**STRUCTURES**

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

Let us create a few structures suitable for graphics. The basic object is a point, which we will assume has an x coordinate and a y coordinate, both integers.



The two components can be placed in structure declared like this:

**Structure Declaration:**

struct point

{

int x;

int y;

};

The keyword **struct** introduces a structure declaration, which is a list of declarations enclosed in braces. The **structure tag** may follow word **struct** (as with **point** here)

The variables named in a structure are called **members**. A **struct** declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just for any basic type. That is,

struct {....} x, y, z;

is syntactically analogous to

int x, y, z;

in the sense that each statement declares x, y and z to be variables of the named type and causes space to be set aside for them.

A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or the shape of a structure.

**Structure Definition:**

struct point pt;

defines a variable **pt** which is a structure of type **struct point**.

A structure can be initialized by following its definition with a list of initializers, each a constant expression,

struct point pt = { 320, 200 };

A member of a particular structure is referred to in an expression by a construction of the form

structure-name.member

The structure member operator “**.**” connects the structure name and the member name.

Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners:



struct rect

{

struct point pt1;

struct point pt2;

};

The **rect** structure contains two **point** structures. If we declare **screen** as

struct rect screen;

then

screen.pt1.x

refers to the ***x***coordinate of the **pt1** member of **screen**.

**STRUCTURES AND FUNCTIONS**

Let us investigate structures by writing some functions to manipulate points and rectangles.

There are at least three possible approaches: pass components separately, pass an entire

structure, or pass a pointer to it. Each has its good points and bad points. The first function, **makepoint**, will take two integers and return a **point** structure:

/\* makepoint: make a point from x and y components \*/

struct point makepoint(int x, int y)

{

struct point temp;

temp.x = x;

temp.y = y;

return temp;

}

Notice that there is no conflict between the argument name and the member with the same name; indeed the re-use of the names stresses the relationship.

**makepoint** can now be used to initialize any structure dynamically, or to provide structure arguments to a function:

struct rect screen;

struct point middle;

struct point makepoint( int, int );

screen.pt1 = makepoint( 0,0 );

screen.pt2 = makepoint( XMAX, YMAX );

middle = makepoint( ( screen.pt1.x + screen.pt2.x ) / 2, ( screen.pt1.y + screen.pt2.y ) / 2 );

The next step is a set of functions to do arithmetic on points. For instance,

/\* addpoints: add two points \*/

struct addpoint(struct point p1, struct point p2)

{

p1.x += p2.x;

p1.y += p2.y;

return p1;

}

Here both the arguments and the return value are structures. We incremented the components in **p1** rather than using an explicit temporary variable to emphasize that structure parameters are passed by value like any others.

If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure. Structure pointers are just like pointers to ordinary variables.

The declaration

struct point \*pp;

says that **pp** is a pointer to a structure of type **struct point**. If **pp** points to a **point** structure, **\*pp** is the structure, and **(\*pp).x** and **(\*pp).y** are the members. To use **pp**, we might write, for example,

struct point origin, \*pp;

pp = &origin;

printf("origin is (%d,%d)\n", (\*pp).x, (\*pp).y);

The parentheses are necessary in **(\*pp).x** because the precedence of the structure member operator **.** is higher then **\***. The expression **\*pp.x** means **\*(pp.x)**, which is illegal here because **x** is not a pointer.

Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If **p** is a pointer to a structure, then

p->*member-of-structure*

refers to the particular member. So we could write instead

printf("origin is (%d,%d)\n", pp->x, pp->y);

Both **.** and **->** associate from left to right, so if we have

struct rect r, \*rp = &r;

then these four expressions are equivalent:

r.pt1.x

rp->pt1.x

(r.pt1).x

(rp->pt1).x

The structure operators **.** and **->**, together with **()** for function calls and **[]** for subscripts, are at the top of the precedence hierarchy and thus bind very tightly.

For example, given the declaration

struct

{

int len;

char \*str;

} \*p;

Then

++p->len

increments **len**, not **p**, because the implied parenthesization is **++(p->len)**. Parentheses can be used to alter binding: **(++p)->len** increments **p** before accessing **len**, and **(p++)->len** increments **p** afterward. (This last set of parentheses is unnecessary.)

In the same way, **\*p->str** fetches whatever **str** points to; **\*p->str++** increments **str** after accessing whatever it points to (just like **\*s++**); **(\*p->str)++** increments whatever **str** points to; and **\*p++->str** increments **p** after accessing whatever **str** points to.

**ARRAYS OF STRUCTURES**

struct marks

{

int marks\_in\_phy;

int marks\_in\_maths;

} student[2];

Declares a structure type **marks**, defines an array **student** of structure of this type, and sets aside storage for them. Each element of the array is a structure.

student[] =

{

45,32,

20,30

}

This initializes the array **student** of structure type **marks.** The initializers are listed in pairs corresponding to the structure members.

**Sizeof OPERATOR**

C provides a compile-time unary operator called **sizeof** that can be used to compute the size of any object. The expressions

sizeof *object*

and

sizeof( *type name* )

yield an integer equal to the size of the specified **object** or **type** in bytes. An **object** can be a variable or array or structure. A type name can be the name of a basic type like int or double, or a derived type like a structure or a pointer.

struct name

{

char firstname[50];

char lastname[50];

} student;

This declares **student** of structure type **marks.**

sizeof student /\* this uses “sizeof object” \*/

sizeof( name ) /\* this uses “sizeof( type name )” \*/

Don’t assume, however, that the size of a structure is the sum of the sizes of its members. Because of the alignment requirements for different objects, there may be unnamed “holes” in a structure. Thus, for instance, if a char is one byte and an int four bytes, the structure

struct

{

char c;

int i;

};

might well require eight bytes, not five. The **sizeof** operator returns the proper value.

**Typedef**

C provides a facility called **typedef** for creating new data type names.

typedef int Length;

makes the name **Length** a synonym for **int.** The type **Length** can be used in declarations, casts, etc., in exactly the same ways that the type **int** can be:

Length len, maxlen;

Length \*lengths[];

Similarly, the declaration

typedef char \*String;

makes **String** a synonym for **char \*** or character pointer, which may be then used in declarations and casts.

It must be emphasized that a **typedef** declaration does not create a new type in any sense; it merely adds a new name for some existing type. Nor are any new semantics: variables declared this way have exactly the same properties as variables whose declarations are spelled out explicitly.

**Unions**

A **union**is a variable that may hold (at different times) objects of different types and sizes,

with the compiler keeping track of size and alignment requirements. The purpose of a **union**- a single variable that can legitimately hold any of one of several types.

The syntax is based on structures:

union u\_tag

{

int ival;

float fval;

char \*sval;

} u;

The variable **u** will be large enough to hold the largest of the three types; the specific size is

implementation-dependent.

Syntactically, members of a **union** are accessed as

union-name.member

or

union-pointer->member

just as for structures. Unions may occur within structures and arrays, and vice versa. The notation for accessing a member of a **union** in a structure (or vice versa) is identical to that for nested structures.

For example, in the structure array defined by

Struct

{

char \*name;

int flags;

int utype;

union

{

int ival;

float fval;

char \*sval;

} u;

} symtab[NSYM];

the member **ival** is referred to as

symtab[i].u.ival

and the first character of the string **sval** by either of

\*symtab[i].u.sval

symtab[i].u.sval[0]

A **union** may only be initialized with a value of the type of its first member; thus **union u**

described above can only be initialized with an integer value.