***STRUCTURES***

* Defines a new type
  + Represents structured collection of data
    - Different type is possible
* Ex: Planet type
  + Name
  + Diameter
  + Number of moons
  + Number of years to complete one solar orbit
  + Number of hours to complete one rotation
* DEFINITION:

typedef struct{

char name[20];

double diameter;

int moons;

double orbit\_time,

rotation\_time;

}planet\_t;

red : declaring a structure

typedef and planet\_t are for renaming struct as a type name.

planet\_t my\_planet;

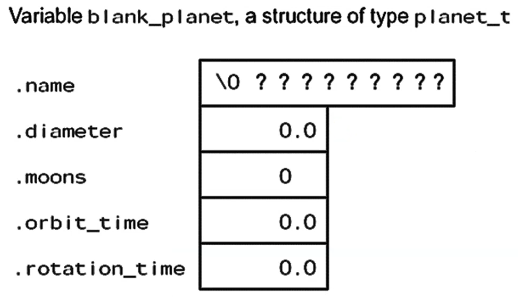
* A name chosen for a component of one structure may be the same as the name of a component of another structure or the same as the name of a variable.
* The typedef statement itself allocates no memory
* A variable declaration is required to allocate storage space for a structured data object.

planet\_t current\_planet,

previous\_planet,

blank\_planet = {“”, 0.0, 0, 0.0, 0.0};

*(each one is 48 bytes, one after another in the memory, continuously allocated)*



* A structure can contain another structure as component

typedef struct{

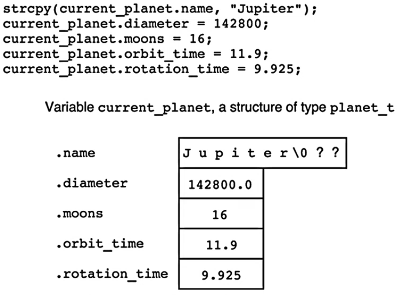
double diameter;

planet\_t planets[9];

char galaxy[STRSIZ];

}solar\_sys\_t;

* ASSIGNING VALUES
  + Direct component selection operator: a dot (.) placed between a structure type variable and a component name to create a reference to the component.



assignment (=) supports only basic types (enumerated, char, double, int, float, logical, pointer/address)

In arrays and structs, you can only use (=) in the declaration.

* typedef {1} {2};
  + 1 : type itself
  + 2 : new name of the type

You can also use structs like this:

struct{

int a;

}v;

v.a = 5;

red: type

You can copy one structure to another:

previous\_planet = current\_planet;

Direct component operator (.) has the highest precedence.

* When a structured variable is passed as an input argument to a function, all of its component values are copied into the components of the function’s corresponding formal parameter.

typedef struct{double r, i;} complex;

double mag(complex c){

return sqrt(c.r \* c.r + c.i \* c.i);

}

complex a = {1.0, -1.0};

mag(a);

|  |  |
| --- | --- |
| **a** .r | 1.0 |
| .i | -1.0 |
|  |  |
|  |  |
| **c**  .r | 1.0 |
| .i | -1.0 |
|  |  |

* When such a variable is used as an output argument, the address-of operator must be applied.

void add(complex x, complex y, complex \*z){

(\*z).i OR \*z.i --> you can use both. (.) operator has the highest priority

OR z -> r

}

complex a = {1.0, -1.0};

complex b = {0.0, 2.0};

complex c;

add(a, b, &c);

|  |  |
| --- | --- |
| **a** .r | 1.0 |
| .i | -1.0 |
| **b** .r | 0.0 |
| .i | 2.0 |
| **c (\*z)** .r | ? |
| .i | ? |
| **x** .r | 1.0 |
| .i | -1.0 |
| **y** .r | 0.0 |
| .i | 2.0 |
| **z** | {address of c} |

void add(const complex \*x,

const complex \*y,

complex \*z){

z -> r = x -> r + y -> r;

z -> i = x -> i + y -> i;

}

We used const for \*x and \*y and didn’t use for \*z because we are manipulating whatever \*z is pointing at and whatever \*y and \*z are pointing at shouldn’t be changed.

const means use this only for input. I don’t want to copy it because copying is expensive.

You can do “x = &…”

You can do this if you want to change what’s pointing x at:

complex \*newp;

newp = x;

newp -> r = …

void add(const complex x,

complex y,

complex \*z){

x.imag += y.imag;

z -> r = x.real;

z -> r = x.r + y.r;

z -> i = x.i + y.i;

}

Having const in front of complex x prevent first operation. const is protecting the caller, not within the function itself so this is overkilling in this case.

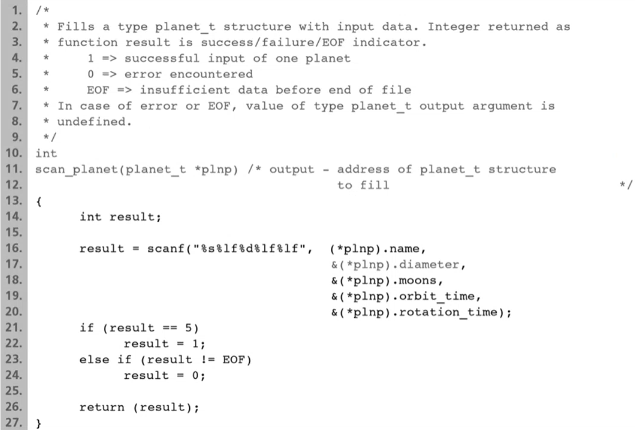
On caller’s perspective, const only makes sense if you have a pointer argument. Because without pointer, you make copy, it’s none of caller’s business.

With using pointer in the function parameter, we are just copying the address which is either 4 bytes or 8 bytes depending on the system used.

* The equality and inequality operators cannot be applied to a structured types as a unit.

complex a, b;

if (a > b) … YOU CANNOT SAY THAT

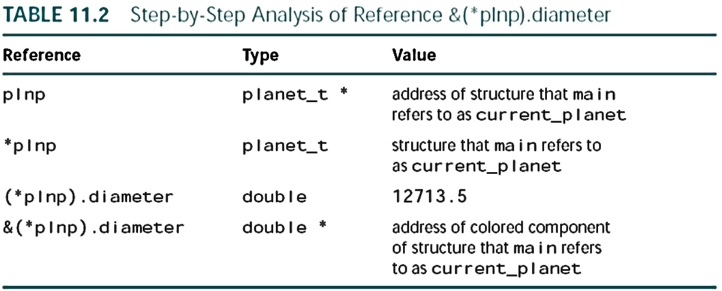


You can use like : &plnp -> diameter etc. Because & sign takes the address of diameter, not the plnp, because “->” has priority over “&”.

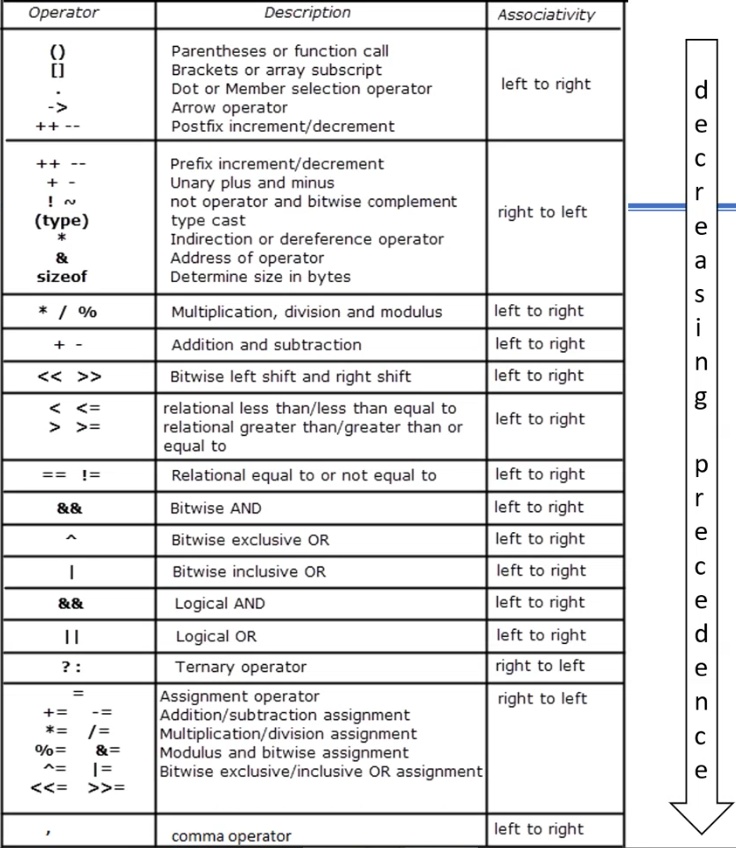
&(a.imag) = &a.imag : a is not pointer

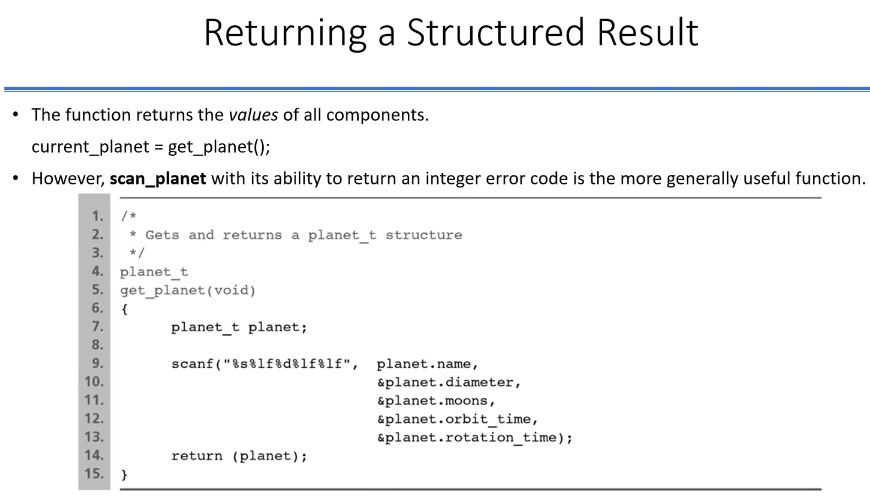
&a->imag = &(a->real) = &(\*a.imag) : a is pointer

plnp.diameter : you cannot do that, you are trying to go to componant of a pointer, there is no such thing



&\*plnp.diameter would select the diameter first and then follow the pointer in plnp to the structure.



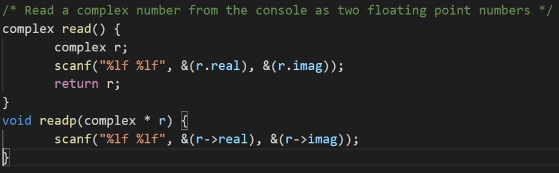


planet\_t a;

a = get\_planet();

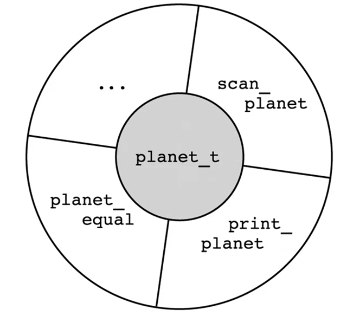
This is expensive because we are copying 20+8+4+8+8=48 bytes everytime we call the function.

If your range is not too broad, you can use unsigned char instead of int. For example hour of a day (0 - 24).



**Abstract Data Type (ADT)**

* a data type combined with a set of basic operations
* We must also provide basic operations for manipulating our own data types.
* If we take the time to define enough basic operations for a structure type, we then find it possible to think about a related problem at a higher level of abstraction.

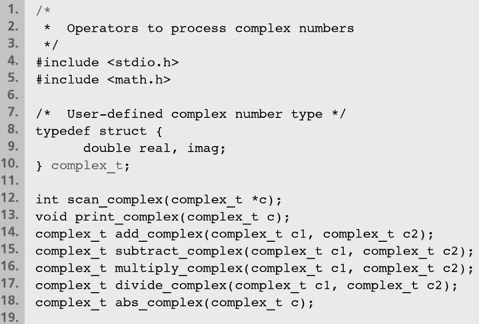


Once these functions are in header file, in a library, I can take the header file and without knowing what’s going on inside the print\_planet function, but I know the impact of it, I know what it does, I can use these functions just by looking them.

Sometimes you might have an ADT that you might be able to use, or sometimes you look into a problem and you say “This problem would have been solved easier if I had this ADT here and that ADT here.”. After the analysis you decide that “Okay, these two ADTs are gonna make my problem solving very easy.”. Then you would implement them and use them in solving that larger problem.

JUST SIMPLIFY THIS WAY:

You have a set of values that you manipulate with a set of functions and when you look at this from a high perspective, availibity of those functions might be useful for you to solve a problem then this would be a good candidate for an ADT. Knowing some of these ADTs that you already known (like a set ADT, queue ADT, stack ADT…) then you already have most of the tools (functions) alongside with those types that you can use them in problem solving for many problems.



**Parallel Arrays & Array of Structures**

* Parallel Arrays

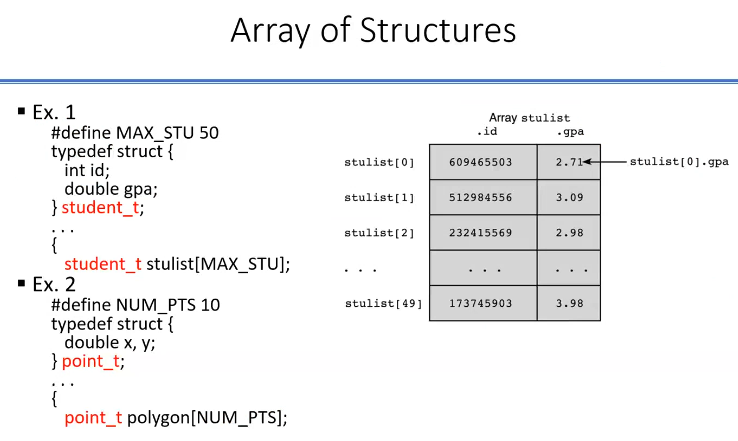
int id[50]; /\* id numbers and \*/

double gpa[50]; /\* gpa’s of up to 50 students \*/

double x[NUM\_PTS], /\* (x, y) coordinates of \*/

y[NUM\_PTS]; /\* up to NUM\_PTS points \*/

* Array of Structures
  + A more natural and convenient organization is to group the information in a structure whose type we define.



HOW MULTIDIMENSIONAL ARRAY LOOKS IN THE MEMORY?

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| . |
| . |
| . |

int b[3][2][4];

b[0][0]

b[0]

***UNIONS (USER DEFINED TYPE)***

There were basic types:

* int, char, …
* Boolean -> int
* enumerated
* pointer

And user defined types (any new type can be obtained from a set operation on existing types):

* structures -> combination of various types (cartesian product : set operation 🡪 A x B )
* unions -> union of 2 or more sets to form a new set of values ( A U B)

Union Types

newtype x;

float

int

now x is either in or float.

x is data that can be interpreted in 2 ways.

* Union : Data object that can be interpreted in a variety of ways.
  + EX: a number can be real number (double) or an integer (int)
* Allows one chunk of memory to be interpreted in multiple ways

typedef union {

int wears\_wig;

char color[20];

}hair\_t;

hair\_t hair\_data;

* hair\_data does not contain both wears\_wig and color components, but either a wears\_wig component referenced by hair\_data.wears\_wig, or a color component referenced by hair\_data.color.
* The amount of memory is determined by the largest component of the union. So 20 bytes is the memory allocation for hair\_data.
* If your hair\_data variable is holding an integer, it will be using first 4 bytes. If it is holding an array of characters, it will be using the entire thing.
* hair\_data variable can only hold one of the components at a given time.
* How to determine interpretation?
  + How to determine whether to use wears\_wig or color?

hair\_t his\_hair;

strcpy(his\_hair.color, “red”);

Now we know that his\_hair variable is pointing to string value, not to int value.

I can call his\_hair.color as my value.

I can’t do his\_hair.wears\_wig. Because this is not defined. But if you change your mind, you can do:

his\_hair.wears\_wig = 1;

------------------------------------------------------------------------------------------------------------------------------------------

typedef union{

int i;

float x;

}mytype;

mytype ix;

ix.i = 10;

…

ix.x = 20.0;

From now on, if you try to reach ix.x, but you use ix.i, semantically this is wrong but compiler is gonna try to interpret whatever the bit representation of 20.0, it is gonna be interpreted as integer. Probably that will be some weird number.

------------------------------------------------------------------------------------------------------------------------------------------

typedef union{

|  |
| --- |
| … |
| ‘a’ |
| ‘b’ |
| ‘c’ |
| ‘d’ |
| ‘e’ |
| … |

char x[5];

char y[3];

}mt;

mt a;

a.x[0] = ‘a’;

a.x[1] = ‘b’;

a.x[2] = ‘c’;

a.x[3] = ‘d’;

a.x[4] = ‘e’;

printf(“%c”, a.y[1]); 🡪 ‘b’

typedef union {

int wears\_wig;

char color[20];

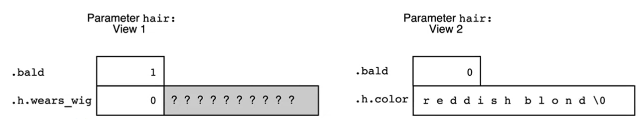
}hair\_t;

typedef struct {

int bald;

hair\_t h;

}hair\_info\_t;



Semantically, knowing what the value of a union variable is at a given time may not be that easy. You need to look at the context.

Referencing the appropriate union component is always the programmer’s responsibility; C can do no checking of the validity of such a component reference.

Using 2 integer or 2 double or etc. in union is not a good idea.

Compute Area and Perimeter

/\*

Computes the area and perimeter of a variety of geometric figures.

\*/

#include <stdio.h>

#define PI 3.14159

/\* Types defining the components needed to represent each shape. \*/

typedef struct{ (SIZE : 24 BYTES)

double area, circumference, radius;

}circle\_t;

typedef struct{ (SIZE : 32 BYTES)

double area, perimeter, width, height;

}rectangle\_t;

typedef struct{ (SIZE : 24 BYTES)

double area, perimeter, side;

}square\_t;

/\* Type of a structure that can be interpreted a different way for each shape \*/

typedef union{ (SIZE : 32 BYTES)

circle\_t circle;

rectangle\_t rectangle;

square\_t square;

}figure\_data\_t;

/\* Type containing a structure with multiple interpretations along with a component whose value indicates the current valid interpretation \*/

typedef struct{ (SIZE : 33 BYTES)

char shape; (c, r ,s)

figure\_data\_t fig;

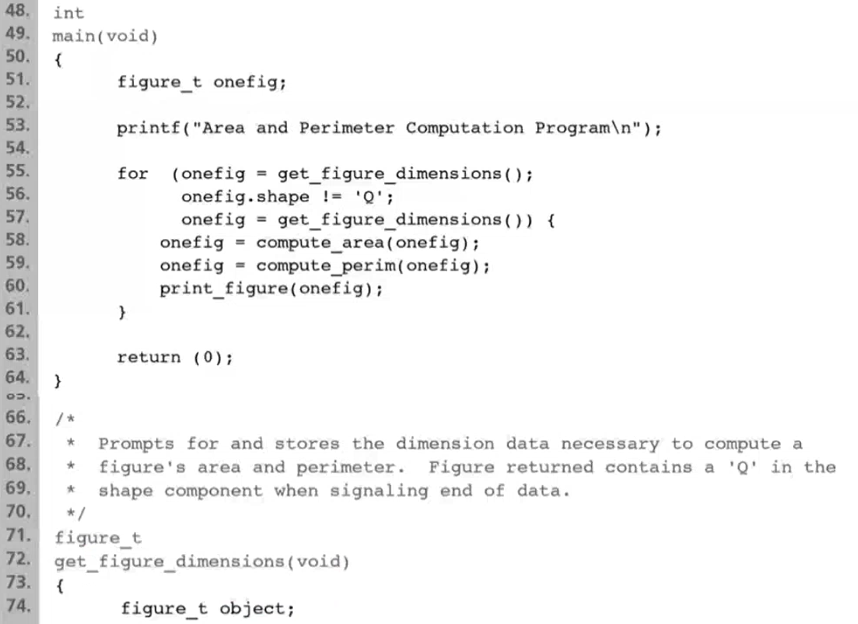
}figure\_t;

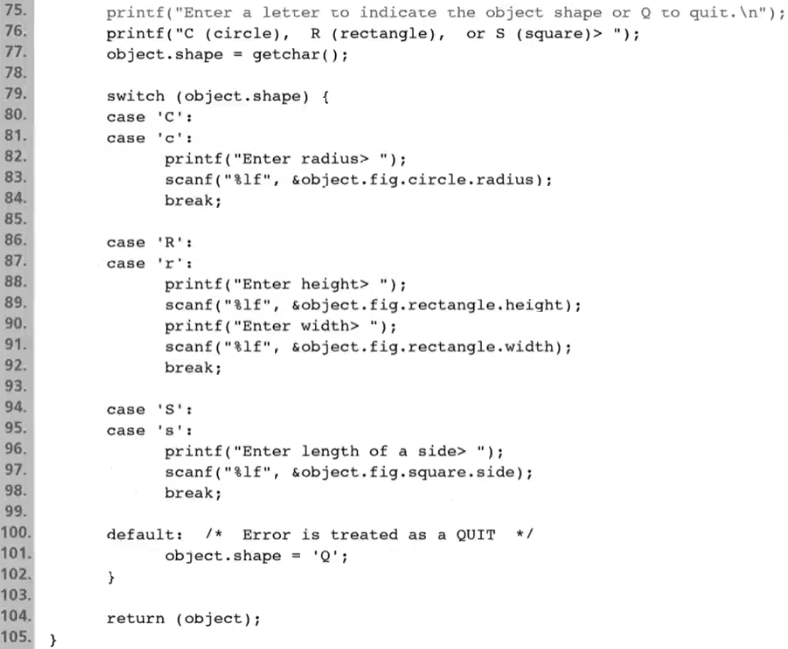
figure\_t get\_figure\_dimensions(void);

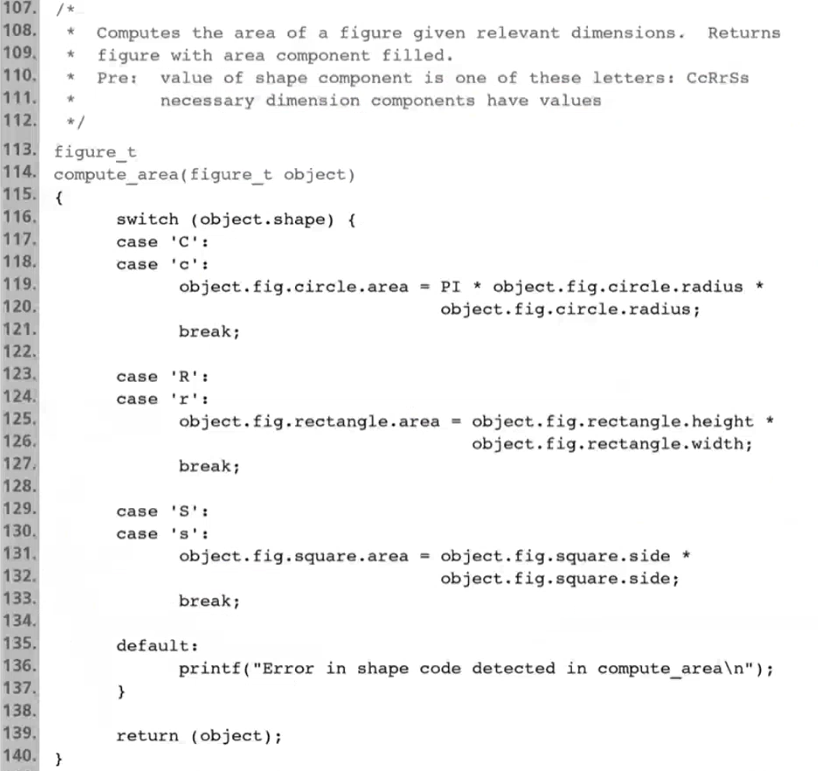
figure\_t compute\_area(figure\_t object);

figure\_t compute\_perim(figure\_t object);

void print\_figure(figure\_t object);







object.fig.circle.radius

struct union struct double

struct vs typedef struct

Basic use of struct:

struct {int x, y;}var;

Named struct:

struct S {int x, y;}; 🡪 in function: int(struct S my\_input);

struct S var;

Typedef and named struct:

struct S {int x, y;};

typedef struct S ST; /\* STs can be S \*/

ST var;

Another way:

typedef struct S{int x, y;} ST; 🡪 You can put S instead of STs

ST var;

Another way:

typedef struct{int x, y;} ST;

ST var;

struct S {int x, y;};

typedef struct S ST;

ST var;

Note that S is only defined within the context of struct.

But you cannot make another S struct.

But we can use the name again:

struct S {int x, y;};

typedef struct S ST;

ST var;

void S(int a)… /\* OK to define S again \*/

However, we ST is in the global namespace:

struct S {int x, y;};

typedef struct S ST;

ST var;

void ST(int a)… /\* ERROR \*/

struct complex1{

double i, r;

};

struct complex2{

double i, r;

char name[1];

};

typedef struct {

double i, r;

char name[8];

}complex3;

typedef struct{

char c;

char c2;

}chart;

printf(“%ld”, sizeof(struct complex1)); 🡪 16

printf(“%ld”, sizeof(struct complex2)); 🡪 24

printf(“%ld”, sizeof(complex2)); 🡪 24

printf(“%ld”, sizeof(chart)); 🡪 2

sizeof gives you how many bytes would that type require if you declare a variable of that type.

struct complex2 should give 17 but it gives 24, why?

Memory on systems, depending on what type of machine you have (32 bit - 64 bit etc.), 64 bit machine means that memory is probably organized in a certain fashion. Memory has “words” (You will see this later). Word means that you can only allocate fix, minimum amount of memory and you can only increase that with certain increments. Those increments is 8 bytes in upper case. So when you talk about the cell, you should have (for efficiency and other purposes) only multiples of 8. In a 32 bit machine, this could be 4.

When CPU once load something from memory, it loads 8 byte at a time. Also it writes 8 byte at a time to the memory.

Although, for first 8 bytes, you get the actual value. That’s why chart is 2 bytes, not 8 bytes.

When you copy 30 bytes structure from one function to another, not exactly 30 bytes copied. There would be some padding area because of incrementing multiples of 8.

%x 🡪 unsigned int