Data Type

defined by user in C

Structures

- **Defining Structures:**

C structures are used to defines groups of contiguous fields, such as records or control blocks. Usually, the structure definition is saved as an #include member in a maclib library, but it can be placed at the top of the program. The commonest way to define structures is with a typedef, as shown below.

```
typedef struct country
   {
   char     name[20];
   int     population;
   char language[10];
   } Country;
```

This defines a structure which can be referred to either as 'struct country' or Country, whichever you prefer. Strictly speaking, you don't need a tag name both before and after the braces if you're not going to use one or the other. But it's standard practice to put them both in and to give them the same name, but with the one after the braces starting with an uppercase letter.

The typedef statement doesn't occupy storage: it simply defines a new type. Variables that are declared with the typedef above will be of type 'struct country', just like population is of type integer.

By default, C will align arithmetic fields on halfword, fullword or doubleword boundaries, depending on the type. You can force the compiler to eliminate slack bytes caused by boundary alignment by using compiler options, such as pack in C/370 or bytealign in SAS/C.

Structures

- **<u>Declaring Structures:</u>**

Once the structure is defined, you can declare a structure variable by preceding the variable name by the structure type name. If you've defined the structure with tags both before and after the braces, it's a matter of style whether you declare the variable using the word struct or not. Both the following lines declare variables of the type defined above.

```
struct country France, Italy, Sweden;
Country Mexico, Canada, Brazil;
```

If you have a small structure you just want to define in the program, you can do the definition and declaration together as shown below. This will define a structure of type 'struct country' and declare three instances of it.

```
struct country
   {
   char     name[20];
   int     population;
   char language[10];
   } India, Singapore, Indonesia;
```

Strictly speaking, the tag name before the brace isn't needed here, but it's usually left in to comment the structure. You can also use it to declare new variables, just as with typedef.

```
struct country Australia;
```

Structures

- **Addressing a Field in a Structure:**

Fields in a structure can be referenced by following the name of the structure by a dot and the name of the field.

```
France.population = 0;
```

You can also declare a pointer to type 'struct country' and use that to reference the fields in a structure of that type. In this case, you follow the pointer name by an arrow and the name of the field.

```
struct country Russia, *sptr;
sptr = &Russia;
sptr->population = 0;
```

In the example above, Russia is of type 'struct country'; *sptr is of type 'pointer to struct country'; and sptr->population is of type integer.

- **Comparing and Copying Structures:**

Individual fields in structures can be compared or copied like any other variables.

```
Italy.population = France.population;
if (memcmp(Canada.language, France.language. sizeof(Canada.language)) == 0);
   {
   Translate(FRENCH);
   }
else;
```

The following line copies the entire structure of Mexico to Brazil.

```
memcpy(&Brazil, &Mexico, sizeof(Brazil));
```

If you're comparing one structure with another, all the usual warnings about slack bytes apply as in other languages.

Unions

- **Defining Unions:**

A union is the equivalent of an assembler org or a COBOL REDEFINE. It allows you to handle an area of memory that could contain different types of variables. The syntax for unions is identical to that for structures. You can use either typedef's or instream definitions: simply replace the word struct with the word union.

You can contain unions within structures, or structures within unions. Either way, the C union is an unwieldy mechanism.

For an alternative way of dealing with redefined areas, see the section called Void Pointers in the Pointer tutorial. In that section, a record is read into a character buffer and then a pointer to a structure describing the particular record layout is loaded with the buffer address using an appropriate cast.

A union might be used to group different record layouts from the same file, or to handle a single field that could contain, for example, either numeric or character data.

```
typedef struct transaction
   {
   int      amount;
   union
     {
     int    count;
     char   name[4];
     } udata;
   char     discount;
   } Transaction;
```

Unions

Declaring a Union

A union type definition contains the union keyword followed by an optional identifier (tag) and a brace-enclosed list of members.

A union definition has the following form:

```
>>-union--+-identifier--------------------------+-------------><
          |                    .-----------.    |
          |                    V           |    |
          '-+------------+--{----member--;-+--}-'
            '-identifier-'
```

A union declaration has the same form as a union definition except that the declaration has no brace-enclosed list of members.

The identifier is a tag given to the union specified by the member list. Once a tag is specified, any subsequent declaration of the union (in the same scope) can be made by declaring the tag and omitting the member list. If a tag is not specified, all variable definitions that refer to that union must be placed within the statement that defines the data type.

The list of members provides the data type with a description of the objects that can be stored in the union.

A union member definition has same form as a variable declaration.

Unions

A member of a union can be referenced the same way as a member of a structure.

For example:

```
union {
      char birthday[9];
      int age;
      float weight;
      } people;
 people.birthday[0] = '\n';
```

assigns '\n' to the first element in the character array birthday, a member of the union people.

A union can represent only one of its members at a time. In the example, the union people contains either age, birthday, or weight but never more than one of these. The printf statement in the following example does not give the correct result because people.age replaces the value assigned to people.birthday in the first line:

```
#include <stdio.h>
#include <string.h>
 union {char birthday[9];
  int age;
  float weight;
} people;
 int main(void) {
  strcpy(people.birthday, "03/06/56");
  printf("%s\n", people.birthday);
```

Unions

```
  people.age = 38;
  printf("%s\n", people.birthday);
}
```

The output of the above example will be similar to the following:

```
03/06/56
&
```

The fields in this structure are referred to as follows:

```
Transaction trans;
trans.amount = 0;
trans.udata.count = 0;
trans.discount = 'N';
```

Just as in any other language, it's up to you to determine what kind of variable is present in any instance. If you do arithmetic on a character string, you'll probably S0C7 abend.

Unions

- **<u>Using Structures within Unions:</u>**

Unions can contain any types of variables, including structures. Be aware that the length of a union is the length of its longest variable. If the name in the previous example had only been 3 characters long, there would have been an undefined slack byte after name, since an integer occupies 4 bytes.If you have a more complex situation than a simple redefine of one scalar variable with another, you may need to use structures in the union. Let's say an area in the record could contain either one long integer or two short integers; one solution would be to define a structure for the two shorts.

```
typedef struct twoshorts
   {
   short smallamount1;
   short smallamount2;
   } TwoShorts;
```

```
typedef union udata
  {
    TwoShorts smallamounts;
    int       bigamount;
   } Udata;
```

Enumerations

An **enumeration** is a data type consisting of a set of values that are named integral constants. It is also referred to as an enumerated type because you must list (enumerate) each of the values in creating a name for each of them. A named value in an enumeration is called an enumeration constant. In addition to providing a way of defining and grouping sets of integral constants, enumerations are useful for variables that have a small number of possible values.

You can define an enumeration data type and all variables that have that enumeration type in one statement, or you can declare an enumeration type separately from the definition of variables of that type. The identifier associated with the data type (not an object) is called an enumeration tag. Each distinct enumeration is a different enumeration type.

Compatible Enumerations

In C, each enumerated type must be compatible with the integer type that represents it. Enumeration variables and constants are treated by the compiler as integer types. Consequently, in C you can freely mix the values of different enumerated types, regardless of type compatibility.

Enumerations

- **Declaring an Enumeration Data Type**

An enumeration type declaration contains the enum keyword followed by an optional identifier (the enumeration tag) and a brace-enclosed list of enumerators. Commas separate each enumerator in the enumerator list. C99 allows a trailing comma between the last enumerator and the closing brace. A declaration of an enumeration has the form:

```
                                .-,----------.
                                V            |
>>-enum--+------------+--{----enumerator-+--}--;---------------><
         '-identifier-'
```

The keyword enum, followed by the identifier, names the data type (like the tag on a struct data type). The list of enumerators provides the data type with a set of values.

In C, each enumerator represents an integer value. In C++, each enumerator represents a value that can be converted to an integral value.

An enumerator has the form:

```
>>-identifier--+---------------------------------+-------------><
               '-=--integral_constant_expression-'
```

To conserve space, enumerations may be stored in spaces smaller than that of an int.

Enumerations

- **Enumeration Constants**

When you define an enumeration data type, you specify a set of identifiers that the data type represents. Each identifier in this set is an enumeration constant.

The value of the constant is determined in the following way:

An equal sign (=) and a constant expression after the enumeration constant gives an explicit value to the constant. The identifier represents the value of the constant expression.

If no explicit value is assigned, the leftmost constant in the list receives the value zero (0).

Identifiers with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous identifier.

- **Defining Enumeration Variables**

An enumeration variable definition has the following form:

```
>>-+-------------------------+----------------------------------->
   '-storage_class_specifier-'
 >--enum--enumeration_data_type_name--identifier----------------->
 >--+-------------------------+----------------------------------><
   '-=--enumeration_constant-'
```

You must declare the enumeration data type before you can define a variable having that type.

- **Defining an Enumeration Type and Enumeration Objects**

You can define a type and a variable in one statement by using 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 declaration. For example:

```
register enum score { poor=1, average, good } rating = good;
```

Bit Fields

- **Declaring and Using Bit Fields in Structures**

Both C allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. These space-saving structure members are called bit fields, and their width in bits can be explicitly declared. Bit fields are used in programs that must force a data structure to correspond to a fixed hardware representation and are unlikely to be portable.

```
The syntax for declaring a bit field is as follows:
>>-type_specifier--+------------+--:--constant_expression--;---><
                   '-declarator-'
```

A bit field declaration contains a type specifier followed by an optional declarator, a colon, a constant integer expression that indicates the field width in bits, and a semicolon. A bit field declaration may not use either of the type qualifiers, const or volatile.

The C99 standard requires the allowable data types for a bit field to include qualified and unqualified _Bool, signed int, and unsigned int. In addition, this implementation supports the following types.

```
int
short, signed short, unsigned short
char, signed char, unsigned char
long, signed long, unsigned long
long long, signed long long, unsigned long long
```

Bit Fields

In all implementations, the default integer type for a bit field is unsigned.

In either language, 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.

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

Bit fields are also subject to the align compiler option. Each of the align suboptions gives a different set of alignment properties to the bit fields. For a full discussion of the align compiler option and the #pragmas affecting alignment.

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

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;
     };
```

Bit Fields

- **<u>Alignment of Bit Fields</u>**

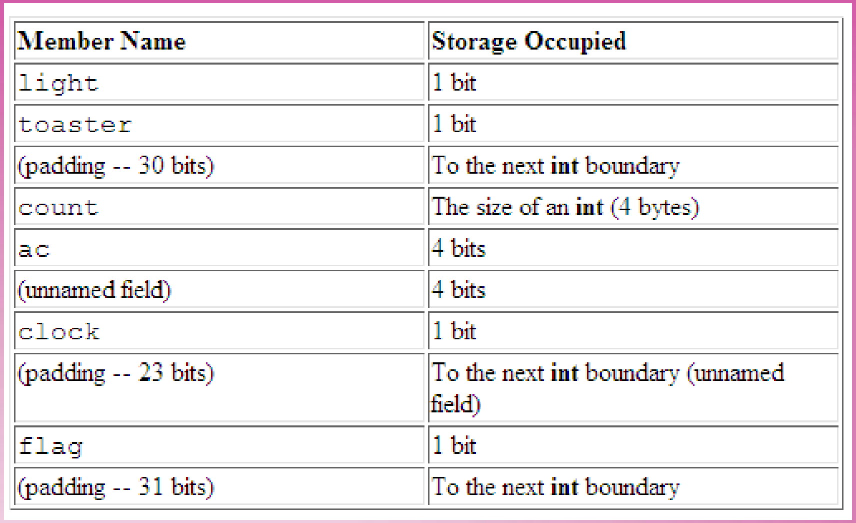
If a series of bit fields does not add up to the size of an int, padding can take place. The amount of padding is determined by the alignment characteristics of the members of the structure.

The following example demonstrates padding, and is valid for all implementations. Suppose that an int occupies 4 bytes. The 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:

Bit Fields



Member Name	Storage Occupied
light	1 bit
toaster	1 bit
(padding -- 30 bits)	To the next **int** boundary
count	The size of an **int** (4 bytes)
ac	4 bits
(unnamed field)	4 bits
clock	1 bit
(padding -- 23 bits)	To the next **int** boundary (unnamed field)
flag	1 bit
(padding -- 31 bits)	To the 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;