 2;
     };
```

Alignment of Bit Fields in Structures

Bit fields are word-aligned but packed as closely as possible into the current word. The first bit field in a sequence of bit fields starts on a word boundary. For example, a structure containing only bit fields is word-aligned, but after the first bit field, the bit fields themselves do not have to begin on word boundaries.

C++ `Unlike ANSI/ISO C, C++ bit fields can be any integral type or enumeration type. When you assign a value that is out of range to a bit field, the low-order bit pattern is preserved and the appropriate bits are assigned.`

If a series of bit fields does not add up to the size of an **int**, padding can take place. The amount of padding is determined by the alignment characteristics of the members of the structure. Bit fields cannot cross word boundaries but are forced to start at the next word boundary. Alignment of structures is described in .

The following example declares the identifier `kitchen` to be of type `struct on_off`:

```
struct on_off {
                  unsigned light : 1;
                  unsigned toaster : 1;
                  int count;            /* 4 bytes */
                  unsigned ac : 4;
                  unsigned : 4;
                  unsigned clock : 1;
                  unsigned : 0;
                  unsigned flag : 1;
                 } kitchen ;
```

The structure `kitchen` contains eight members totalling 16 bytes. The following table describes the storage that each member occupies:

Member Name	Storage Occupied
`light`	1 bit
`toaster`	1 bit
(padding, 30 bits)	to next **int** boundary
`count`	the size of an **int**
`ac`	4 bits
(unnamed field)	4 bits
`clock`	1 bit
(padding, 23 bits)	to next **int** boundary (unnamed field)
`flag`	1 bit
(padding, 31 bits)	to next **int** boundary

All references to structure fields must be fully qualified. For instance, you cannot reference the second field by `toaster`. You must reference this field by `kitchen.toaster`.

The following expression sets the `light` field to `1`:

```
kitchen.light = 1;
```

When you assign to a bit field a value that is out of its range, the bit pattern is preserved and the appropriate bits are assigned. The following expression sets the `toaster` field of the `kitchen` structure to 0 because only the least significant bit is assigned to the `toaster` field:

```
kitchen.toaster = 2;
```

Bit Fields under the align Compiler Option
Bit fields are also subject to the **align** compiler option.

The default alignment in AIX, for example, is **align=power**. When it is in effect, bit fields are aligned as described in Alignment of Bit Fields in Structures. Bit fields have the following alignment properties under the

twobyte and **packed** suboptions in AIX.

twobyte

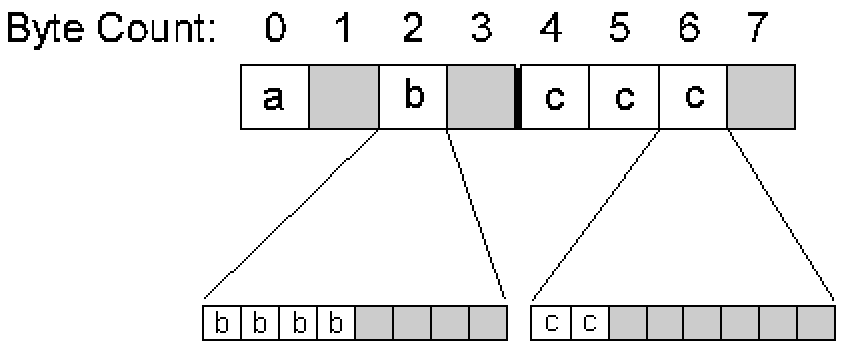
Bit fields are packed into a word and are aligned on a halfword boundary. Bit fields cannot cross word boundaries but are forced to start at the next halfword boundary even if they start on a halfword boundary.

A bit field with a width of 0 (zero) forces the next member to start at the next halfword boundary even if it is not a bit field and even if the zero-width bit field is already at a halfword boundary. A structure containing nothing but zero-width bit fields has a length equal to twice the number of zero-width bit fields.

In the following example, the bit fields in the structure `species` are aligned according to the **align=twobyte** option:

```
#pragma options align=twobyte
struct species {
      char a;
      int : 0;
      int b : 4;
      int c : 18;  /* 8 + 8 + 2 bits */
};
```

The following figure shows the layout of `species`. The shaded areas are padding.



Bit field `b` starts on a halfword boundary because of the unnamed zero-width **int** bit field. It occupies the first 4 bits of the third byte (byte 2 in the figure.) Because bit field `c` is larger than 2 bytes, it cannot cross the word boundary between bytes 3 and 4, but is forced to start at byte 4. It occupies bytes 4 and 5 (the first two bytes of the second word) and 2 bits of byte 6.

packed

Bit fields are packed into a 1 byte space. Bit fields that cross byte boundaries are forced to start at the next available byte boundary.

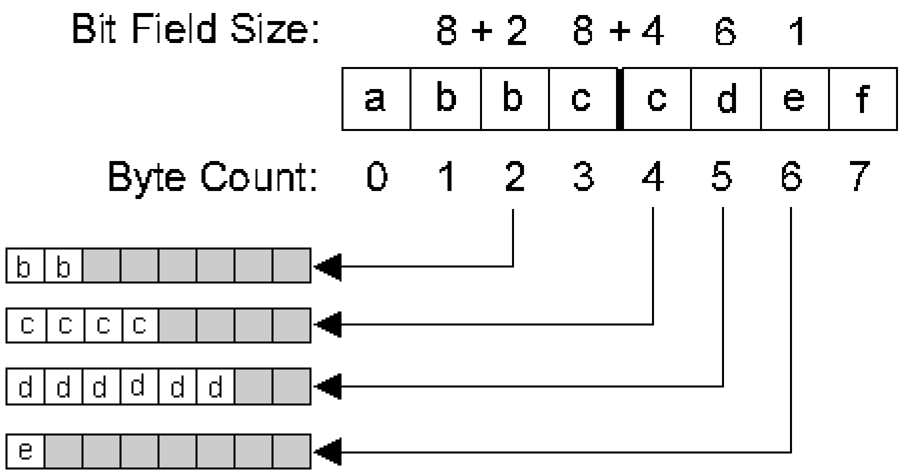
A bit field with a width of 0 (zero) forces the next member to start at the next byte boundary. If the zero-width bit field is already at a byte boundary, the next structure member starts there. A non-bit field member following a bit field is aligned on the next byte boundary.

In the following example, the bit fields in the structure `order` are aligned according to the **align=packed** option:

```
#pragma options align=packed
struct order {
      char a;
      int b : 10;
      int c : 12;
      int d : 6;
      int : 0;
      int e : 1;
```

```
          char f;
     };
```

The following figure shows the layout of `order`. The shaded areas are padding.



Because bit field `c` is longer than 1 byte and cannot straddle the boundary between bytes 2 and 3, it must start at byte 3. Likewise, field `d` cannot cross the byte boundary between bytes 4 and 5; it is forced to start at byte 5. The zero-width bit field between field `d` and field `e` forces bit field `e` to start at byte 6.

RELATED CONCEPTS
Declarators
Initializers

RELATED REFERENCES
Incomplete Types
Structure and Union Member Specification
Examples of Structure Declaration and Use