ALGONQUIN COLLEGE

MODULE 6 : USER-DEFINED TYPES

Professor : Dave Houtman

Office: T323

Office Hrs: Monday 15:30 – 16:00

Wednesday 15:30 – 16:00

Friday 15:30 – 16:00

Email: houtmad@algonquincollege.com

6.1 User-Defined Types – Introduction

- Just as C comes with built-in data types (`int`, `char`, `float`…), the language also allows programmers to create their own composite types, called ***user-defined types*** (UDTs). A UDT is a data type derived from other data types, both primitive and reference. Hence a UDT can include, not just `int`s, `char`s, `float`s, etc., but also pointers, including pointers to arrays, functions, and even to other UDTs.

- Just as you can create an array of `int`s or `char`s, you can also create an array of a particular type of UDT (once it has been declared). Furthermore, you can create pointers to UDTs, and use pointer arithmetic to iterate through an array of UDTs.

6.1 User-Defined Types – Introduction

- UDTs fall into three catagories:
  - ***Enumerations*** allow the programmer to create data types of enumerated sets, essentially a list of variables whose associated value is automatically incremented by one.
  - ***Structures*** package different primitive data types together in a convenient fashion that allows the programmer to access each element of the structure individually using the '.' notation. Structures allow the user to create something approximating an OOP class, but without public, private and protected access modifiers.
  - ***Unions*** allow the programmer to reference elements inside a data type by *overlaying* two or more types on top of one another. Thus four `chars` and one `int` may occupy the same location in memory, and be referenced by different names. This even extends to the level of individual bits inside the union.
- Also, you can combine all of the above capabilities into a single UDT

6.1 User-Defined Types - Introduction

- All UDTs have the same overall structure, regardless of which type of UDT they are:

```
keyword tag {
    UDT_elements (chars, ints, pointers, etc.)
    ...
} var_name1, var_name2,...;
```

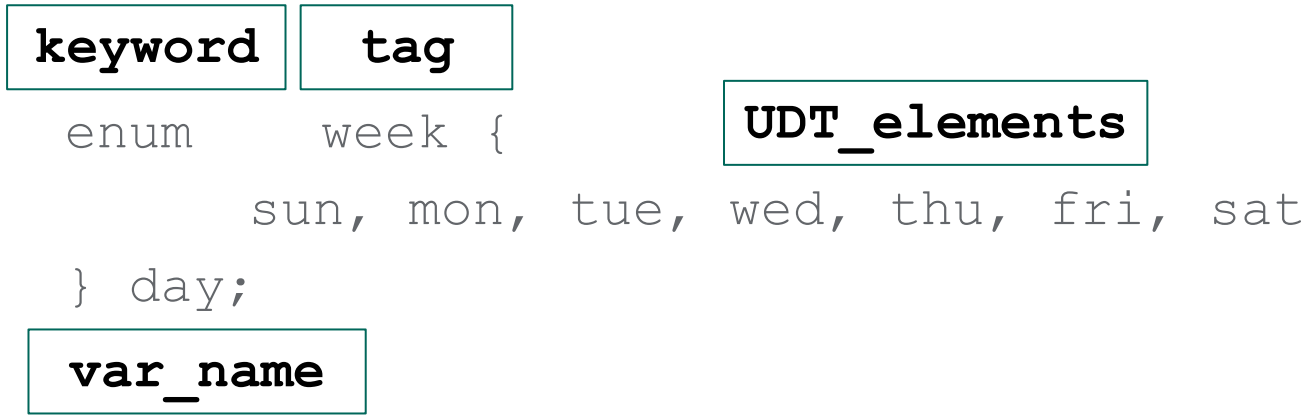
where:

- **keyword** is one of `enum`, `struct`, or `union`;
- **tag** is the name of the type that *you* supply; it is the equivalent of `int`, `float`, `char`, etc. in a primitive data type;
- **var_name** are variable names that you supply, each of the type **tag**.

- As we'll see, either the `tag` or the `var_name` can be omitted depending on the circumstances—but not both in the same declaration.

6.2 Enumeration Types

- The simplest UDT is an `enum`, a variable that assigns an iterated sequence of values to a set of variable names. For example, the declaration:



creates a UDT of type `week`, which is defined as the set of values between the braces. `day` is a variable of type `week`; it can take any of the values in the braces.

6.2 Enumeration Types

- Since a UDT is just a data type like any other, we can create the type in one place, and then use it in a declaration elsewhere. For example, if we create the type

keyword	tag

```
enum  week {
        sun, mon, tue, wed, thu, fri, sat};
};
```

We can use it to make a declaration:

```
enum week day;
```

keyword	tag	var_name

6.2 Enumeration Types

- An `enum` works as follows. Each UDT element inside the braces is called an ***enumerator***. Each enumerator is declared as a `const int`, and is assigned a value, starting from zero, and incremented sequentially. So you might think of the above declaration as creating the following set of variables:

```
const int sun = 0;
const int mon = 1;
const int tue = 2;
...
const int sat = 6;
```

6.2 Enumeration Types

- The difference is that all this information is packed into one UDT; whenever you declare

```
enum week day1, day2, today;
```

each `day` variable can be set to (and only to) the enumerators declared in the enumeration type.

- Each variable can be set to one of the values in the enumeration type and used in an expression exactly the same as any primitive data type:

```
day1 = mon; day2 = tues;
void setDay(enum week);   //forward declar'tn

if (day1 < day2)
     printf("The 2nd day is day %d", day2);
...
```

6.2 Enumeration Types

- By default, the first enumerator in the braces is set to zero, and each succeeding value is automatically incremented by one. This can be overridden at any point, e.g.

```
enum week{sun=6, mon=0, tue, wed, thu, fri, sat};
```

this sets `sun = 6, mon = 0, tue = 1, wed = 2…, sat = 5`

- In actual fact, we usually don't worry about the actual value of an `enum` after it is set. Part of the purpose of `enum`s is to use the element name rather than the number it stands for.

6.2 Enumeration Types

- Since an enumeration type is just another, new kind of type, we can use `typedef` to simplify declarations:

```
enum week {sun, mon, tue, wed, thu, fri, sat};
typedef enum week week;
```

This says: use `week` (the second `week` in the statement) as a shortcut for the enumeration type `week` (the first `week`). This is just a semantic shortcut to ease the use of the `week` data type later on. For example, we can now declare `week` variables using:

```
week day1, day2;      // which looks like
                      // a proper declaration
```

instead of

```
enum week day1, day2;    // which looks a bit odd
```

6.2 Enumeration Types

- Enumeration types should be used to help keep code orderly. They are particularly useful when there is a natural order to a sequence of items:

```
enum months {jan=1, feb, mar, apr, may, jun, jul,
                    aug, sep, oct, nov, dec};

enum judoBelts {white, yellow, orange, green,
                   blue,brown, black};

enum elements {H = 1, He, Li, Be, B, C, N, O, F …};

enum mattressSize {single, double, queen, king};


enum olympicMedals {none, bronze, silver, gold};

enum resistorColourCode {black, brown, red, orange,
    yellow, green, blue, violet, grey, white};
```

6.2 Enumeration Types

- Variables may be declared at the same time the enumeration type is itself specified, for example;

```
enum decision {maybe = -1, no, yes} outcome1,
                                    outcome2;
```

This declares `outcome1` and `outcome2` as being of type `decision`. Both outcomes can have one of three values only: `maybe` (= -1), `no` (=0), and `yes` (=+1). Note that these values extend the associated 'true' and 'false' values built in to the C language. Assuming `outcome1` is set to one of the three enumerators, we can use it as follows to call on one of two functions:

```
result = outcome1 ? itsPossible(): noWay();
```

6.2 Enumeration Types

- If you only need to create new variables, and you won't be using the enumeration tag in any way, you can forgo the tag name altogether. For example, in the enumeration type,

```
enum decision {maybe = -1, no, yes} outcome1,
                                    outcome2;
```

nothing is lost by dropping `decision` from the declaration:

```
enum {maybe = -1, no, yes} outcome1, outcome2;
```

except that, not having the tag, you can't declare any other new variables with that name. Also, you can't pass a variable of that 'type' to a function, because that 'type' no longer exists.

6.2 Enumeration Types

- Since an enumeration type is a legitimate type, it can be both passed to and returned from a function. However, because the native data type of an enumerator is a `const int`, you can't perform math on a enumeration type directly…

```
enum rank {private, corporal, sergent, captain
   major, colonel, general} newRank, oldRank;

typedef enum rank rank; // rank = enum rank
rank promotion(rank);   // forward declaration
   ...
oldRank = private;
newRank = promotion(oldRank);  //general?
   ...
rank promotion(rank soldier){
   return(soldier += 6);  // not allowed
}
```

6.2 Enumeration Types

- However, with some clever casting, it is possible to convert an `enum` type into something you can perform math on, and then convert is back to its UDT:

```
enum rank {private, corporal, sergeant, captain,
   major, colonel, general} newRank, oldRank;

typedef enum rank rank;
rank promotion(rank);   //forward declaration

   ...
oldRank = private;
newRank = promotion(oldRank); // congrats!
   ...
rank promotion(rank soldier){
   return((rank)(((int)soldier)+6)); //OK
}
```

6.2 Enumeration Types

Note: If you don't use the `typedef` trick, then you must use `enum` in front of *every* reference to your enumeration tag name. For example, without the `typedef` trick, the previous example becomes:

```
enum rank {private, corporal, sergeant, captain,
  major, colonel, general} newRank, oldRank;

enum rank promotion(enum rank soldier);
   ...
oldRank = private;
newRank = promotion(oldRank);
   ...
enum rank promotion(enum rank soldier){
   return((enum rank)(((int)soldier)+1));
}
```

6.3 Structures - Introduction

- Structures form the second, and most powerful, UDT in C.
- Structures allow you to aggregate a 'package' of different data types, which may include any combination of:
  - Primitive data types: `int`, `float`, `char`, …
  - Arrays
  - Pointers
  - Functions
  - Any other UDT, including `enum`s, `union`s, and `structure`s
- `enum`s could be replaced with `#DEFINE` statements, and `union`s don't get a lot of use—they aren't strictly essential. But some form of a `structure` is essential to any advanced computer language.

6.3 Structures

- The following example declares a structure for a playing card. Each card has only two features: its suit (clubs, spades, hearts, or diamonds) and its value, or pips (ace, two, three…Jack, Queen, King). A reasonable structure to hold this information would therefore be:

```
struct card{
  int pips;  // 1 = ace,… 11 = Jack, 12 = Queen…
  char suit; // clubs, spades, hearts, diamonds
};
```

We then declare our `card` variable as:

```
struct card c1, c2;
```

6.3 Structures

- Alternately, we can create our new card variables along with the `struct`:

```
struct card{
  int pips;  // 1 = ace, 11 = Jack, 12 = Queen…
  char suit; // clubs, spades, hearts, diamonds
} c1, c2;    // c1, c2 are of type card
```

- As before, we can leave the `card` tag name out entirely, if we don't plan to use that data type afterward:

```
struct{      // missing tag name is okay here
  int pips;  // 1 = ace,… 11 = Jack, 12 = Queen…
  char suit; // clubs, spades, hearts, diamonds
} c1, c2;    // what type? Doesn't matter.
             // we've got our variables c1 and c2
             // based on the struct shown
```

6.3 Structures

- Or we can use `typedef` to create a shortcut to our card data type, using the trick seen earlier:

```
typedef struct card{
  int pips;  // 1 = ace, 11 = Jack, 12 = Queen…
  char suit; // clubs, spades, hearts, diamonds
} card;

card c1, c2; // looks like a 'normal' declaration
```

Note that this use of `typedef` is really no different writing

```
struct card{
  int pips;  // 1 = ace, 11 = Jack, 12 = Queen…
  char suit; // clubs, spades, hearts, diamonds
} card;

typedef struct card card;
card c1, c2;
```

6.3 Structures

- Recall that with arrays, you can (and should) initialize all the values of an array using `ar[] = {0};` the same trick works with structures, and should be used to ensure that `all` *the members* of a structure are properly initialized:

```
struct card{
    int pips;
    char suit;
} card1 = {0};    // ensures all members
                  // of card1 are set to 0
```

This sets `pips` and `suit` to 0.

If a structure containing an array of chars and it is set to 0, then this initializes the 'string' being pointed to to the termination character, `'\0'` (the ASCII `NUL` character). So initializing your values to 0 works as a proper termination default for pretty much every data type.

6.3 Structures

- Structure members can be assigned in the same way that arrays are assigned, using the curly braces, either via:

```
struct card{
    int pips;
    char suit;
} c1 = {1,'S'}, c2 = {8,'H'};
```

Or, using the `typedef` trick:

```
typedef struct card{
    int pips;
    char suit;
} card;
card c1={1,'S'}, c2={8,'H'};
```

6.3 Structures

- To access an element within the card structure *after* the point of declaration, you case use the 'dot' specifier. For example, assuming we wish to make `c1` the jack of diamonds, we could write:

```
c1.pips = 11;      // The Jack
c1.suit = "D";     // of Diamonds
```

Note: this '.' notation is where C++, and later, Java gets their dot notation to access and object's properties and methods

6.3 Structures

- If we wish to create an array of card structures, then we can use

```
struct card{
  int pips;  // 1 = ace, 11 = Jack, 12 = Queen…
  char suit; // clubs, spades, hearts, diamonds
} cards[52];
```

Alternately, using the `typdef` trick:

```
typedef struct card{
  int pips;  // 1 = ace, 11 = Jack, 12 = Queen…
  char suit; // clubs, spades, hearts, diamonds
} card;

card cards[52] = {0}; //cards is an array of 52
                      //card types, with ea. mem-
                      //ber of ea. card set to 0
```

6.3 Structures

- Note that we can use other UDTs within the structure. Since every card in a standard deck will have one of four suits and thirteen values, we can write our structure as:

```
enum pips {ACE=1, TWO, THREE, FOUR, FIVE, SIX,
  SEVEN, EIGHT, NINE, TEN, JACK,  QUEEN, KING};
enum suit {SPADES, CLUBS, HEARTS, DIAMONDS};

struct cardtype{
    enum pips p;
    enum suit s;
} cards[52];
```

To initialize our deck, we can use:

```
for (int cardCtr = 0; cardCtr < 52; cardCtr++){
   cards[cardCtr].p=(enum pips)((cardCtr % 13)+1);
   cards[cardCtr].s=(enum suit)(cardCtr/13);
}
```

6.3 Structures

- Or, using `typedef` again:

```
typedef enum {ACE=1, TWO, THREE, FOUR, FIVE, SIX,
SEVEN, EIGHT, NINE, TEN, JACK,  QUEEN, KING} pips;

typedef enum {SPADES, CLUBS, HEARTS, DIAMONDS} suit;

typedef struct cardtype{
    pips p;
    suit s;
} card;
card cards[52];
```

Now, to initialize our deck, we can use:

```
for (int cardCtr = 0; cardCtr < 52; cardCtr++){
   cards[cardCtr].p=(pips)((cardCtr % 13)+1);
   cards[cardCtr].s=(suit)(cardCtr/13);
}
```

6.3 Structures

- Note the use of `(pips)` and `(suit)` to cast our values to pips and suit. To do this, `pips` and `suit` must be declared *outside* `cardtype`
- However, since enumeration types are constant integers and `cardCtr` is already an integer, we could simplify this code to:

```
struct card{
  enum {ACE=1, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
        EIGHT, NINE, TEN, JACK, QUEEN, KING} p;
  enum {SPADES, CLUBS, HEARTS, DIAMONDS} s;
} cards[52];

...
int cardCtr;
...
for (int cardCtr = 0; cardCtr < 52; cardCtr++){
   cards[cardCtr].p = (const)((cardCtr % 13)+1);
   cards[cardCtr].s = (const)(cardCtr/13);
}
```

thus bypassing the need to declare `p` and `s` altogether.

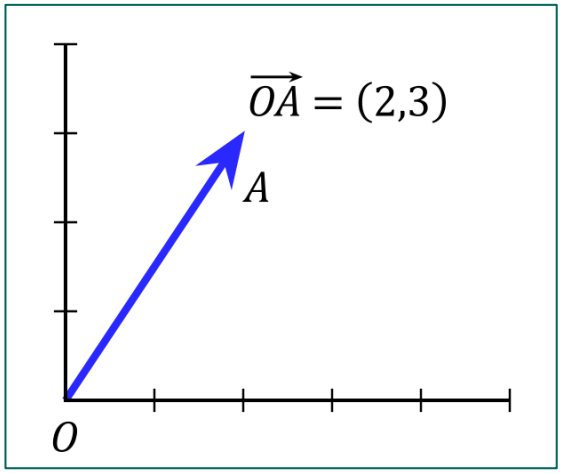
6.3 Structures

- In this last example, since we didn't require a cast to `pips` and `suit`, we could put the actual declaration of these types inside `cardtype`;

- This attention to detail may seem a bit excessive and confusing, but in fact it's no different than what you do in Java when you set up a class and derive an array of objects from it. The only real difference is that in C, the declarations are a bit more packed.

- In Java, things can get even more complicated, since you have inheritance and access modifiers (public, private, and protected) to deal with; that doesn't happen in C (no objects!).

Example 16 : The 2D-Vector Structure



Program Description:

A vector is a mathematical quantity having both magnitude and direction. Vectors are usually represented as a pair of numbers, e.g. {2,3}. This can be represented graphically in a 2-dimensional grid as an arrow drawn from the origin at {0,0} to the point {2,3}.

In this example, a structure is declared which holds the contents of a 2D vector, along with a small library of functions designed to perform simple actions such as adding and subtracting vectors, finding the magnitude of a vector, etc.

Example 16: The 2-D Vector Structure

```
houtmad@ubuntu:~/examples$ Example16
The length of v1 is: 3.162278
The sum of v1 and v2 has components 2.000000, 1.000000
houtmad@ubuntu:~/examples$
```

The code that generates the above output is:

```
#include <stdio.h>
#include "vector.h"

int main(void){
   vector v1 = {1.0, 3.0},  v2 = {1.0, -2.0};

   double length = magnitude(v1);
   printf("The length of v1 is: %lf\n", length);

   vector v = addVectors(v1, v2);
   printf("The sum of v1 and v2 has components"
       "%lf, %lf\n", v.x, v.y);
}
```

Example 16: The 2-D Vector Structure

First, the `vector.h` file is given by:

```
typedef struct vector{
   double x, y;
} vector;

extern double getXComponent(vector);
extern double getYComponent (vector);
extern double magnitude (vector);
extern unsigned int isOrthogonal(vector, vector);
extern vector addVectors(vector, vector);
```

And in the `main.c` file:

Example 16: The 2-D Vector Structure

```
#include "vector.h"
#include <math.h>   // used for sqrt in magnitude()

double getXComponent(vector V){return (V.x);}
double getYComponent (vector V){return (V.y);}

double magnitude (vector V){
   double mag_sqrd = (V.x * V.x)+(V.y * V.y);
   return (sqrt(mag_sqrd));  // needs math.h
}

vector addVectors(vector V1, vector V2){
   static vector vRet;
   vRet.x = V1.x + V2.x;
   vRet.y = V1.y + V2.y;
   return vRet;
}

unsigned int isOrthogonal(vector V1, vector V2){
   double isOrtho = (V1.x * V2.x) + (V1.y * V2.y);
   return (isOrtho != 0);
}
```

6.3 Structures

- Structures can contain any combination of primitive data types, arrays pointers, and even other structures. However, when initializing the elements of a pointer, its important to remember the following rules:

1. `Ar` and `"abc"` return the address of the first element in the array (of `int`s, `char`s, etc.) Therefore, they are always rvalues. They can be assigned to a pointer, but they cannot always be treated like pointers, since they cannot be reassigned a new value.

2. A pointer to an array of chars is a special case; it can be assigned a string after declaration. Hence `pAr = "abc"` is allowed, but `Ar[]="abc"` is not. This applies to pointers and arrays inside structures as well, as we'll see shortly.

3. All other 'late' assignments are prohibited, unless they are done piecewise, i.e. `Ar[0] = 1; Ar[1] = 2; Ar[2] = 4;` etc., or via special functions like `strcpy()`.

6.3 Structures

- Consider the following example:

```
struct address {
    char *pStreet;
    int   aptNum;
    char *pCityAndProv;
    char *pPostalCode;
};

typedef struct address address;
```

To create and initialize addresses based on this structure, we could use:

```
address homeAddress={0}, busAddress={0};
```

6.3 Structures

- To initialize address variables at the time of declaration, we've used

```
address homeAddress={0}, busAddress={0};
```

This ensures that every value in the structure is initialized to 0. Most particularly, the pointers are defaulted to 0, which is recognized by C as `NULL`, a special value that essentially means: "the pointer does not point to anything"—just what we want, if we don't know what will be stored in the structure initially.

6.3 Structures

- If we know what we want to put inside our structures, we can assign a set of values to our address variables at the point of declaration, we could also use:

```
address homeAddress =
    {"First Ave.", 5, "Ottawa, ON", "K1S2T3"},
    busAddress =
    {"Carling Ave", 213, "Ottawa, ON", "K1B03A"};
```

Alternately, we could set each element individually using dot notation *after* the point of declaration:

```
homeAddress.pStreet = "First Ave.";
homeAddress.aptNum = 5;
homeAddress.pCityAndProv = "Ottawa, ON";
homeAddress.pPostalCode = "K1S2T3";
```

6.3 Structures

- If the structure is declared as follows

```
typedef struct address {
   char Street[20];
   int  aptNum;
   char CityAndProv[20];
   char PostalCode[7];
} address;
```

then you can use the same method to assign values to your structure elements as is shown above:

```
address homeAddress =
   {"First Ave.", 5, "Ottawa, ON", "K1S2T3"},
   busAddress =
   {"Carling Ave", 213, "Ottawa, ON", "K1B03A"};
```

6.3 Structures

However, if the structure is declared as follows, without definite array sizes

```
typedef struct address {
   char Street[];
   int  aptNum;
   char CityAndProv[];
   char PostalCode[];
} address;
```

then you cannot use the same method to assign values to your structure. This flags an error:

```
address homeAddress =
    {"First Ave.", 5, "Ottawa, ON", "K1S2T3"},
    //etc.
```

Cannot make late assignment to unsized array member of structure

6.3 Structures

- Furthermore, you cannot do the following (according to the 2nd rule in the list given previously)

```
homeAddress.Street = "First Ave."; //WRONG
homeAddress.aptNum = 5;
homeAddress.CityAndProv = "Ottawa, ON";//WRONG
homeAddress.PostalCode = "K1S2T3"; //WRONG
```

As before, pointers allow for a late assignment of strings, arrays do not.

Cannot make late assignment to unsized array

6.3 Structures

- As indicated above, we can create an array of structures as follows

```
struct address {
    char Street[20];
    int  aptNum;
    char CityAndProv[20];
} addr[20];
```

Assuming these elements were properly assigned, you can access them as you would for any array:

```
strcpy(addr[8].Street, "First Ave.");
addr[8].aptnum = 5;
strcpy(addr[8].CityAndProv, "Ottawa, ON.");
printf("Address is #%d %s %s\n",
    addr[8].aptnum, addr[8].Street,
    addr[8].CityAndProv);
```

6.3 Structures—pointers to structures

- Since a structure is just another data type, we can have a pointer to a structure (in exactly the same way as we can have a pointer to an `int`, `float`, `char`, etc.)

```
struct employee {
    char *last;
    char *first;
    int age;
} *pEmployee;
```

Caution: unassigned pointer can't be used to access members or it will give a segmentation fault

However, note that `pEmployee` is just a pointer. Despite all the words that have been written in the structure declaration, the entire result of this declaration is just a single 4 byte pointer (on a 32-bit architecture), the same as any other pointer to a primitive data type.

This declaration does not result in the creation of an actual structure, just a pointer to a structure.

6.3 Structures—pointers to structures

- To actually use a pointer to a structure, you must assign the address of an actual structure to the pointer first; you can't do much with an unassigned pointer otherwise.

```
struct employee{
   char *last;
   char *first;
   int age;
} Emp = {"Smith","Al",25}, *pEmployee = &Emp;
```

Here, `pEmployee` is set to the address of an existing `employee` structure (`Emp`). Without this step, any attempt to access any member of `pEmployee` would result in a segmentation fault.

6.3 Structures—pointers to structures

- In the case of a pointer to a structure, a special notation exists that allows us to access the members of the structure via the pointer. Called the ***member access operator***, it is represented by a special symbol,

```
->
```

(which is just a negative sign followed by a greater than sign.) Thus, if `*ptr` is an address to a structure, then the elements of that structure may be accessed using either:

```
(*ptr).member1
```

Or, using the member access operator, the equivalent expression is:

```
ptr->member1
```

Why does this seemingly-redundant operator exists at all? See: *http://stackoverflow.com/questions/13366083/why-does-the-arrow-operator-in-c-exist*.

6.3 Structures—pointers to structures

- So given the same `employee` structure declared above, we can use the member access operator notation to make the same assignments indirectly, using the pointer. Given:

```
struct employee{
   char *last;
   char *first;
   int age;
} Emp, *pEmployee = &Emp;
```

we can initialize our structure members as:

```
pEmployee->last = "Smith";
pEmployee->first = "Al;
pEmployee->age = 25;
```

6.3 Structures—pointers to structures

- Recall that pointers and arrays are almost equivalent, except that an array reserves space to hold the entire array, while a pointer just reserves space for a pointer. So the following is allowed

```
struct employee{
   char *last;
   char *first;
   int age;
} arEmployee[10];

arEmployee->last = "Smith";
arEmployee->first = "Al;
arEmployee->age = 25;
```

because `arEmployee` is the address of the first element of the array. Since actual space was assigned by this declaration, there is no danger of a seg fault.

6.3 Structures—pointers to structures

- However, some caution is in order when attempting to access the elements of an array of structures using traditional array notation. Given

```
struct employee {
    char *last;
    ...
} arEmployee[10];
```

again, you *can* access an array member (e.g. the second element) using

```
arEmployee[1].last
```

as before. Or you can treat the array like a pointer:

```
(*(arEmployee+1)).last = "Smith";
```

which is equivalent to:

```
(arEmployee+1)->last = "Smith";
```

6.3 Structures—pointers to structures

- However, given

```
struct employee {
   char *pStreet;
    ...
} arEmployee[10];
```

you *cannot* do the following:

```
arEmployee[1]->last = "Smith";
```

-> expects a pointer to a structure, not the structure itself

Recall that the -> notation expects a pointer to a structure before the member access operator. `arEmployee[1]` *is* a structure, *not* a pointer to a structure. To use -> with this array, you would need to have an element of the structure return its address (so it looks like a pointer):

```
(&arEmployee[1])->last = "Smith";
```

6.3 Structures—the size of a structure

- Internally, structures work almost the same as arrays, except that unlike arrays, structures contain variably-size members. We would therefore expect

```
struct dataTypeCollection{
    char a;          // address = start address
    short b;         //    + sizeof(char)
    int c;           //    + sizeof(short)
    long d;          //    + sizeof(int)
    long long e;     //    + sizeof(long)
    float f;         //    + sizeof(long long)
    double g;        //    + sizeof(float)
    char *h;         //    + sizeof(double)
 } type;
```

So the `sizeof(type)` should be 1+2+4+4+8+4+8+4 = 35. But this is not quite the case…

6.3 Structures—the size of a structure

- The C compiler is free to pad out structure members to align the bytes internally according to various hardware byte boundaries, improving access speed at the cost of a few extra bytes of memory. Using `type` from the previous slide, the following code produces the results shown at right:

```
printf("Start addr of char  = %p\n", &type.a);     //0xbfb3a8c0
printf("Start addr of short = %p\n", &type.b);     //0xbfb3a8c2
printf("Start addr of int   = %p\n", &type.c);     //0xbfb3a8c4
printf("Start addr of long  = %p\n", &type.d);     //0xbfb3a8c8
printf("Start addr of long long= %p\n", &type.e); //0xbfb3a8cc
printf("Start addr of float = %p\n", &type.f);     //0xbfb3a8d4
printf("Start addr of double = %p\n", &type.g);    //0xbfb3a8d8
printf("Start addr of *char  = %p\n", &type.h);    //0xbfb3a8dc

printf("Size of this structure is %zu\n", sizeof(type)); // 36
```

So gcc pads out the first char with an additional one, giving the structure a total size of 36, when 35 was expected.

6.3 Structures – that contain other structures

- Structures may also contain other structures. Assume that the following structures have been declared:

```
typedef struct dept {
   char     deptName[30];
   char     campus[14];
   int      phoneNumber;
} dept;

typedef struct address {
    char *pStreet;
    int   aptNum;
    char *pCityAndProv;
    char *pPostalCode;
} address;
```

We can use these structures inside other structures

6.3 Structures – that contain other structures

Then a student's record might be stored as:

```
typedef struct SR{
   long                studNumber;
   char               *lastName;
   char               *firstName;
   dept                department;
   address            *pHomeAddr;
} studentRecord;
```

We can then use this structure to initialize:

```
studentRecord BillSmith = {123456, "Smith",
  "Bill", {"ICT", "Woodroffe", 5551212},…};
```

Notice that the `SR` structure itself contains both a structure and a pointer to a strucuture. **`department`** is a **`dept`** structure variable, while **`pHomeAddr`** is *a pointer to* the **`address`** structure seen above.

6.3 Structures – that contain other structures

- To access a structure within a structure, use the dot notation just as you would in Java. For example, if we declare

```
studentRecord DebraJones;
```

Assuming this was correctly assigned proper values at the point of declaration, we can then refer to this student's department record name using:

```
DebraJones.department.deptName
```

Notice that since `deptName` is a fixed array of `chars`, once an address is assigned, the variable `DebraJones.department.deptName`, returns the address of the first character in the string that consists of the department name.

6.3 Structures – that contain other structures

- Notice that the LHS of the above statement is interpreted (from left to right) as:

```
(DebraJones.department).deptName
```

i.e. `DebraJones` is an `SR` structure that has a member (or property, if you prefer) called `department` of type `dept`. So the information in parenthesis resolves to a `dept` structure. In turn, `department` has a `deptName` member.

Note that the same rules about late assignments apply. Since `deptName` refers to a fixed array, you cannot assign a string after the point of declaration.

6.3 Structures – that contain other structures

- So how do you assign a string to

`DebraJones.department.deptName`

after it has been declared? As in an earlier example, use `strcpy()` to load anything into that array after the initial declaration. For example:

```
#include <string.h>
#include <stdio.h>
// dept, address, and SR structure declarations go here

int main(void){
    studentRecord DebraJones;
    strcpy(DebraJones.department.deptName, "ICT");
    printf("%s\n", DebraJones.department.deptName);
}
```

6.3 Structures – that contain other structures

- The final member of the `SR` structure is a pointer to a structure, called `pAddr`, where, as before:

```
typdef struct SR {
   long                  studNumber;
   char                 *lastName;
   char                 *firstName;
   dept                  department;
   address              *pHomeAddr;
} studentRecord;
```

and `address` was declared as:

```
typedef struct address {
   char *pStreet;
   int   aptNum;
   char *pCityAndProv;
   char *pPostalCode;
} address;
```

6.3 Structures – that contain other structures

- To access the members of the structure pointed to by `pAddr`, you would follow the same recipe as before:

```
DebraJones.(*pHomeAddr).aptNum = 5;
```

Caution: watch out for segmentation fault (see Section 6.4)

As before, the construction,

```
(*pointer_to_a_structure).member_name
```

has an special format, which can be used here:

```
pointer_to_a_structure->member_name
```

6.3 Structures – that contain other structures

Note that

```
DebraJones.(*pHomeAddr).aptNum = 5;
```

can be written as:

```
DebraJones.pHomeAddr->aptNum = 5;
```

What about the other members of this structure that were pointers to chars?

```
struct address {
    char          *pStreet;
    int            aptNum;
    char          *pCityAndProv;
    char          *pPostalCode;
};
```

6.3 Structures

How do we assign these members? It's no different than the previous examples throughout the course. Recall that when you declare a pointer, like this

```
char *pStreet;
```

you need to assign an address to `pStr`. This can be done using the address of the first character of the string at the point of declaration, like this

```
pStreet = "Main St.";
```

Assigning a value to a pointer inside a structure is no different.

```
DebraJones.pAddr->pStreet = "Main St.";
```

However, this assumes you know what you want to pass at the time the structure is declared? What happens is you wish to store data after the structure has been declared to a string that is not a string literal at compile time…? (See Section 6.4)

6.3 Structures

- Why use a pointer to a structure rather than just use a structure? *Unlike arrays, pointers are not passed by reference (i.e. by pointer)*. When you pass a structure to a function, you pass a copy of the whole thing…

  For example, using the `employee` structure given previously, lets look at the number of bytes passed into a `studentRecord` structure:

```
size_t getSRSize(studentRecord); // forward declaration

int main(void){
   studentRecord DebraJones;
   printf("%d\n", getSRSize(DebraJones));   // 64 bytes
}

size_t getSRSize(studentRecord s){
   return(sizeof(s));              // size of entire record
}
```

Using a pointer to the same structure, the value returned would be 4 bytes.

6.3 Structures – segmentation faults

- A segmentation fault is a somewhat cryptic run-time error that occurs when your code attempts to access memory that has not been reserved for use.

  Structures containing pointers to other structures offer one common opportunity for segmentation faults. Consider the following two structures:

```
struct B {
   char b[500];
};

struct A {
   struct B *pB;
} myA = {0};
```

6.3 Structures – segmentation faults

In this declaration, B is declared first, followed by A. The order is important, since `struct A` contains a reference to `struct B`.

```
struct B {
   char b[500];
};

struct A {
   struct B *pB;
} myA = {0};
```

However, defining a variable `myA` and assigning 0's to its members ***does not mean*** that `struct B` is initialized as well; `myA` contains only a single pointer to `struct B`, and not the 500 chars in `struct B` itself. In other words, `*pB = NULL` following this declaration—and that's all.

6.3 Structures – segmentation faults

This becomes obvious if we ask for the size of both structures.

```
sizeof(struct A) == 4
sizeof(struct B) == 500
```

So while `struct A` contains a member that points to `struct B`, that's all it does; its size is the size of a pointer on a 32-bit architecture. Any attempt to reference the members of B via `myA.pB->b[0]` at this point will generate a *seg fault*.

6.3 Structures – segmentation faults

- By comparison, consider the following structure declaration

```
struct A {
    int * p;
    struct B {
        char b[500];
    }sB;
} myA;
```

In this case, `sizeof(struct A) == 504`, since defining `struct A` (i.e. when you declare `myA`) causes space to be allocated for *both* the pointer `*p` *and* the 500-character array `b`.

Question: How would you access an element of the array b?

6.4 Note: Passing an Array of Structures to a Function

- As mentioned, a single structure is passed to a function by value, not by reference. But if you pass an array of structures to a function, then, just as with any array of primitive types, a pointer is passed instead. So given an array of employees declared as

```
employees employeeAr[50];
```

we might use the following function prototype to pass this array of structures to the `doPayroll ()` function:

```
void doPayroll (employees []);
```

and then actually calling the function, we have:

```
doPayroll (employeeAr);
```

6.4 Note: Passing an Array of Structures to a Function

- Alternately, this function could be declared as

```
void doPayroll(employees* e);
```

In either case, what gets passed is a pointer to an array, not the array itself. This means *inside* the function, you'll use the −> notation, not the

```
e[].member
```

notation. To repeat a previous message: even though you may think you're passing an array to a function, you're really just passing a pointer.

Question: inside a function that takes a structure as an argument, how would you access a member of the i^{th} element?

6.3 Structures

- So, *given a choice*, should you use pointers to structures, or just structures? That depends on the application. Where structures are small, pass them by value; where they are large and computer systems slow, pass them by reference.

  Of course, when you pass an array of structures to a function, you *are*—*as with all arrays passed to a function*—only passing a pointer to the address of the first structure. Once again, pointer arithmetic applies. So if a structure contains, say, 44 bytes of data in different formats, and if a pointer to an array of such structures is incremented, e.g.

  ```
  pStructure++;
  ```

  then the pointer jumps 44 bytes forward to the next element in the array of structures. But `pStructure` itself only increases by 1 value: pointer arithmetic applies to arrays of structures, as it does to all arrays.

6.3 Structures

- To summarize the benefits of passing a pointer to a structure into a function, rather that the structure itself:

1. Passing a pointer to a structure as an argument of a function is computationally less time-consuming; you pass four bytes, not 4000.
2. You can alter the original data without the necessity of having to return an `employee` structure. Therefore, your function can return a `void` rather than a structure—which is also time-consuming, since the whole thing must be passed a second time—and still get the job done by using the address directly. And,
3. If you only need to change one `char` in a structure, it doesn't make much sense to transfer the *entire* structure. Passing a pointer as an argument is much more efficient. And finally,
4. An array of structures is automatically converted to a pointer when passed into a function: you have no choice but to use pointers inside a function anyway.

6.3 Structures – segmentation faults

- Consider again our previous 'nested' structure UDT

```
typdef struct SR {
   long                 studNumber;
   char                *lastName;
   char                *firstName;
   struct dept          department;
   struct address      *pAddr;
} studentRecord;
```

where `address` was declared as:

```
struct address {
   char *pStreet;
   int   aptNum;
   char *pCityAndProv;
   char *pPostalCode;
};
```

6.3 Structures – segmentation faults

If we attempt to create

```
studentRecord DebraJones;
```

and set

```
DebraJones.pAddr->aptNum = 5;
```

then, while this will not generate any compile time errors, it *will* generate a segmentation fault at run time. `DebraJones` may be declared and have space allocated, but that does not extend to anything pointed to by any of 'her' members. Attempting to set the member value of a structure being pointed to, when *that* structure itself has not been properly initialized, will cause your program to attempt to access space not reserved for it, and generate segmentation fault errors during execution.

- To overcome this problem, we need to be able to reserve space for an `address` structure 'on the fly', during execution…

6.4 Dynamic memory allocation

- C supports three general forms of memory storage allocation; you've already seen two:
  - ***Automatic*** – memory reserved whenever a variable is declared, i.e. (implicitly) using the `auto` storage class. This happens whenever program flow enters a block of code and a non-static variable is delcared: memory is allocated automatically when the block is entered and deallocated when it is exited. This happens at run time.
  - ***Static*** – created using the `static` storage type, i.e. as with automatic, it is created inside a function, but preserved when the function exits. Static-allocated memory persists for the lifetime of the program, whether it is still needed or not—you can't free up static memory once it is created.

Note: 'static' has slightly different meanings in C++ and Java. In these OOP languages, static properties and methods are not instantiated separately, but act as a single shared resource across different instances of a class.

6.4 Dynamic memory allocation

- Additionally, C supports *dynamic memory allocation*. This type of memory allocation occurs 'on the fly' within the program when space for arrays and structures, whose size is unknown at compile time, needs to be allocated.

  Prior to the ANSI C99 standard, variable length automatic arrays were not allowed at compile time, and the only way to allocate an array of unknown size was to do so dynamically, using calls to three functions: `malloc`, `calloc`, and `realloc`.

6.4 Dynamic memory allocation

- Unlike automatic and static storage, dynamic memory gives the programmer complete control over when to allocate and deallocate memory. Unlike higher-level languages which have built-in garbage collection, the programmer is entirely responsible for freeing up memory after is use is finished (using the `free()` function). Failure to do so results in a *memory leak*, which eventually consumes all available RAM and usually triggers an error from the OS.

  So in addition to freeing up memory when it is no longer needed, the programmer also needs to test for the possibility that there is no memory available.

6.4 Dynamic memory allocation

- When a pointer is declared, whether inside a structure or not, if it is *not* set to the starting address of a string literal or predefined structure/data type, then it *must* point to a place in memory that is reserved for it (via the OS) to be of use, otherwise a segmentation fault will result. This is what dynamic memory allocation allows you to do: reserve space in memory to store information after he point of declaration.

6.4 Dynamic memory allocation – `malloc()`

- The four functions dynamic memory functions are located in `<stdlib.h>` and are used entirely for dynamic memory allocation (the '`_alloc`'s) and deallocation ('`free`').

  `malloc``()` — takes as a single argument the number of bytes to reserve, and returns a void pointer to the start of the reserved location in memory. If the call to `malloc` was unsuccessful, a `NULL` is returned. The official declaration for `malloc` is:

```
void *malloc(size_t size);
```

  Recall that `size_t` is an `unsigned int` that stores the maximum address size available on a system (it is `typedef`ed in `stddef.h`).

  For example, to allocate space for 5000 integers, use:

```
int *mem, n = 5000;
mem = malloc(n * sizeof(int));
```

6.4 Dynamic memory allocation – `calloc()`

- **`calloc`**`()` — takes two arguments—the number of data types required and the size of each type—and returns a void pointer to the start of the reserved location in memory. Unlike `malloc`, `calloc` (which stands for 'contiguous allocation') initializes the memory space to 0. If the call to `calloc` was unsuccessful, a `NULL` is returned.

```
void *calloc(size_t n, size_t element_size);
```

Therefore, on 32-bit architectures, `sizeof()` returns the correct type to the `element_size` argument. This *should* work on 64-bit architectures as well, *provided* `sizeof()` returns 8-byte `unsigned int`s.

As an example of `calloc()` (having an effect identical to `malloc`);

```
int *mem, n=5000;
mem = calloc(n, sizeof(int));
```

Or, as a 'one-liner': `int n=5000, *mem=calloc(n, sizeof(int));`

6.4 Dynamic memory allocation – `realloc()`

- **`realloc`**`()` — changes the size of a block of memory. It takes two arguments—a pointer to a currently allocated block of memory, along with the new `size` of bytes required. If the call to `realloc` is unsuccessful, a `NULL` is returned.

```
void *realloc(void *ptr, size_t size);
```

If the new space is larger than the old space, the contents will be unchanged. The new space is *not* initialized. If the new space is smaller than the old space, bytes will be truncated off the end of the space.

6.4 Dynamic memory allocation – `realloc()`

- Generally, `realloc` attempts to keep the new base address of the memory the same as the old, but it may be necessary to allocate a fresh block of memory (deallocating the old memory). So when using `realloc`, always refresh the pointer address, since it may have changed, and you're no longer pointing to the new location that holds the extra data.

  As with `malloc` and `calloc`, a `NULL` pointer indicates you were not successful, however the memory is not disturbed, and the old pointer can still be used.

6.4 Dynamic memory allocation – `free()`

- **`free`**`()` — deallocates the memory by passing a pointer to the base address of a previously allocated space in memory. If the pointer passed via `free()` is a `NULL` then the request is ignored.

```
void free(void *ptr);
```

Only pointers to space previously allocated by `malloc`, `calloc`, or `realloc`, and not already deallocated by `free()` are permitted. Any other attempt to use `free()` on a `void*` results in an error being triggered.

6.4 Dynamic memory allocation - `alloca()`

- **`alloca()`** is a non-ANSI function for allocating memory. Like `malloc`, `alloca()` is a C function that returns a pointer to free memory based on the byte size requested. The format is:

```
void *alloca(size_t size);
```

- Unlike `malloc()`, `alloca()` allocates memory off the stack, thus enjoying the following benefits:
  - Although not part of the ANSI C standard, it is a standard within the GNU C library (and is recognized by GCC)
  - It sits on the stack and so provides faster access than heap memory
  - When you exit a function, the stack is popped, along with the `alloca()` memory; no need to use `free()`, and no possibility of memory leaks.
  - Repeated `malloc()` calls can fragment memory. But since `alloca()` uses the stack only, memory can't be fragmented. The worst you can do is generate a stack overflow.

6.4 Dynamic memory allocation - `alloca()`

- However, `alloca()` has disadvantages as well:
  - It is non-standard, and thus not universally portable to other systems
  - It must be used with some caution, since it can mess up the stack if used inappropriately, e.g. you should not make a global pointer to `alloca()`-allocated memory
  - It is temporary memory only, hence acts like 'local', not 'global' memory.
  - Declaring

```
char mem[1000];
```

inside a function gets you the same thing as

```
char* mem = alloca(1000);
```

So why bother with `alloca()`?

For a spirited discussion on the tradeoffs involved using `alloca()`, see:

http://stackoverflow.com/questions/1018853/why-is-alloca-not-considered-good-practice

6.4 Dynamic memory allocation – using `void*`

- As a general rule, pointers should point to *something*; a pointer to nothing is dangerous because the compiler needs to know how big the data type is that it is pointing to. Memory allocation is the one exception, since otherwise you'd need a separate `_alloc` function for each data type—but then what about allocating memory for structures of unknown size?

- In C++, this problem is resolved using two built-in functions, `new()` (in place of the `_allocs`— which can be used to reserve heap space for objects)—and `delete()`, which replaces `free()`.

- In Java, you go one step further: as with C++, `new()` replaces the `_allocs`, but garbage collection replaces `free()` / `delete`() altogether. Note that garbage collection makes your life easier, but it also adds overhead. Java needs to track allocation and count references (via *data structures*!), all of which slow performance and add complexity to the language, while occupying more space in memory.

6.4 Dynamic memory allocation – with structures

- Dynamic allocation resolves the problem of passing data to a location in memory assigned *after* the point of declaration, as the following code demonstrates. Again, assuming:

```
typedef struct address {
 char *pStreet;
 int   aptNum;
 char *pCityAndProv;
 char *pPostalCode;
} address;
```

```
typedef struct SR{
   long studNumber;
   char *lastName;
   char *firstName;
   dept department;
   address *pHomeAddr;
} studentRecord;
```

6.4 Dynamic memory allocation – with structures

- If we declare

```
studentRecord DebraJones = {0};
```

Then we can access the members of this `struct`, provided we set the pointer address equal to the address of a defined space that holds an actual `address` structure, i.e. we must `malloc()` a space for an address structure.

6.4 Dynamic memory allocation – with structures

```
char street[30];  // temporary array to hold streetname

// return pointer to address structure
address *pDJaddr = (address*)malloc(sizeof(address));

// set address in pDJaddr into pHomeAddr
DebraJones.pHomeAddr = pDJaddr;
DebraJones.pHomeAddr->pStreet = (char *)malloc(30);

printf("%s","Enter Street name\n");
scanf("%s", street);
strcpy(DebraJones.pHomeAddr->pStreet, street);

printf("Debra Jones lives on %s.\n",
   DebraJones.pHomeAddr->pStreet);
```

6.4 Dynamic memory allocation – with structures

- In this code, we've used two data types to hold temporary values: `DJaddr`, which is a pointer to an `address`, `street`, which is a pointer to a char. We can simplify our code still further by bypassing these two temporary types.

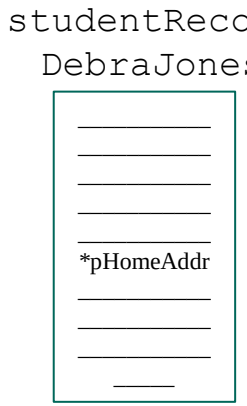
```
studentRecord DebraJones = {0};

DebraJones.pHomeAddr = (address*)malloc(sizeof(address));

DebraJones.pHomeAddr->pStreet =
                        (char *)malloc(30*sizeof(char));

printf("%s", "Enter Street name\n");
scanf("%s", DebraJones.pHomeAddr->pStreet);

printf("Debra Jones lives on %s.\n",
                         DebraJones.pHomeAddr->pStreet);
```

6.4 Dynamic memory allocation – with structures

- It's worth looking at the following three lines in some detail

```
studentRecord DebraJones = {0};

DebraJones.pHomeAddr = (address*)malloc(sizeof(address));

DebraJones.pHomeAddr->pStreet =
                        (char *)malloc(30*sizeof(char));
```

The first line allocates 64 bytes of space for the `DebraJones` `studentRecord`. This includes 48 bytes of space for the `dept` structure, and 16 bytes for the remaining three pointers and type `long` student number.

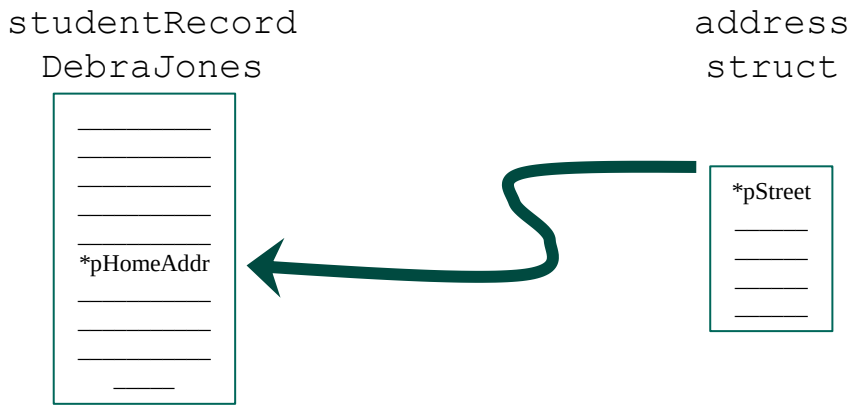


6.4 Dynamic memory allocation – with structures

- The second line

```
DebraJones.pHomeAddr = (address*)malloc(sizeof(address));
```

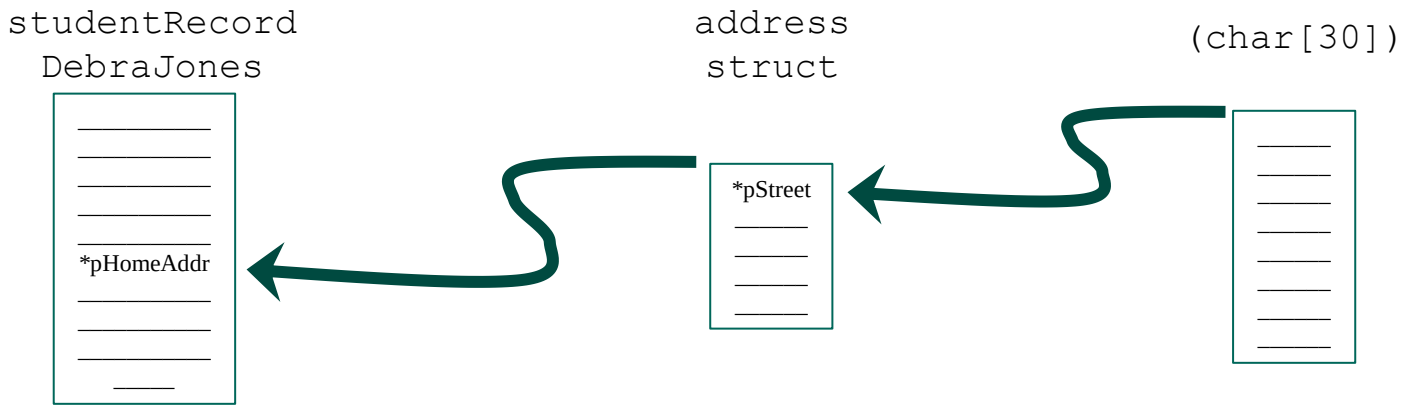
does two things. First, it allocates 16 bytes to store `address` structure information, and then returns the value of the starting address at that location in memory to `pHomeAddr`.



6.4 Dynamic memory allocation – with structures

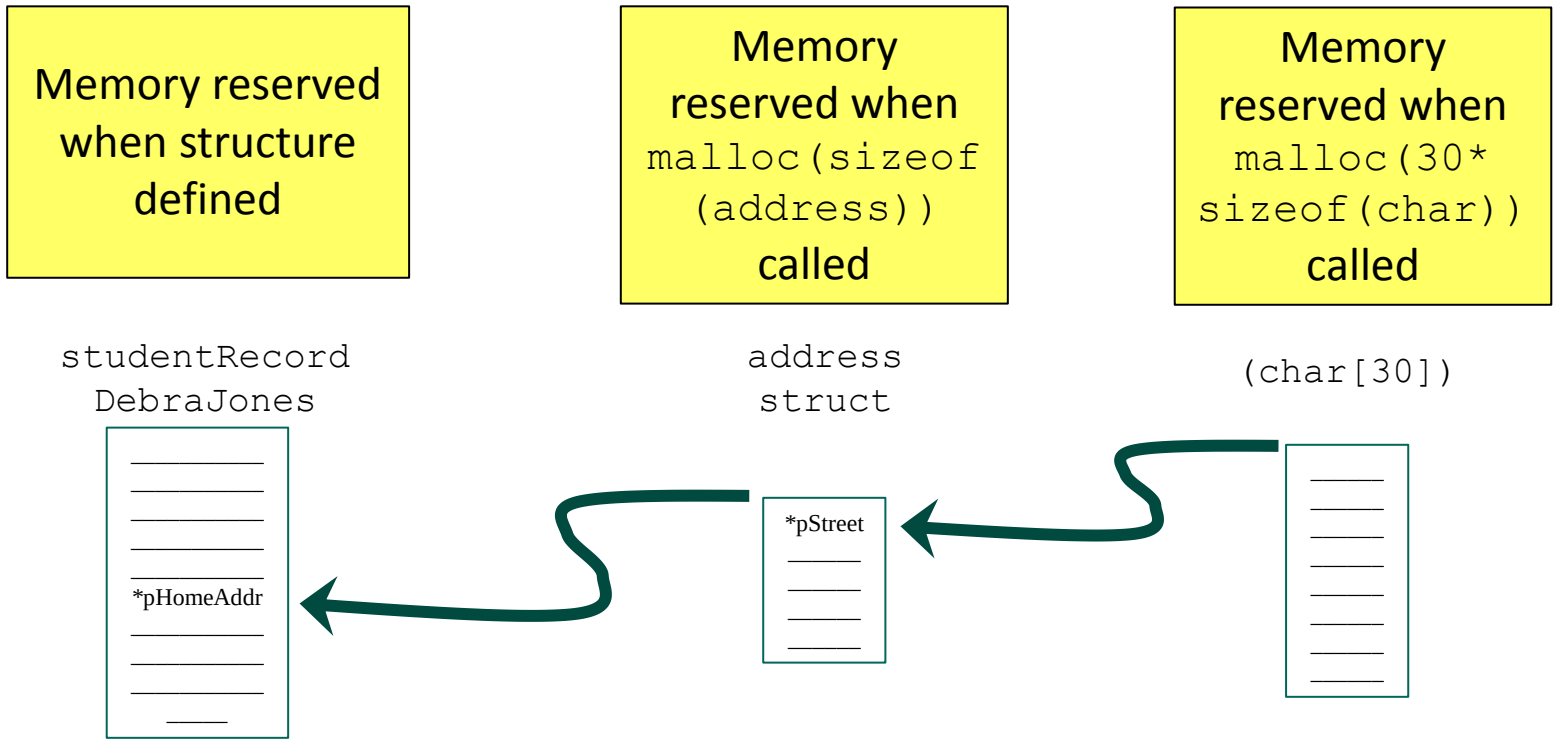
- The third line does roughly the same things as the second line, but on an array of `chars` instead of an `address` structure. This time, the address of the start of the array of `chars` is returned, this time to the location in memory corresponding to the `pStreet` pointer.

```
DebraJones.pHomeAddr->pStreet =
                        (char *)malloc(30*sizeof(char));
```



6.4 Dynamic memory allocation – with structures

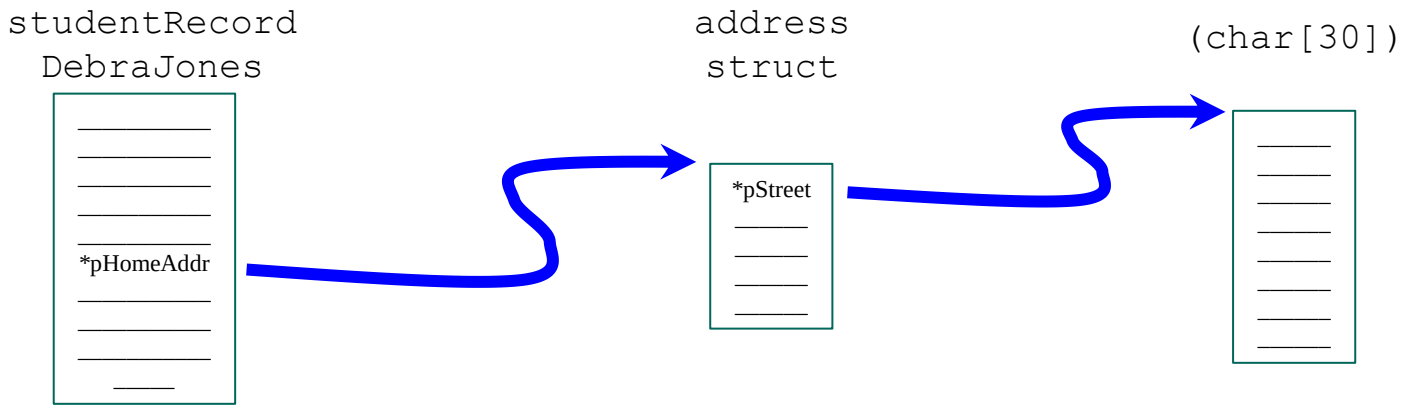
Note that, while memory for the `DebraJones` structure was created by the declaration of the `studentRecord` structure (the first of the three lines above), both the `address` and character arrays—*the things being pointed to*—needed to be `malloc`ed into existence.



6.4 Dynamic memory allocation – with structures

- The result of all this activity is that when the time comes to store the characters that make up the street name entered by the user, `pStreet` stores the address of the array of 30 characters, and the address of `pStreet` itself is returned via `DebraJones.pHomeAddress`. In the end, what `scanf()` sees is just what it needs: the address of an array of `char`s, suitable for storing a string of up to 30 characters.

```
scanf("%s", DebraJones.pHomeAddr->pStreet);
```



6.5 Unions

- A union is a UDT that allows a set of data types (including pointers, enumeration types and structures) to share the same location in memory. Think of a union as a way of overlaying different data on top of one another at the same place in memory. For example:

```
union mytypes {
    char      c[4];
    short     s[2];
    int       i;
} data;
```

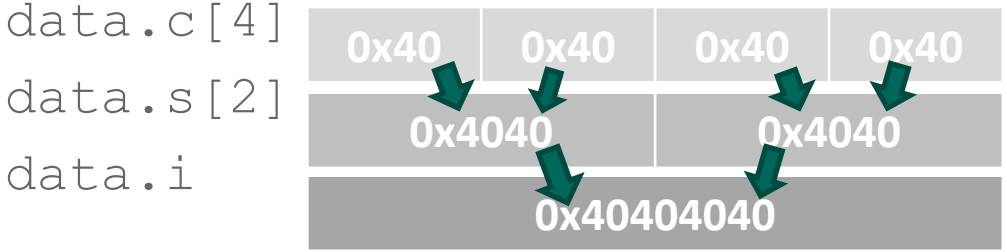
`data` can refer to a `char`, `int` or `double`, all occupying the same space in memory

6.5 Unions

- To refer to an element of a union (or structure), use the dot notation. For example, given the above declaration we can store some bytes into the character array `c[4]` which overlaps the other values in the union:

```
for (int j = 0; j < 4; j++)
   data.c[j] = '@';  // '@' = 0x40 in ASCII
printf("data.i = %x in hex ", data.i);
printf("and = %d in dec\n", data.i);
```

Our union declaration has the effect of ensuring that anything we do to the array of 4 chars will also change `data.s` and `data.i` as well:



6.5 Unions

- The output is:

```
data.i = 40404040 in hex and = 1077952756 in dec
```

- Note that unions allow for a (potentially dangerous) form of ***overloading*** (see the following slides for examples)

6.5 Unions

- When the union contains member elements of different sizes, the compiler reserves space for the largest element, and aligns the elements to the starting (i.e. lowest) address. For example, if we define:

```
union mytypes {
    char      c;
    short     s;
    int       i;
} data;
```

The size of the space reserved is 4 bytes—the size of an `int`, the largest member—and `c`, `s` and `i` are all left-aligned to the starting address of `i`. So:

```
sizeof(data) == sizeof(data.i) == 4;
sizof(data.s) == 2;  sizeof(data.c) = 1;
```

6.5 Unions—Typical Uses

- Unions are useful when applications require that a given space be used for different interpretations. These uses fall into two general categories:

(1) Unions can be used to conserve space. For example, when two or more mutually exclusive types of data can be used to store information, unions allow the programmer to record only the most relevant. Consider the following code for employee ID information. Only one piece of information needs to be stored to locate the employee. In a large database, this could save valuable resources:

```
typedef struct {
   union {
      unsigned int     employeeIDNum;
      unsigned long    SINum;
      char             payrollNum[10];
   };
   enum IDtype {ID, SIN, PRNum} ID;
} employeeID;
```

Overlaps employee ID information

Type of ID stored in union

6.5 Unions—Typical Uses

(2) Unions can be used to overlay one data type on top of another, allowing for alternate interpretations. One common example is:

```
union DW32{
   unsigned int      DWORD;
   unsigned short    WORD[2];
   unsigned char     BYTE[4];
};
```

We can then refer to the words and bytes of DWORD according to their natural boundaries. Using structures, we can even extend this to smaller components still, like nibbles and bits.

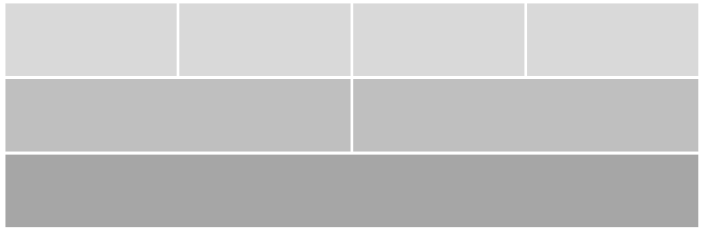
6.5 Unions—Inherent Dangers

- Unions lead to weak typing and so are rarely used, except in the above two cases
- The declaration for unions is identical to the declaration for structures, and this can lead to confusion. Whereas structures contain all their member elements separately, union elements are all zero-based at the same location—they coexist on top of one another. Be sure you understand which of the UDTs you're using whenever you come to a UDT declaration
- Some authors suggest that rather than using unions to convert between different data types, you should use bit masking instead. This would seem to be the minority position, and the overlapping of bits, bytes, and words is commonplace in C/C++ programming.

6.5 Unions - Inherent Dangers - *Endianness*

- The greatest danger in using unions comes from the problem of endians. As an example, consider the following code, taken from an earlier example:

```
union mytypes {
   char     c[4];
   short    s[2];
   int      i;
} data;
```



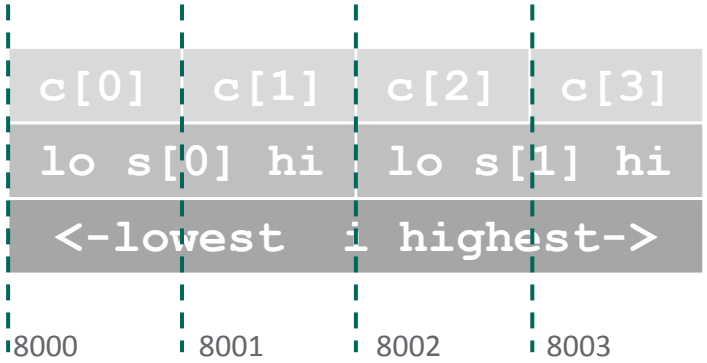
If we check the addresses of the various components of `mytypes` we find the following:

```
&c[0] = 8000   &c[1] = 8001
&c[2] = 8002   &c[3] = 8003

&s[0] = 8000   &s[1] = 8002

&i    = 8000
```

6.5 Unions - Inherent Dangers - *Endianness*

- The actual format is represented below. The memory is stored in little endian format, with the lowest valued byte of information at the lowest address

```
union mytypes {
    char      c[4];
    short     s[2];
    int       i;
} data;
```

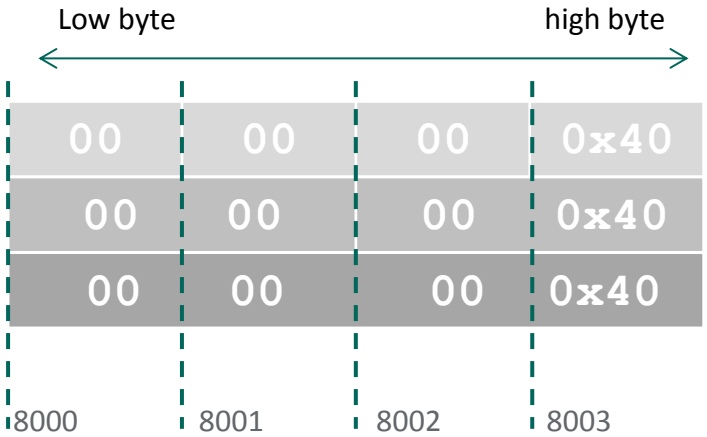


6.5 Unions - Inherent Dangers - *Endianness*

optional

So if the value stored in the character array is 00 00 00 40 (all in hex), then the bytes need to be read backward to obtain the integer value: 0x40000000

```
union mytypes {
    char      c[4];
    short     s[2];
    int       i;
} data;
```

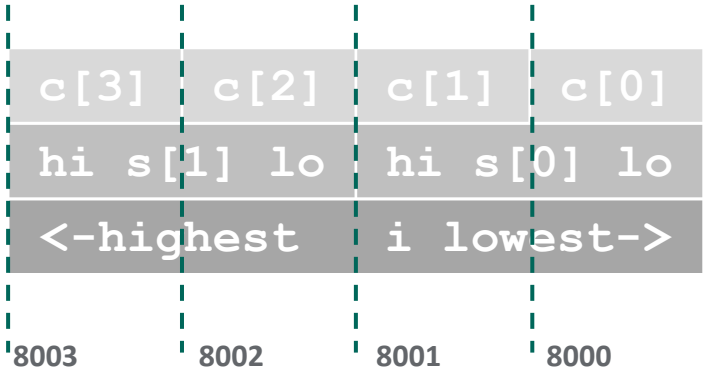


6.5 Unions - Inherent Dangers - *Endianness*

optional

- An alternate interpretation is that our model of memory addressing is wrong; that we should count *down* in memory (reading from left to right). This interpretation also works:

```
union mytypes {
    char      c[4];
    short     s[2];
    int       i;
} data;
```



So if the value stored in the character array is 0x40 00 00 00, then the bytes can be read forward (from left to right) to obtain the integer value: 0x40000000 (= 1073741824). So for little endian storage, we actually count down in address space, not up, as you usually expect.

6.5 Unions - Inherent Dangers - *Endianness*

- Which way is correct? There isn't a right or wrong answer to that question: it depends on whether the designers of the system elected to use big endian or little endian formatting. Note that the problem arises because of an inherent contradiction in the way we express information: the number line is written so that addresses generally increase from left to right; but numbers are written with the greatest values on the left, and the smallest on the right. This leads to an apparent inconsistency in how we actually store information.

- The following code is suggested as one possible test for endianness*:

```
int isLittleEndian() {
    int i = 1;
    char *low = (char*) &i;
    return ((*low )? 0 : 1);
}
```

*Taken, with modifications, from Mark Loiseau's website, at http://blog.markloiseau.com/2012/05/find-endianness-in-c/

6.5 Unions - Inherent Dangers - *Endianness*

- Endianness is always present, but totally invisible to most programmers; unless you need to work at the level of bits and bytes, *or with unions*, you may never hit this problem in a lifetime of programming. The list at right gives some common file formats and their endianness.
- The endian problem is not limited to software; it pervades hardware as well. See the list at right for the endianness of different hardware platforms.

Big Endian	Little Endian
Photoshop	BMP
JPEG	GIF
MacPaint	QuickTime
PNG	

Big Endian	Little Endian
Motorola	Intel
SPARC	VAX
PowerPC	DEC Alpha

"This is an attempt to stop a war. I hope it is not too late and that somehow, magically perhaps, peace will prevail again." - Danny Cohen, 1980, writing on the Endian Wars

With information taken from various sites, especially http://people.cs.umass.edu/~verts/cs32/endian.html. The quote at right, and the hardware table, are from http://teaching.idallen.com/cst8281/10w/notes/110_byte_order_endian.html

6.6 Bitfields

- Structures and unions have a curious feature: you can declare the width of a `field` of individual bits inside `int`s and `unsigned` members, and give names to each field. The format is:

```
struct tagname {
   int          a : fieldsize1, b : fieldsize2,…;
   unsigned     c : fieldsize3, d : fieldsize4,…;
};
```

where `a`, `b`, `c`, and `d` are fieldnames, and `fieldsizeX` is the width of bits.

(Yes, endianness comes into play here. Whether the compiler assigns bitfields left-to-right or right-to-left is machine dependent.)

6.6 Bitfields

- For example, we can use the following structs:

```
typedef struct {
   unsigned N0 : 4, N1 : 4, N2 : 4, N3 : 4,
            N4 : 4, N5 : 4, N6 : 4, N7 : 4;
} nibbles;

typedef struct {
  unsigned B0:1, B2:1, B3:1, B4:1, B5:1, B6:1,
  B7:1, B8:1, B9:1, B10:1, B11:1, B12:1, B13:1,
  B14:1, B15:1, B16:1, B17…B30:1, B31:1, B32:1;
} bits;
```

6.6 Bitfields

…to modify our earlier union code example to:

```
union DW32{
    unsigned int        DWORD;
    unsigned short      WORD[2];
    unsigned char       BYTE[4];
    nibbles             NIBBLE;
    bits                BIT;
} myWord = {0};
```

We can use this as follows:

```
    myWord.BIT.B4 = 1;          // …10000b
    printf(%d, myWord);         // = 16
    myWord.NIBBLES.N0 = 15;     // ANDs in …01111b
    printf(%d, myWord);         // = 31
```

6.6 Bitfields

- Bitfields also apply to unions. One common example involves using bitfields to pull the different colours out of a 24-bit color value. Assume the bottom 24 bits of an `int` are used to store three bytes of color information. Each byte stands for a red, green, or blue value:

```
union {
 unsigned color24;
 struct {
      unsigned RED:8, GREEN:8, BLUE:8;
 }RGB;
} colour;


colour.RGB.GREEN = 0xFF; // dark green
```

6.8 Miscellaneous Notes on UDTs

- Note that, since an enumeration type is always a constant integer, enumeration types are always primitive data types; by default, they are naturally passed by value, never by reference. Of course, nothing prevents you from using a pointer to an enumeration type in an expression, declaration, array, function header, etc.
- Similarly, a `union` is a single piece of data (even though it consists of overlapping `int`s, `float`s, etc.) so it is passed directly, rather than as a reference data type
- Structures are (potentially) 'very complicated primitive types'—an obvious contradiction. As such, they occupy a middle ground between primitive data types and referenced data types (like arrays). The decision was made to pass structures by value, but decision potentially compromises speed for simplicity of use.

6.8 Miscellaneous Notes on UDTs

- `enum`s exist in Java, but `union`s do not—that feature was considered too low-level for Java. `union`s also rely very heavily on the programmer's understanding the underlying hardware, and Java is supposed to be agnostic about what hardware it runs on. On the other hand, structures are not needed in Java, since a `class` does pretty much everything you'd want a `struct` to do. So in the end, of the three UDTs found in C, only `enum`s exist in Java.

- While tag names may be omitted, it is generally considered good programming practice to supply them. Alternately, use `typedef` to create an alias to a `struct`, `union`, or `enum`.

6.8 Miscellaneous Notes on UDTs

- UDTs should generally be put in their own header file. Again, this is not essential, but it is considered good programming practice.
- In enumerated types, while you can set the type, you cannot print out the enumerator name itself e.g.

```
printf("%s", card[0].ACE_OF_HEARTS);
```

(after all, its just a number, not a string.) To use a string version of the variable name, create a structure that holds both the enumeration type *and* a pointer to an array of characters that holds the name of the enumerator.

- Similarly, unions need to be stored with information that indicates 'what type of union is this?', otherwise the programmer has to keep track externally.

6.8 Miscellaneous Notes on UDTs

- Note that the usual rules about global and local declarations apply. If we declare:

```
struct card{
 int pips;  // 1 = ace, 11 = Jack, 12 = Queen…
 char suit; // clubs, spades, hearts, diamonds
} c1;       // then c1 is global to the file

int main(void){
   struct card c2; // c2 is local to main()
   ...
}
```

…the same thing goes for `enum`s and `union`s.

Review of Pointers and Arrays

- If we create a structure

```
typedef struct point{
    int x, y;
} point;
```

then we can create both arrays of the `point` type and pointers to this type as well:

```
point Points[] = {1,2}, *pPoints;
```

and then set

```
pPoints = Points;
```

Again, `Points` returns the address of the first element of the array of `Points[]`.

Review of Pointers and Arrays

- Note that in the example above `Points[]` only contains a single element: one `point` structure containing the values:

```
Points[0].x == 1
Points[0].y == 2
```

To initialize an even larger array of, say, three `points`, use

```
point Points[] = {{1,2},{3,4},{5,6}}, *pPoints;
```

If we set

```
pPoints = Points;
```

then `Points` again returns the address of the first element of the array of `Points[]`. But now `pPoints` points to an array of three elements, and we can start to use pointer arithmetic.

Review of Pointers and Arrays

- As before, when we used primitive data types for our arrays, we are limited in what we can do *after* a declaration is made:

```
point Points[1], *pPoints;

Points[] = {1,2};   //WRONG

*pPoints = {1,2};   //WRONG
```

Review of Pointers and Arrays

- Note that when arrays and strings are members of a structure, the same rules apply:

```
typedef struct data{
   char a[3];
   char *b;
} data;
```

Given

```
data Ar[4], *pAr;
```

then an attempt to directly assign a value to the array is forbidden:

```
Ar[0].a = "ab";   // WRONG
```

However, you can assign the pointer value to the string directly (using member access operator, ->):

```
pAr->b = "ab";    // CORRECT
```

Review of Pointers and Arrays

- The –> notation is important here. Since, in the declaration

```
data Ar, *pAr = &Ar;
```

`pAr` is a pointer then

```
pAr.b = "ab";       // WRONG
```

You *must* use either

```
pAr->b = "ab";    // CORRECT
```

or

```
(*pAr).b = "ab";    //CORRECT
```

Crib sheet: Summary of Structure/Member Assignments

Given:

```
struct S {
   int i, arI[20], *pI;
   char c, arC[20], *pC;
}  s, arS[10], *pS;
```

A structure must be assigned space before its members can be accessed:

- `s, arS[10]` ensure that space is assigned automatically; you can assign values at the point of declaration using the `{}` notation, or use the 'dot' notation item-by-item afterwards;
- `*pS` just declares a pointer, not a structure. Use `pS=&s`, `pS=arS`, or `pS = (S*)malloc(n*sizeof(S));` to assign a useable address to the pointer `pS`.

Once a structure is defined, its individual members may be assigned values:

- `i`, `arI[20]`, `c` and `arC[20]` have space allocated during the initial structure declaration;
- Late assignments to `arI[20]` and `arC[20]` are prohibited; use `memcpy()` and `strcpy()` (respectively) to move information pointed to at one address into another location;
- `*pI` and `*pC` initially exist as pointers only; to use them, you must allocate space using either `.pI=&I`, `.pI=arI`, or `.pI=(int*)malloc(n*sizeof(int))`, (and use the equivalent expressions for `.pC`). With string literals only, you can use `.pC="abc…"` at any point;
- For structures within structures , use the dot notation to access individual members;
- For pointers to structures (e.g. `struct T *pT`) used as members within structures, use `.pT=&T`, `.pT=arT`, or `.pT=(T*)malloc(n*sizeof(T));` --the same as above for `*pS`.