18ES611 Embedded System Programming

Sarath tv

Structures

C arrays allow you to define type of variables that can hold several data items of the same kind but structure is another user defined data type available in C programming, which allows you to combine data items of different kinds.

A structure can be considered as a template used for defining a collection of variables under a single name. Structures help programmers to group elements of different data types into a single logical unit.

A single structure might contain integer elements, floating– point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members.

Since members are NOT the same type/size, ***they are not as easy to access as array elements that are the same size.***

Structure variable names are NOT replaced with a pointer in an expression (like arrays)

Defining a Structure

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

struct statement

The struct statement defines a **new data type**, with more than one member for your program.

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition.

At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional.

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

```
struct structure_name
 {
data_type member1;
data_type member2; . .
data_type memeber;
 };
```

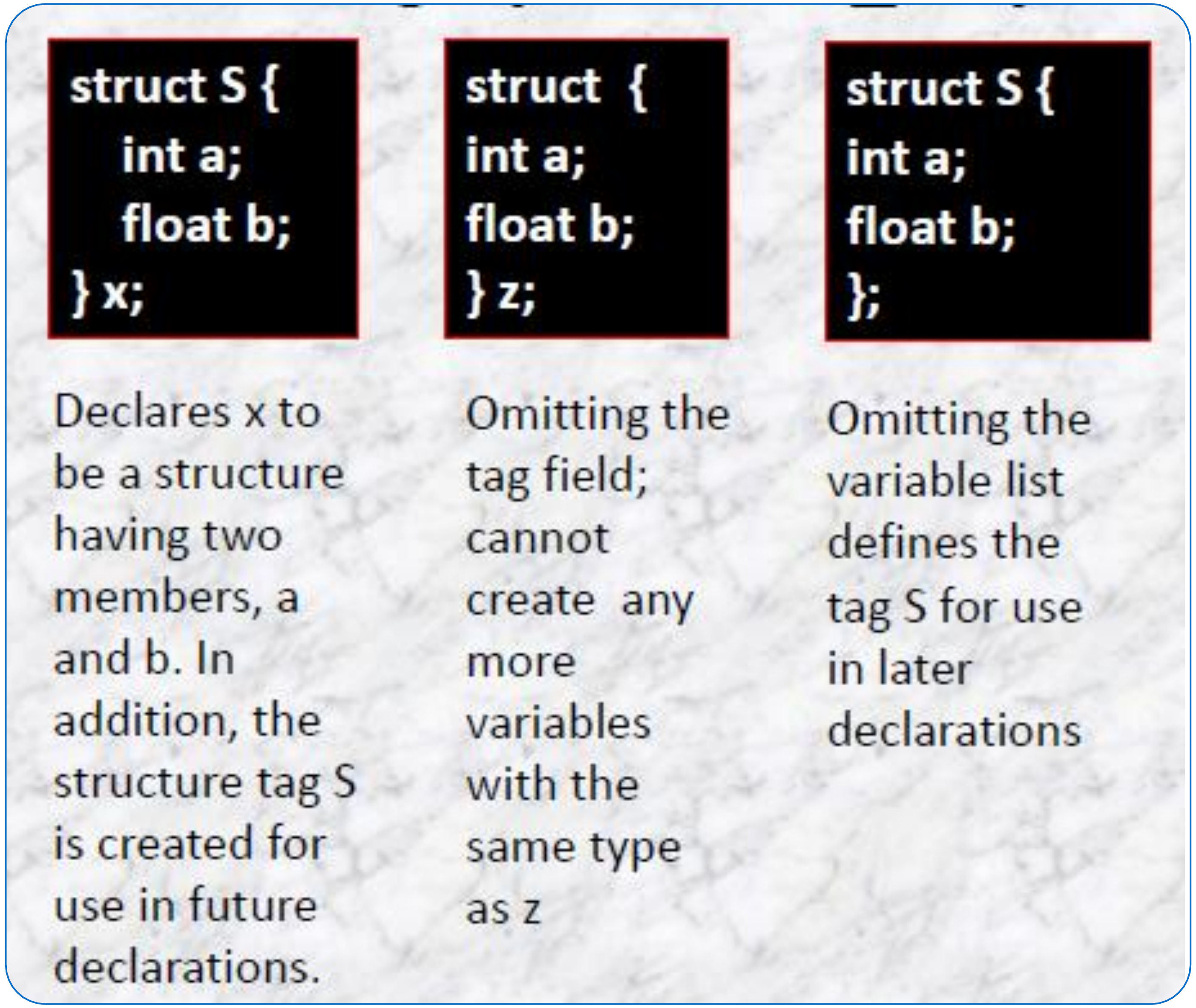
```
struct person
{
char name[50];
int citNo;
float salary;
};
```

```
struct person person1, person2,
person3[20];
```

This declaration above creates the derived data type struct person.

Another way of creating a structure variable is:

```
struct person
{
char name[50];
int citNo;
float salary;
} person1, person2, person3[20];
```



```
struct S {
    int a;
    float b;
} x;
```

Declares x to be a structure having two members, a and b. In addition, the structure tag S is created for use in future declarations.

```
struct  {
int a;
float b;
} z;
```

Omitting the tag field; cannot create any more variables with the same type as z

```
struct S {
int a;
float b;
};
```

Omitting the variable list defines the tag S for use in later declarations

Keyword typedef while using structure

Writing **struct structure_name variable_name**; to declare a structure variable isn't intuitive as to what it signifies, and takes some considerable amount of development.

Developers generally use **typedef** to name the structure as a whole.

typedef keyword is used in creating a type comp (which is of type as struct complex).

Then, two structure variables comp1 and comp2 are created by this comp type.

```
typedef   struct
complex {
int imag;
float real;
} comp;

int main() {
comp      comp1,
comp2;
}
```

ex=STR_2

Accessing members of a structure

There are two types of operators used for accessing members of a structure.

- Member operator(.)
- Structure pointer operator(->)

Any member of a structure can be accessed as:

structure_variable_name.member_name

Passing structures to a function

Passing structure by value

A structure variable can be passed to the function as an argument as a normal variable.

If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

Example :STR_arg

Passing structure by reference

The **memory address of a structure variable** is passed to function while passing it by reference.

If structure is passed by reference, **changes made to the structure variable inside function definition reflects in the originally passed structure variable**

Ex- STR_arg_ref

In this program, structure variables dist1 and dist2 are passed by value to the add function (because value of dist1 and dist2 does not need to be displayed in main function).

But, dist3 is passed by reference ,i.e, address of dist3 (&dist3) is passed as an argument. Due to this, the structure pointer variable d3 inside the add function points to the address of dist3 from the calling main function. So, any change made to the d3 variable is seen in dist3 variable in main function.

C Programming Structure and Pointer

Structures can be created and accessed using pointers. A pointer variable of a structure can be created as below:

```
struct name {
   member1;
   member2;
};
int main()
{
   struct name *ptr;
}
```

Here, the pointer variable of type struct name is created.

Accessing structure's member through pointer

A structure's member can be accessed through pointer in two ways:

- Referencing pointer to another address to access memory
- Using dynamic memory allocation

1. Referencing pointer to another address to access the memory

STR_access_reference

Using -> operator to access structure pointer member.

Structure pointer member can also be accessed using -> operator.

(*personPtr).age is same as personPtr->age

(*personPtr).weight is same as personPtr->weight

STR_access_reference

2. Accessing structure member through pointer using dynamic memory allocation

ptr = (cast-type*) malloc(byte-size)

Example STR_ACCESS_DYNMC

Alignment *requirement*

Storage for the basic C datatypes on an x86 or ARM processor doesn’t normally start at arbitrary byte addresses in memory. Rather, each type except char has an ***alignment requirement***;

chars can start on any byte address,

but **2-byte** shorts must start on an **even** address,

4-byte **ints** or **floats** must start on an address **divisible** by **4**,

and 8-byte longs or doubles must start on an address **divisible by 8.**

Signed or unsigned makes no difference.

Padding

Now we’ll look at a simple example of variable layout in memory. Consider the following series of variable declarations in the top level of a C module:

```
char *p;
char c;
int x;
```

on a 32-bit machine 4 bytes of pointer would be immediately followed by 1 byte of char and that immediately followed by 4 bytes of int.

The storage for c follows immediately. But the 4-byte alignment requirement of x forces a gap in the layout; it comes out as though there were a fourth intervening variable, like this:

```
char *p;     /* 4 or 8 bytes */
char c;      /* 1 byte */
char pad[3]; /* 3 bytes */
int x;       /* 4 bytes */
```

The pad[3] character array represents the fact that there are three bytes of waste space in the structure

Compare what happens if x is a 2-byte short:

```
char *p;
char c;
short x;
```

In that case, the actual layout will be this:

```
char *p;      /* 4 or 8 bytes */
char c;       /* 1 byte */
char pad[1];  /* 1 byte */
short x;      /* 2 bytes */
```

On the other hand, if x is a long on a 64-bit machine

```
char *p;
char c;
long x;
```

we end up with this:

```
char *p;     /* 8 bytes */
char c;      /* 1 byte
char pad[7]; /* 7 bytes */
long x;      /* 8 bytes */
```

If you have been following carefully, you are probably now wondering about the case where the shorter variable declaration comes first:

```
char c;
char *p;
int x;
```

If the actual memory layout were written like this

```
char c;
char pad1[M];
char *p;
char pad2[N];
int x;
```

what can we say about `M` and `N`?

First, in this case `N` will be zero. The address of `x`, coming right after `p`, is guaranteed to be pointer-aligned, which is never less strict than int-aligned.

The value of `M` is less predictable. If the compiler happened to map `c` to the last byte of a machine word, the next byte (the first of `p`) would be the first byte of the next one and properly pointer-aligned. M would be zero.

It is more likely that `c` will be mapped to the *first* byte of a machine word. In that case M will be whatever padding is needed to ensure that `p` has pointer alignment - 3 on a 32-bit machine, 7 on a 64-bit machine.

Intermediate cases are possible. M can be anything from 0 to 7 (0 to 3 on 32-bit) because a char can start on any byte boundary in a machine word.

Structure alignment and padding

A struct instance will have the alignment of its widest scalar member. Compilers do this as the easiest way to ensure that all the members are self-aligned for fast access.

Consider this struct:

```
struct foo1 {
    char *p;
    char c;
    long x;
};
```

Assuming a 64-bit machine, any instance of `struct foo1` will have 8-byte alignment. The memory layout of one of these looks unsurprising, like this:

```
struct foo1 {
    char *p;     /* 8 bytes */
    char c;      /* 1 byte
    char pad[7]; /* 7 bytes */
    long x;      /* 8 bytes */
};
```

trailing padding on structures.

stride address of a structure-- **It is the first address following the structure data that has the same alignment as the structure.**

the compiler will behave as though the structure has trailing padding out to its stride address

Consider this example on a 64-bit x86 or ARM machine:

```
struct foo3 {
    char *p;     /* 8 bytes */
    char c;      /* 1 byte */
};

struct foo3 singleton;
struct foo3 quad[4];
```

You might think that `sizeof(struct foo3)` should be 9, but it's actually 16. The stride address is that of `(&p)[2]`. Thus, in the `quad` array, each member has 7 bytes of trailing padding, because the first member of each following struct wants to be self-aligned on an 8-byte boundary. The memory layout is as though the structure had been declared like this:

```
struct foo3 {
    char *p;     /* 8 bytes */
    char c;      /* 1 byte */
    char pad[7];
};
```

THANK YOU!!!!!