

STRUCT, TYPEDEF, ENUM & UNION

In this session we will learn struct and union types,
typedef and enum

STRUCT, TYPEDEF, ENUM & UNION

Structure (`struct`)

- With array, we can only declare one data type per array.
- For different data type, we need another array declaration.
- It is single type aggregate data type.
- `Struct` overcomes this problem by declaring composite data types which can consist different types.
- A structure is a collection of related data items stored in one place and can be referenced by more than one names.
- These data items are different basic data types. So, the number of bytes required to store them may also vary.
- A structure type is a user-defined composite type.
- It is composed of fields or members which can be different types.

STRUCT, TYPEDEF, ENUM & UNION

- In C++, a `struct` is same as a `class` except that its members are `public` by default.
- In order to use a structure, we must first declare a structure template.
- The variables in a structure are called elements or members.
- In C, you must explicitly use the `struct` keyword to declare a structure however in C++, this is unnecessary, once the type has been defined.
- In C99, the allowable data types for a bit field include qualified and unqualified `_Bool`, `signed int`, and `unsigned int`.
- The default integer type for a bit field is `signed`.
- You have the option of declaring variables when the structure type is defined by placing one or more comma-separated variable names between the closing brace and the semicolon.

STRUCT, TYPEDEF, ENUM & UNION

- Structure variables can be initialized. The initialization for each variable must be enclosed in braces.
- Both structure types and variables follow the same scope as normal variables, as do *all identifiers*.
- If you define a structure within a function, then you can only use it within that function.
- Likewise if you define a structure outside of any function then it can be used in any place throughout your program.
- Example, to store and process a student's record with the elements chIdNum (identification number), chName, chGender and nAge, we can declare the following structure,

```
struct student {  
    char chIdNum[5];  
    char chName[10];  
    char chGender;  
    int nAge;  
};
```



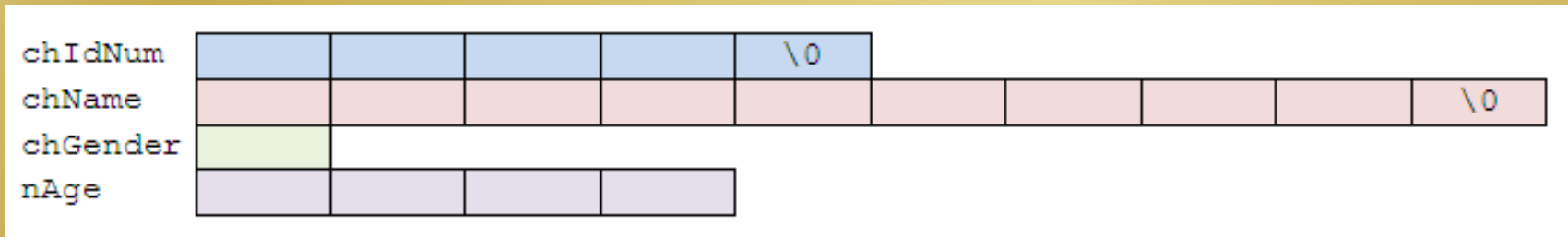
tag

STRUCT, TYPEDEF, ENUM & UNION

- Here, `struct` is a keyword that tells the compiler that a structure template is being declared and `student` is a tag that identifies its data structure.
- Tag is not a variable; it is a label for the structure's template.
- Note that there is a semicolon after the closing curly brace.
- A structure tag simply a label for the structure's template but you name the structure tag using the same rules for naming variables.

STRUCT, TYPEDEF, ENUM & UNION

- The previous sample template for the structure can be illustrated as follow (note the different data sizes),



STRUCT, TYPEDEF, ENUM & UNION

Define, declare and initialize

- Compiler will not reserve memory for a structure until it is declared.
- A structure declaration names a type and specifies a sequence of variable values called 'members' or 'fields' of the structure that can have different types.
- An optional identifier, called a 'tag', gives the name of the structure type and can be used in subsequent references to the structure type.
- A variable of that structure type holds the entire sequence defined by that type.
- Declaring structure variables can be done in the following way,

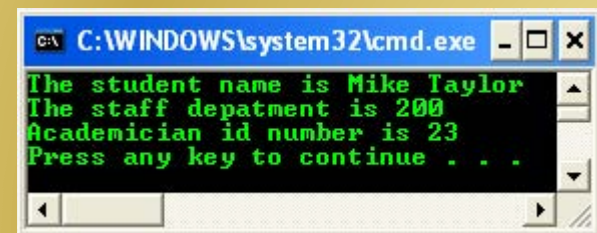
```
struct MyEmployee    // defines a structure variable named EmpData
{
    char chName[20];
    int nIdNum;
    long nDepatment;
} EmpData;
```

STRUCT, TYPEDEF, ENUM & UNION

- The struct `MyEmployee` structure has three members: `chName`, `nIdNum`, and `nDepartment`.
- The `chName` member is a 20-element array, and `nIdNum` and `nDepartment` are simple members with `int` and `long` types, respectively.
- The identifier `MyEmployee` is the structure identifier.
- Then we can declare variables of type `struct MyEmployee` like the following,

```
struct MyEmployee student, staff, academician;
```

[Struct basic example](#)



```
C:\WINDOWS\system32\cmd.exe
The student name is Mike Taylor
The staff department is 200
Academician id number is 23
Press any key to continue . . .
```


STRUCT, TYPEDEF, ENUM & UNION

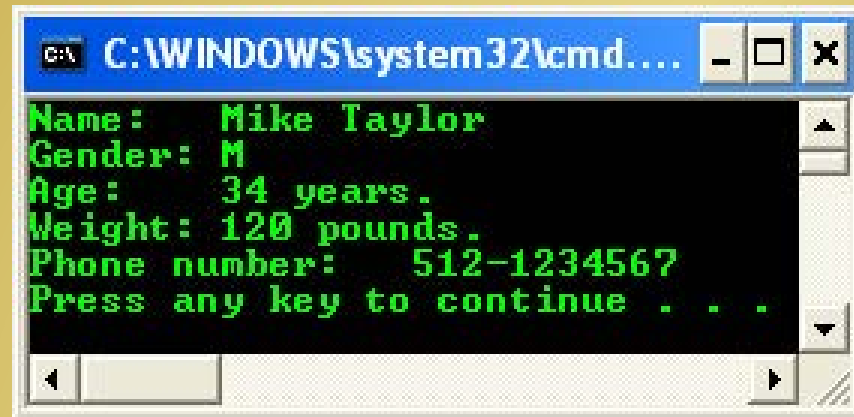
- Another example, anonymous struct (struct without tag),

```
struct /* defines an anonymous struct and a */  
{ /* structure variable named area */  
    float fwidth, flength;  
} area;
```

- The area structure has two members with float type, fwidth and flength.
- The structure type has no tag and is therefore is unnamed or anonymous.
- The nested structures, called anonymous structures allows you to declare a structure variable within another structure without giving it a name.
- However, C++ does not allow anonymous structures.
- You can access the members of an anonymous structure as if they were members in the containing structure.

STRUCT, TYPEDEF, ENUM & UNION

- Anonymous structure example



A screenshot of a Windows command prompt window. The title bar shows the path "C:\WINDOWS\system32\cmd....". The command prompt displays the following text in green on a black background:

```
Name:  Mike Taylor
Gender: M
Age:   34 years.
Weight: 120 pounds.
Phone number: 512-1234567
Press any key to continue . . .
```

STRUCT, TYPEDEF, ENUM & UNION

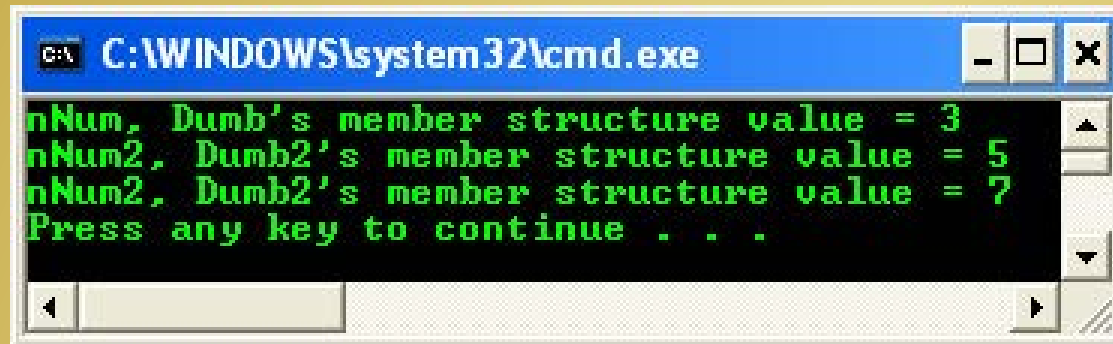
- Anonymous structures can be useful when the tag named is not needed.
- This is the case when one declaration defines all structure instances.
- Embedded or nested structures are often anonymous. For example,

```
struct MyCube{  
    struct    /* anonymous structure */  
    {  
        int width, length;  
    } area;  
    int height;  
} CubeData;
```

- Nested structures can also be accessed as though they were declared at the file-scope level.

STRUCT, TYPEDEF, ENUM & UNION

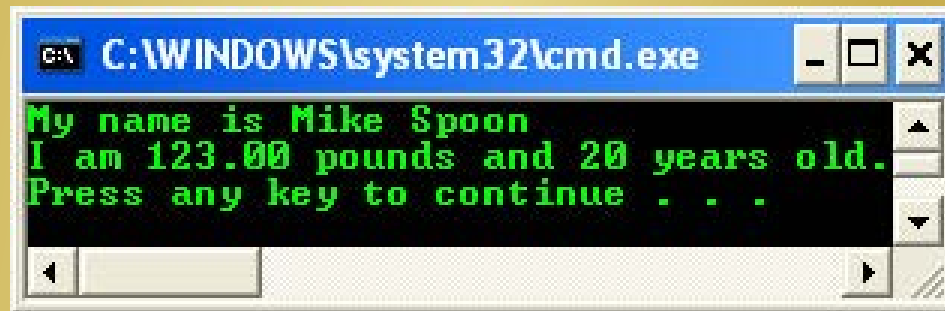
- Nested structure example



```
C:\WINDOWS\system32\cmd.exe
nNum, Dumb's member structure value = 3
nNum2, Dumb2's member structure value = 5
nNum2, Dumb2's member structure value = 7
Press any key to continue . . .
```

STRUCT, TYPEDEF, ENUM & UNION

- Optionally, we can declare variables when the structure type is defined by placing one or more comma-separated variable names between the closing brace and the semicolon.
- Structure variables can be initialized.
- The initialization for each variable must be enclosed in braces.
- [Define & declare structure example](#)



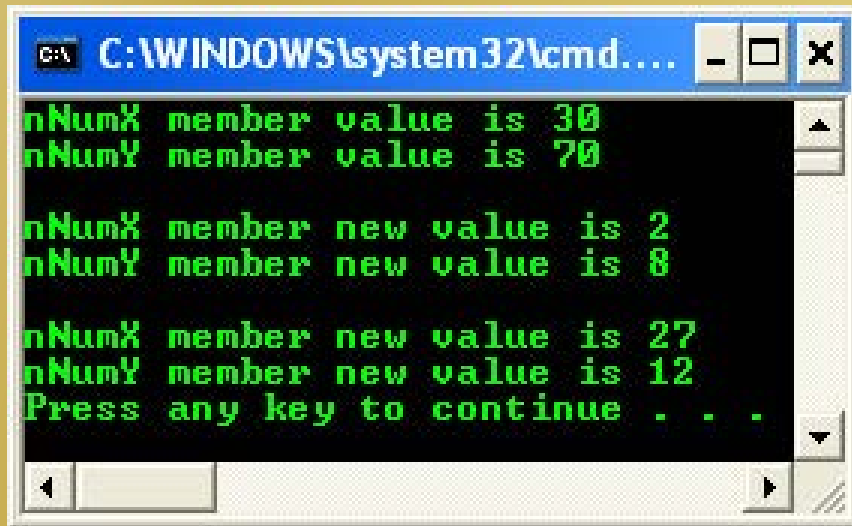
A screenshot of a Windows command prompt window. The title bar reads "C:\WINDOWS\system32\cmd.exe". The window contains the following text in green on a black background:

```
My name is Mike Spoon  
I am 123.00 pounds and 20 years old.  
Press any key to continue . . .
```

The text is displayed in a monospaced font. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

STRUCT, TYPEDEF, ENUM & UNION

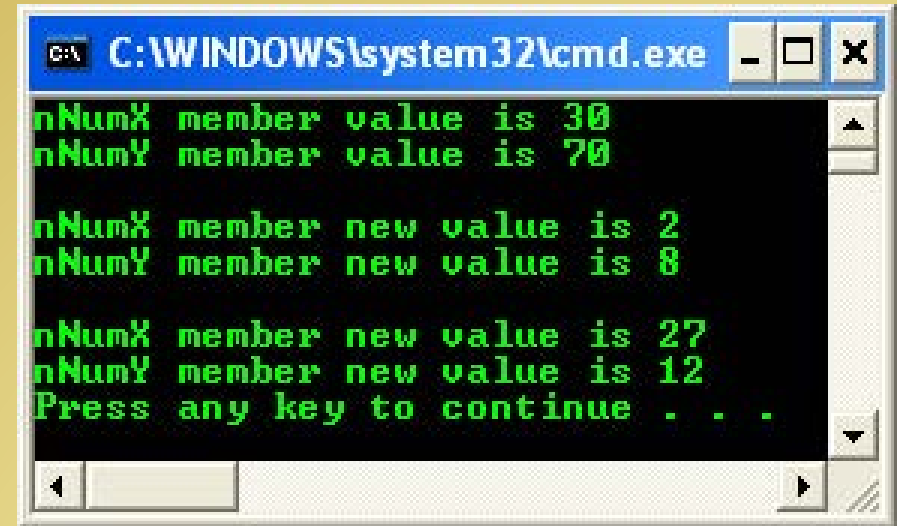
- Define, declare & initialize a structure example



```
C:\WINDOWS\system32\cmd.exe
nNumX member value is 30
nNumY member value is 70

nNumX member new value is 2
nNumY member new value is 8

nNumX member new value is 27
nNumY member new value is 12
Press any key to continue . . .
```



```
C:\WINDOWS\system32\cmd.exe
nNumX member value is 30
nNumY member value is 70

nNumX member new value is 2
nNumY member new value is 8

nNumX member new value is 27
nNumY member new value is 12
Press any key to continue . . .
```

- Define, declare & initialize a structure example 2

STRUCT, TYPEDEF, ENUM & UNION

Declaring and Using Bit Fields in Structures

- A structure declarator can also be a specified number of bits, called a 'bit field'.
- Its length is set off from the declarator for the field name by a colon.
- A bit field is interpreted as an integral type.
- Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow.
- These space-saving structure members bit fields, and their width in bits can be explicitly declared.
- Used in programs that must force a data structure to correspond to a fixed hardware representation and it is not portable.

STRUCT, TYPEDEF, ENUM & UNION

- A bit field declaration contains a type specifier followed by an optional declarator, a colon, a constant integer expression that indicates the field width in bits, and a semicolon.

type-specifier declarator opt : constant-expression;

- A bit field declaration may not use either of the type qualifiers, `const` or `volatile`.
- The following structure example has four bit-field members `left`, `right`, `front` and `rear`, occupying 4, 3, 4 and 5 bits respectively,

```
struct direction { // declare direction bit field
    int left : 4;    // 00000000 0000XXXX
    int right : 3;   // 00000000 0XXX0000
    int front : 4;   // 00000XXX X0000000
    int rear : 5;    // XXXXX000 00000000
};
```

- In this case, total bits is 16, equal to 2 bytes.

STRUCT, TYPEDEF, ENUM & UNION

- The following restrictions apply to bit fields. You cannot,
 1. Define an array of bit fields.
 2. Take the address of a bit field.
 3. Have a pointer to a bit field.
 4. Have a reference to a bit field.

STRUCT, TYPEDEF, ENUM & UNION

Alignment of Bit Fields

- If a series of bit fields does not add up to the size of an `int`, padding can take place.
- The amount of padding is determined by the alignment characteristics of the members of the structure.
- The following example demonstrates padding.
- Suppose that an `int` occupies 4 bytes ($4 \times 8 = 32$ bits). The example declares the identifier `onoffpower` to be of type `struct switching`,

```
struct switching {  
    unsigned light : 1;  
    unsigned fridge : 1;  
    int count;           /* 4 bytes */  
    unsigned stove : 4;  
    unsigned : 4;  
    unsigned radio : 1;  
    unsigned : 0;  
    unsigned flag : 1;  
} onoffpower ;
```

STRUCT, TYPEDEF, ENUM & UNION

- The structure switching contains eight members totaling 16 bytes.
- The following table describes the storage that each member occupies,

Member Name	Storage Occupied	Total	Total
light	1 bit	1 bit	32 bits
fridge	1 bit	1 bit	
(padding up to 30 bits)	To the next int boundary	30 bits	
count	The size of an int (4 bytes)	4 x 8 = 32 bits	32 bits
stove	4 bits	4 bits	32 bits
(unnamed field)	4 bits	4 bits	
radio	1 bit	1 bit	
(padding up to 23 bits)	To the next int boundary (unnamed field)	23 bits	
flag	1 bit	1 bit	32 bits
(padding up to 31 bits)	To the next int boundary	31 bits	
	16 bytes = 64 bits	4 x 32 bits = 128 bits	128 bits

STRUCT, TYPEDEF, ENUM & UNION

- All references to structure fields must be fully qualified. For instance, you cannot reference the second field by `fridge`.
- You must reference this field by `onoffpower.fridge`.
- The following expression sets the `light` field to 1,

```
onoffpower.light = 1;
```

- When you assign to a bit field a value that is out of its range, the bit pattern is preserved and the appropriate bits are assigned.
- The following expression sets the `fridge` field of the `onoffpower` structure to 0 because only the least significant bit is assigned to the fridge field,

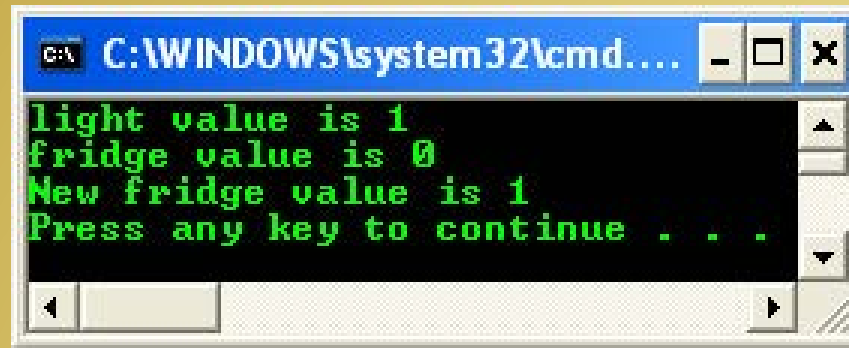
```
onoffpower.fridge = 2;
```

- But the following expression sets the `fridge` field of the `onoffpower` structure to 1.

```
onoffpower.fridge = 5;
```


STRUCT, TYPEDEF, ENUM & UNION

- [Bit fields structure example](#)



```
C:\WINDOWS\system32\cmd....
light value is 1
fridge value is 0
New fridge value is 1
Press any key to continue . . .
```

- Real implementation examples can be found in the Linux socket programming in [fabricating the udp, tcp, ip and other protocol headers](#).

STRUCT, TYPEDEF, ENUM & UNION

Accessing the Structure Member

- A "member-selection expression" refers to members of structures (and unions, classes).
- Such an expression has the value and type of the selected member.
- The member access operators . (dot) and -> (arrow: minus + greater than symbols) are used to refer to members of structures (also unions and classes).
- Member access expressions have the value and type of the selected member.
- There are two methods that can be used to access the members using dot or an arrow operator.

STRUCT, TYPEDEF, ENUM & UNION

- The two syntaxes are,
 1. *postfix-expression* . *Identifier*
 2. *postfix-expression* -> *identifier*
- In the first form, *postfix-expression* represents a value of struct (or union, class object) type and *identifier* names a member of the specified structure (or union, class object).
- The value of the operation is that of *identifier* and is an left-value if *postfix-expression* is an left-value.
- In the second form, *postfix-expression* represents a pointer to a structure or union, and *identifier* names a member of the specified structure or union.
- The value is that of *identifier* and is an left-value.
- The two forms of member-selection expressions have similar effects.

STRUCT, TYPEDEF, ENUM & UNION

- An expression involving arrow member-selection operator (`->`) is a shorthand version of an expression using the dot (`.`) if the expression before the dot consists of the indirection operator (`*`) applied to a pointer value.
- Therefore,

`expression->identifier`

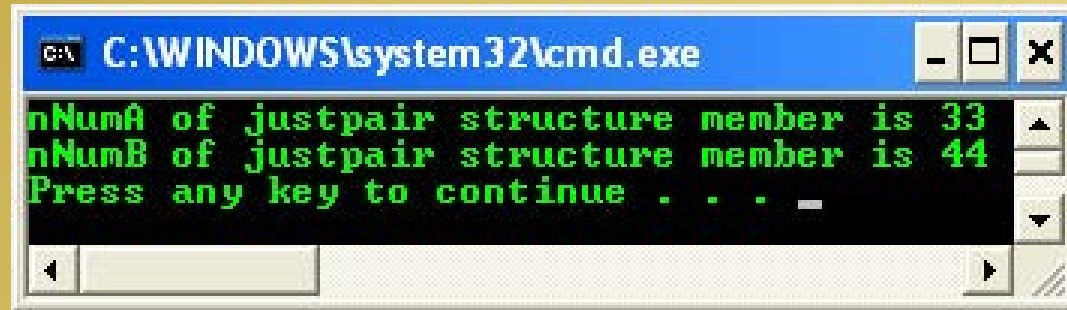
- is equivalent to,

`(*expression).identifier`

- provided that `expression` is a pointer value which is normally the case.

STRUCT, TYPEDEF, ENUM & UNION

- How to access structure member example



```
C:\WINDOWS\system32\cmd.exe
nNumA of justpair structure member is 33
nNumB of justpair structure member is 44
Press any key to continue . . .
```

- A member-selection expression for the `justitem` structure is,

```
justitem.pSctPtr = &justitem;
```

- In this case, the address of the `justitem` structure is assigned to the `pSctPtr` member of the structure.
- This means that `justitem` contains a pointer to itself.

STRUCT, TYPEDEF, ENUM & UNION

```
(justitem.pSctPtr)->nNumA = 33;
```

- In this case, the pointer expression `justitem.pSctPtr` is used with the member-selection operator (`->`) to assign a value of 33 to the member `nNumA`.

```
justalist[8].nNumB = 44;
```

- This statement shows how to access and select an individual structure member from an array of structures.
- An integer 44 was assigned to `nNumB` `justalist`'s ninth structure member.

STRUCT, TYPEDEF, ENUM & UNION

Arrays of Structures

- Suppose you would like to store and manipulate the record of 100 students.
- It would be tedious and unproductive to create 100 different student array variables and work with them individually.
- It would be much easier to create an array of student structures.
- Structures of the same type can be grouped together into an array.
- We can declare an array of structures just like we declare a normal array variable.
- e.g., for 100 student records, we can declare a structure like the following,

```
struct student{  
    int nIdNum, nAge;  
    char chName[80];  
    char chGender;  
}studRecord[100];
```

STRUCT, TYPEDEF, ENUM & UNION

- Or something like the following statements,

```
struct student{  
    int nIdNum, nAge;  
    char chName[80];  
    char chGender;  
};
```

- And later in our program we can declare something like this,

```
struct student studRecord[100];
```

- This statement declares 100 variables of type struct student.

STRUCT, TYPEDEF, ENUM & UNION

- As in arrays, we can use a subscript to reference a particular student record.
- For example, to print the name of the seventh student, we could write the following statement,

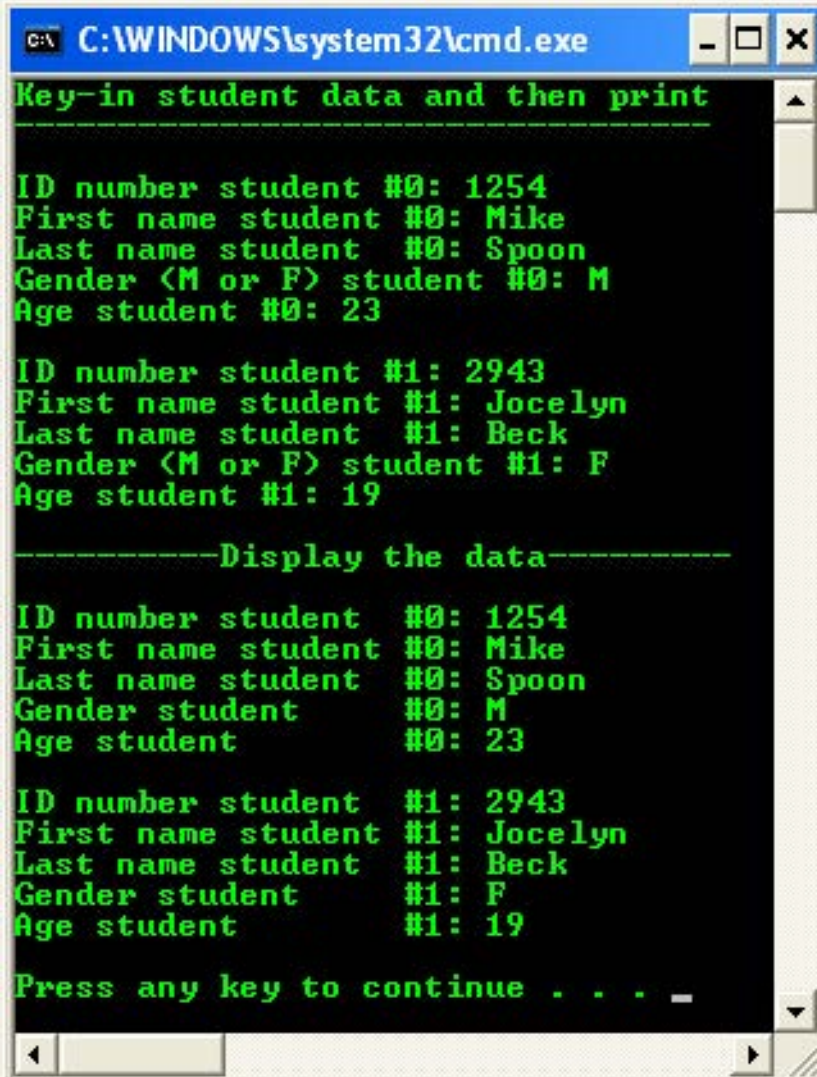
```
printf("%\n", studRecord[6].chName;
```

- Example of initializing all the student names to blanks and their ages to 0, we could do this simply by using `for` loop as shown below,

```
for(i=0; i<100; i++)  
{  
    studRecord[i].chName = " ";  
    studRecord[i].nAge = 0;  
}
```

STRUCT, TYPEDEF, ENUM & UNION

- Array of structure program example



```
C:\WINDOWS\system32\cmd.exe

Key-in student data and then print
-----

ID number student #0: 1254
First name student #0: Mike
Last name student #0: Spoon
Gender <M or F> student #0: M
Age student #0: 23

ID number student #1: 2943
First name student #1: Jocelyn
Last name student #1: Beck
Gender <M or F> student #1: F
Age student #1: 19

-----Display the data-----

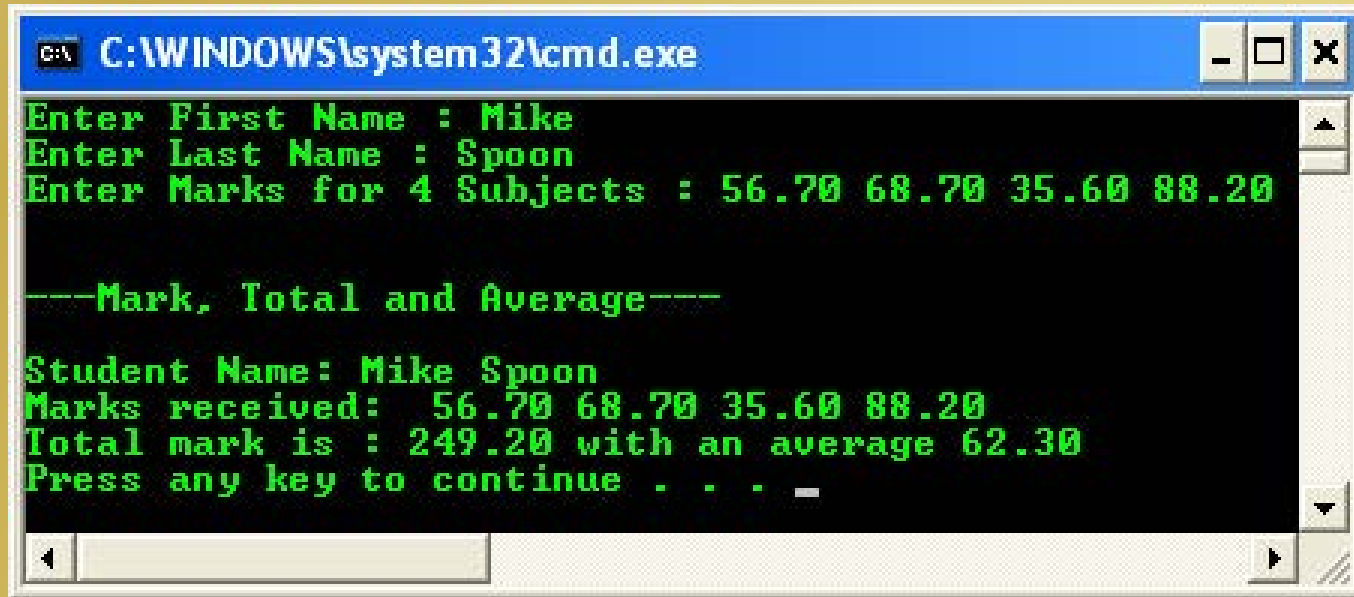
ID number student #0: 1254
First name student #0: Mike
Last name student #0: Spoon
Gender student #0: M
Age student #0: 23

ID number student #1: 2943
First name student #1: Jocelyn
Last name student #1: Beck
Gender student #1: F
Age student #1: 19

Press any key to continue . . .
```

STRUCT, TYPEDEF, ENUM & UNION

- Example demonstrates structure containing arrays



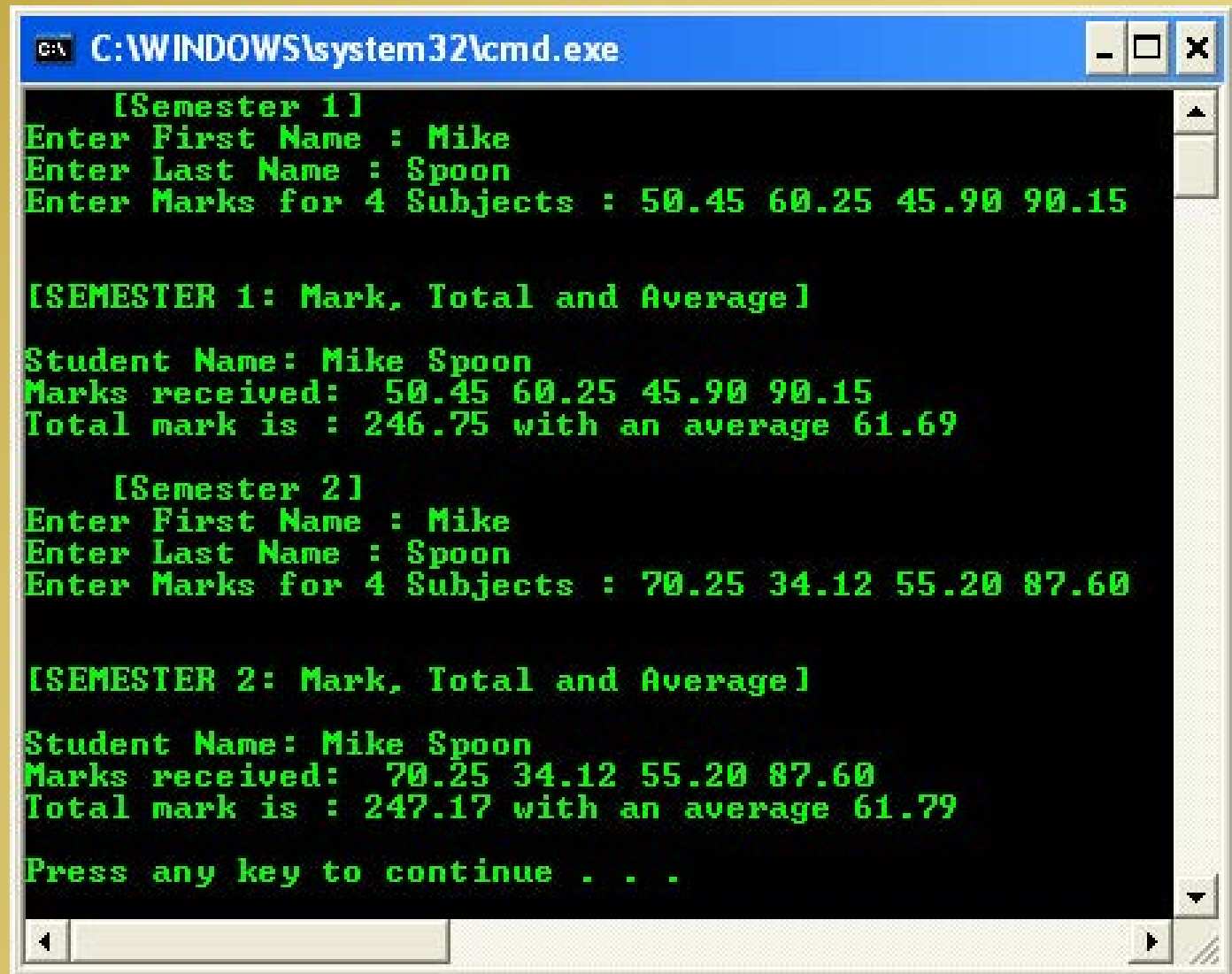
```
C:\WINDOWS\system32\cmd.exe
Enter First Name : Mike
Enter Last Name : Spoon
Enter Marks for 4 Subjects : 56.70 68.70 35.60 88.20

---Mark, Total and Average---

Student Name: Mike Spoon
Marks received: 56.70 68.70 35.60 88.20
Total mark is : 249.20 with an average 62.30
Press any key to continue . . . _
```

STRUCT, TYPEDEF, ENUM & UNION

- Example demonstrates array of structure that containing arrays
- So, take note on the difference between an array of structure and structure containing array.



```
C:\WINDOWS\system32\cmd.exe

[Semester 1]
Enter First Name : Mike
Enter Last Name : Spoon
Enter Marks for 4 Subjects : 50.45 60.25 45.90 90.15

[SEMESTER 1: Mark, Total and Average]
Student Name: Mike Spoon
Marks received: 50.45 60.25 45.90 90.15
Total mark is : 246.75 with an average 61.69

[Semester 2]
Enter First Name : Mike
Enter Last Name : Spoon
Enter Marks for 4 Subjects : 70.25 34.12 55.20 87.60

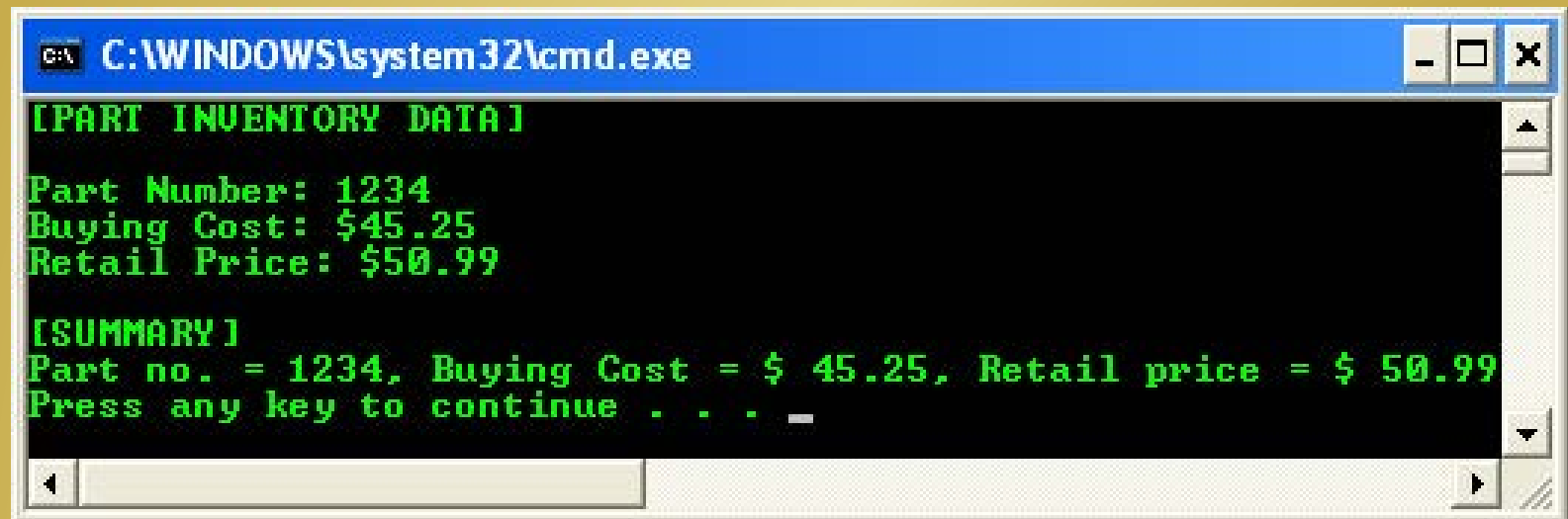
[SEMESTER 2: Mark, Total and Average]
Student Name: Mike Spoon
Marks received: 70.25 34.12 55.20 87.60
Total mark is : 247.17 with an average 61.79

Press any key to continue . . .
```


STRUCT, TYPEDEF, ENUM & UNION

Structure and Function

- Structure variables may be passed as arguments and returned from functions just like scalar variables.
- [Let take a look at the following program example.](#)



A screenshot of a Windows command prompt window. The title bar shows the path `C:\WINDOWS\system32\cmd.exe`. The window contains the following text output from a C program:

```
[PART INVENTORY DATA]

Part Number: 1234
Buying Cost: $45.25
Retail Price: $50.99

[SUMMARY]
Part no. = 1234, Buying Cost = $ 45.25, Retail price = $ 50.99
Press any key to continue . . . _
```

STRUCT, TYPEDEF, ENUM & UNION

- We have declared the structure, `inventory`, at the top of the source file.
- This is called an *external* declaration and the scope is the entire file after the declaration.
- Since all the functions in the file use this structure tag, the structure must be visible to each of them.
- The `main()` calls `readpart()` to read data into a structure and return it to be assigned to the variable, `items`.
- Next, it calls `printpart()` passing `items` which merely prints the values of the structure fields, one at a time.
- External declarations of structure templates and prototypes facilitate consistent usage of tags and functions.
- Sometimes, external structure tag declarations will be placed in a separate header file, which is then made part of the source file by an include directive (`#include`).
- From this example, we can see that using structures with functions is no different than using any scalar data type like `int`.

STRUCT, TYPEDEF, ENUM & UNION

- When the function `readpart()` is called, memory is allocated for all of its local variables, including the `struct inventory` variable, `part`.
- As each data item is read, it is placed in the corresponding field of `part`, accessed with the dot operator.
- The value of `part` is then returned to `main()` where it is assigned to the variable `items`.
- As would be the case for a scalar data type, the value of the return expression is copied back to the calling function.
- Since this is a structure, the entire structure (each of the fields) is copied.
- For our `inventory` structure, this isn't too bad because only two floats and an integer.
- If structure having many members or nested, many values would need to be copied.
- For call to `printpart()`, an `inventory` structure is passed to the function.
- Recall that in C, all parameters are passed by value, hence the value of each argument expression is copied from the calling function into the cell allocated for the parameter of the called function.
- Again, for large structures, this may not be a very efficient way to pass data to functions.

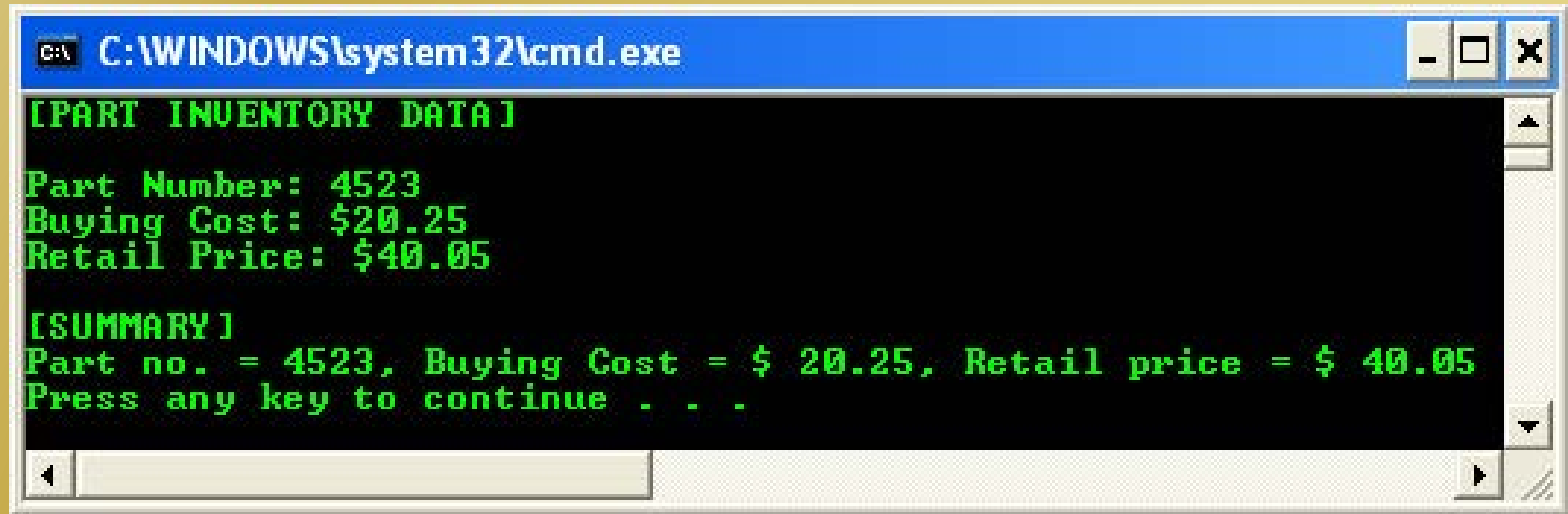
STRUCT, TYPEDEF, ENUM & UNION

Structure, Function and Pointers

- Passing and returning structures to functions may not be efficient, particularly if the structure is large.
- We can eliminate this excessive data movement and computer resources usage by passing pointers to the structures to the function, and access them indirectly through the pointers.
- The following example is a modified version of our previous program which uses pointers instead of passing entire structures.

STRUCT, TYPEDEF, ENUM & UNION

- [Structure, function and pointers example](#) (using * operator)



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a black background with green text. The output of a program is displayed as follows:

```
[PART INVENTORY DATA]

Part Number: 4523
Buying Cost: $20.25
Retail Price: $40.05

[SUMMARY]
Part no. = 4523, Buying Cost = $ 20.25, Retail price = $ 40.05
Press any key to continue . . .
```

The window includes standard Windows window controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

- The code is very similar previous example, but we have changed the prototypes and functions to work with pointers.
- The argument of `readpart()` function is a pointer to the `inventory` structure.
- While `items` variable declared in `main()` is `inventory` structure type.
- The function accesses the object pointed to by `pPartPtr`, and uses the dot operator to access a member of that object.

STRUCT, TYPEDEF, ENUM & UNION

- Since `pPartPtr` points to an object of type `struct inventory`, we dereference the pointer to access the members of the object using the following statements,

```
( *pPartPtr ).nPartNo  
( *pPartPtr ).fBuyCost  
( *pPartPtr ).fSellingPrice
```

- Similar changes have been made to `printpart()` function.
- Note, the parentheses are necessary here because the `.` operator has higher precedence than the indirection operator, `*`.
- We must first dereference the pointer, and then select its appropriate member.
- Since, for efficiency, pointers to structures are often passed to functions, and, within those functions, the members of the structures are accessed, the operation of dereferencing a structure pointer and a selecting a member is very common in programs.

STRUCT, TYPEDEF, ENUM & UNION

- As we have discussed previously, we can use arrow (\rightarrow) operator instead of indirection operator ($*$) because,

`pPartPtr->nPartNo` is equivalent to `(*pPartPtr).nPartNo`

- The left hand expressions are equivalent ways of writing expressions on the right hand side.
- Our code for `readpart ()` could use the following more readable alternative expressions,

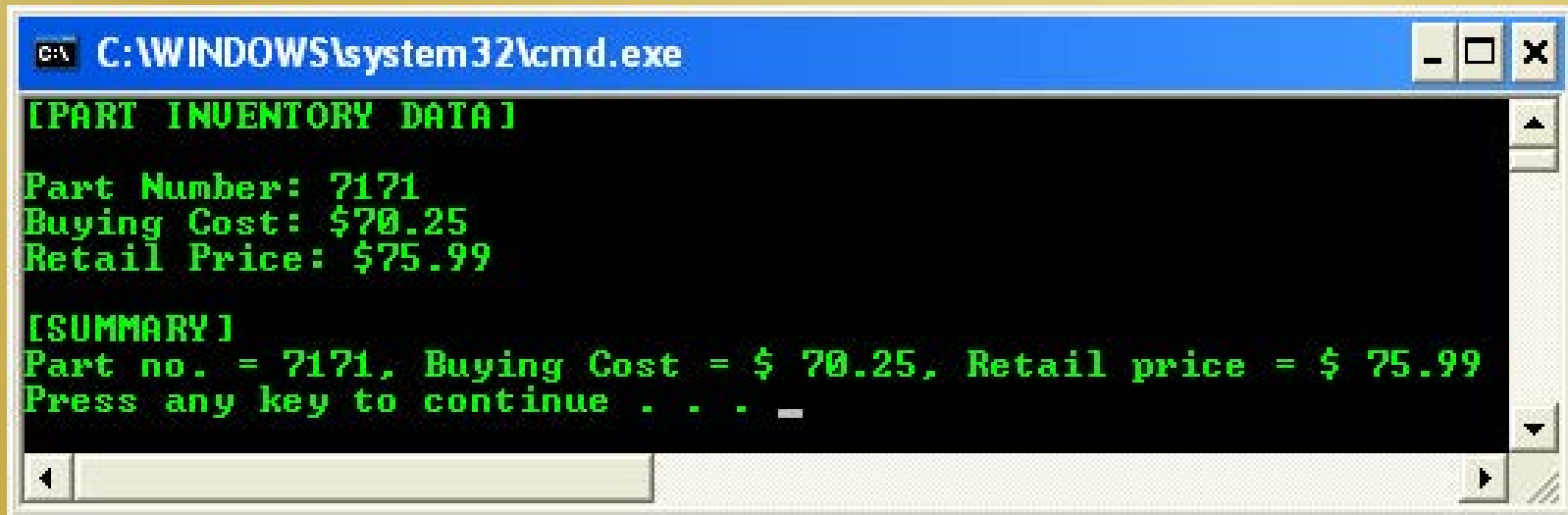
`pPartPtr->nPartNo`

`pPartPtr->fBuyCost`

`pPartPtr->fSellingPrice`

STRUCT, TYPEDEF, ENUM & UNION

- [Structure, function and pointer example 2](#) (using -> operator)



A screenshot of a Windows command prompt window. The title bar is blue and contains the text 'C:\WINDOWS\system32\cmd.exe'. The window has standard minimize, maximize, and close buttons. The command prompt area has a black background with green text. The output is as follows:

```
[PART INVENTORY DATA]

Part Number: 7171
Buying Cost: $70.25
Retail Price: $75.99

[SUMMARY]
Part no. = 7171, Buying Cost = $ 70.25, Retail price = $ 75.99
Press any key to continue . . . _
```

At the bottom of the window, there is a horizontal scrollbar and a small icon in the bottom right corner.

STRUCT, TYPEDEF, ENUM & UNION

Creating Header File and File Re-inclusion Issue

- It is a normal practice to separate structure, function, constants, macros, types and similar definitions and declarations in a separate header file(s).
- Then, those definitions can be included in `main()` using the include directive (`#include`).
- It is for reusability and packaging and to be shared between several source files.
- Include files are also useful for incorporating declarations of external variables and complex data types.
- You need to define and name the types only once in an include file created for that purpose.
- The `#include` directive tells the preprocessor to treat the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears.

STRUCT, TYPEDEF, ENUM & UNION

- The syntax is one of the following,

1. `#include "path-spec"`

2. `#include <path-spec>`

- Both syntax forms cause replacement of that directive by the entire contents of the specified include file.
- The difference between the two forms is the order in which the preprocessor searches for header files when the path is incompletely specified.
- Preprocessor searches for header files when the path is incompletely specified.

STRUCT, TYPEDEF, ENUM & UNION

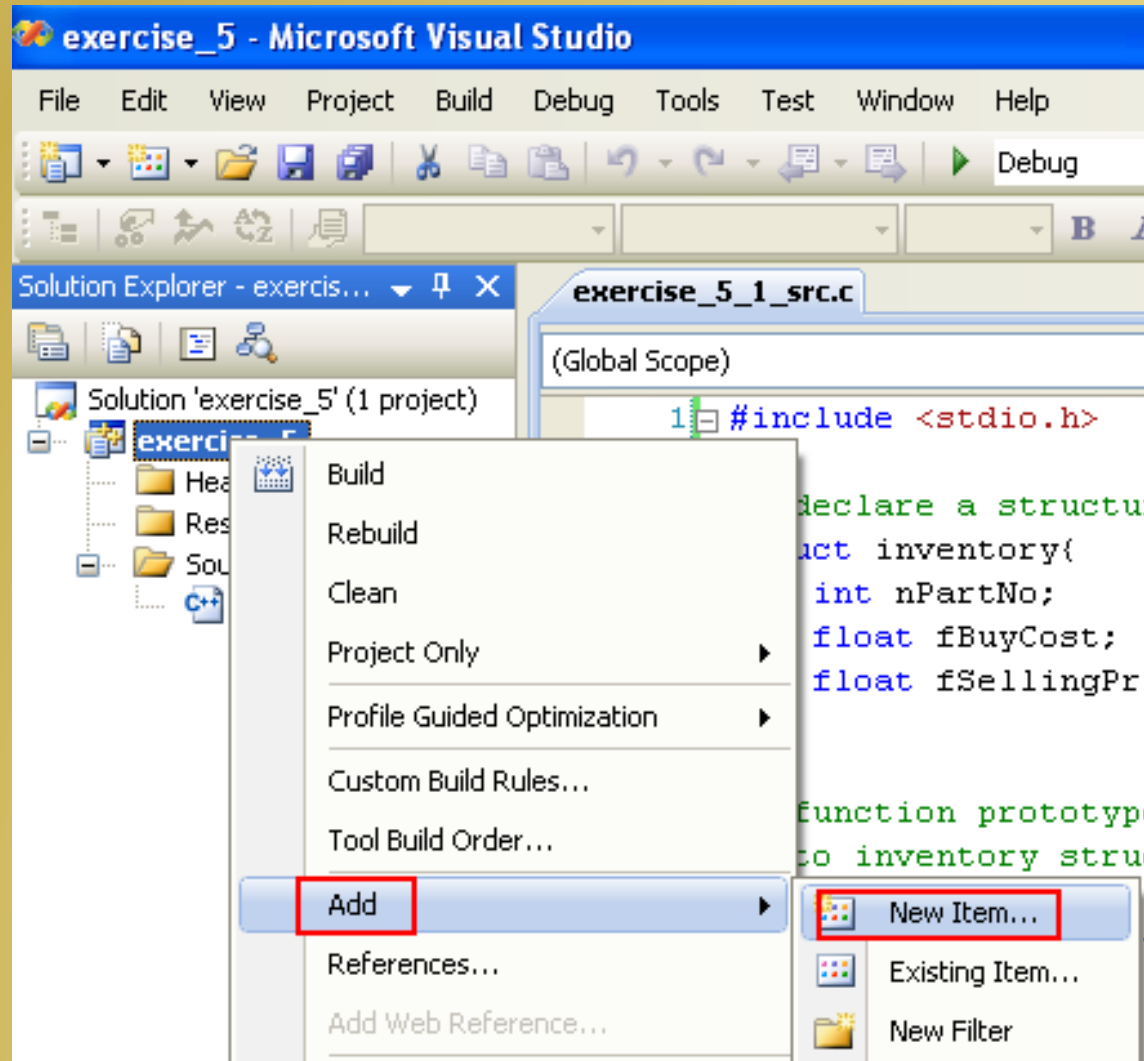
Syntax Form	Action
Quoted form	This form instructs the preprocessor to look for include files in the same directory of the file that contains the <code>#include</code> statement, and then in the directories of any files that include (<code>#include</code>) that file. The preprocessor then searches along the path specified by the <code>/I</code> compiler option (MS VC++), then along paths specified by the <code>INCLUDE</code> environment variable.
Angle-bracket form	This form instructs the preprocessor to search for include files first along the path specified by the <code>/I</code> compiler option (MS VC++), then, when compiling from the command line, along the path specified by the <code>INCLUDE</code> environment variable.

STRUCT, TYPEDEF, ENUM & UNION

- The preprocessor stops searching as soon as it finds a file with the given name.
- If you specify a complete, unambiguous path specification for the include file between double quotation marks (" "), the preprocessor searches only that path specification and ignores the standard directories.
- By using the previous program example, let separate the structure and function declarations and definitions in a header file.

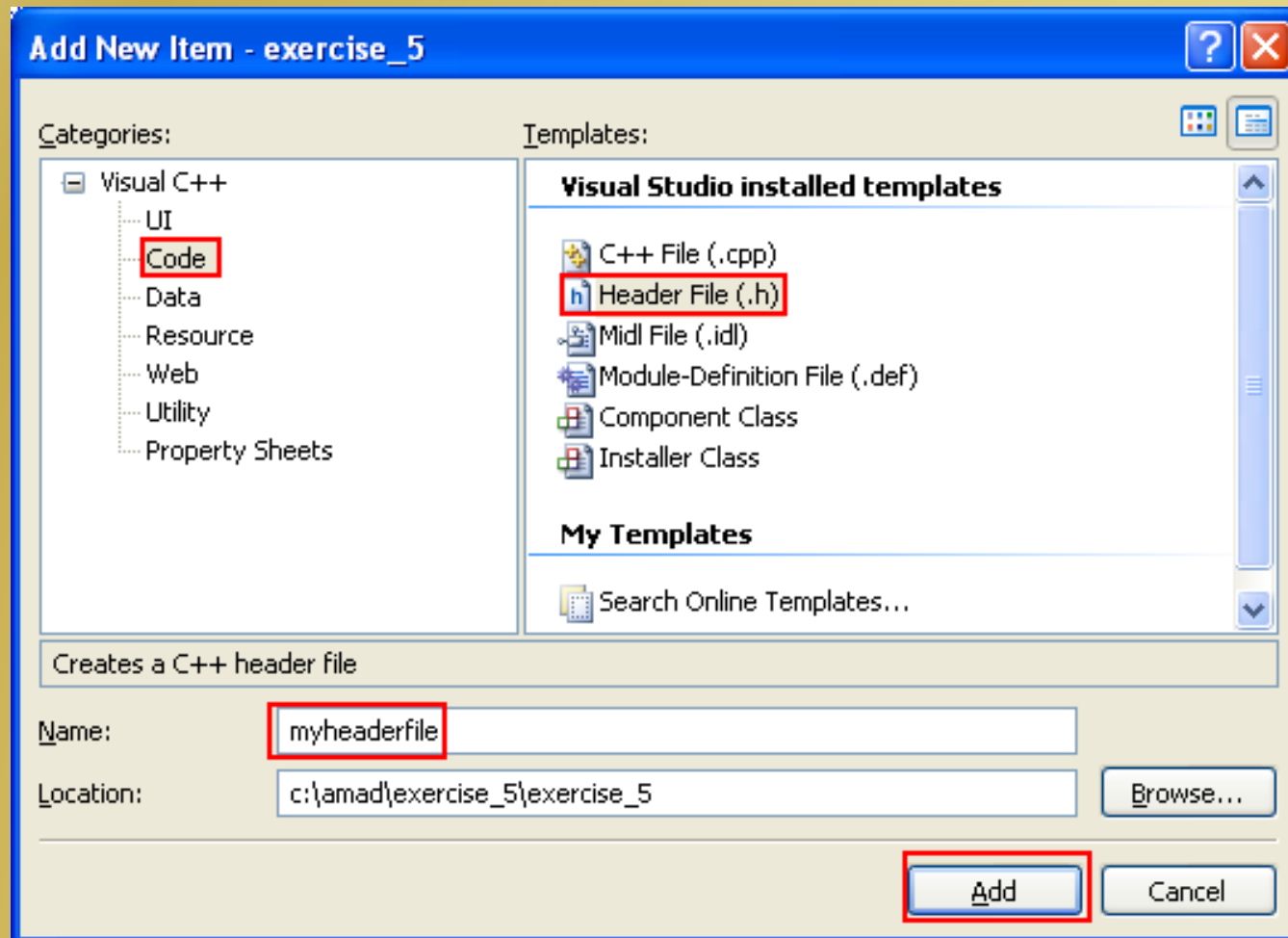
STRUCT, TYPEDEF, ENUM & UNION

- Step: Firstly, add new header file to the existing project.



STRUCT, TYPEDEF, ENUM & UNION

- Step: Put the meaningful header file name.



STRUCT, TYPEDEF, ENUM & UNION

- Step: Next we add (in this case we just cut and paste from main source file) the structure and function definition to the header file.
- It is a normal situation where one header file includes another.
- It can easily result that a certain header file is included more than once (re-inclusion).
- This may lead to errors, if the header file defines structure types or typedefs, and is certainly wasteful.
- Therefore, we often need to prevent multiple inclusion of a header file by using,

```
#ifndef __FILE_NAME_EXTENSION__  
#define __FILE_NAME_EXTENSION__  
...  
#endif
```

STRUCT, TYPEDEF, ENUM & UNION

- The macro `__FILE_NAME_EXTENSION__` indicates that the file has been included once already.
- Its name should begin with `'__'` to avoid conflicts with user programs, and it should contain the name of the file and some additional text, to avoid conflicts with other header files.
- In our example, the `#ifndef` statements should be,

```
#ifndef __MYHEADERFILE_H__  
#define __MYHEADERFILE_H__
```

Declaration and definition go here

```
#endif /* __MYHEADERFILE_H__ */
```

STRUCT, TYPEDEF, ENUM & UNION

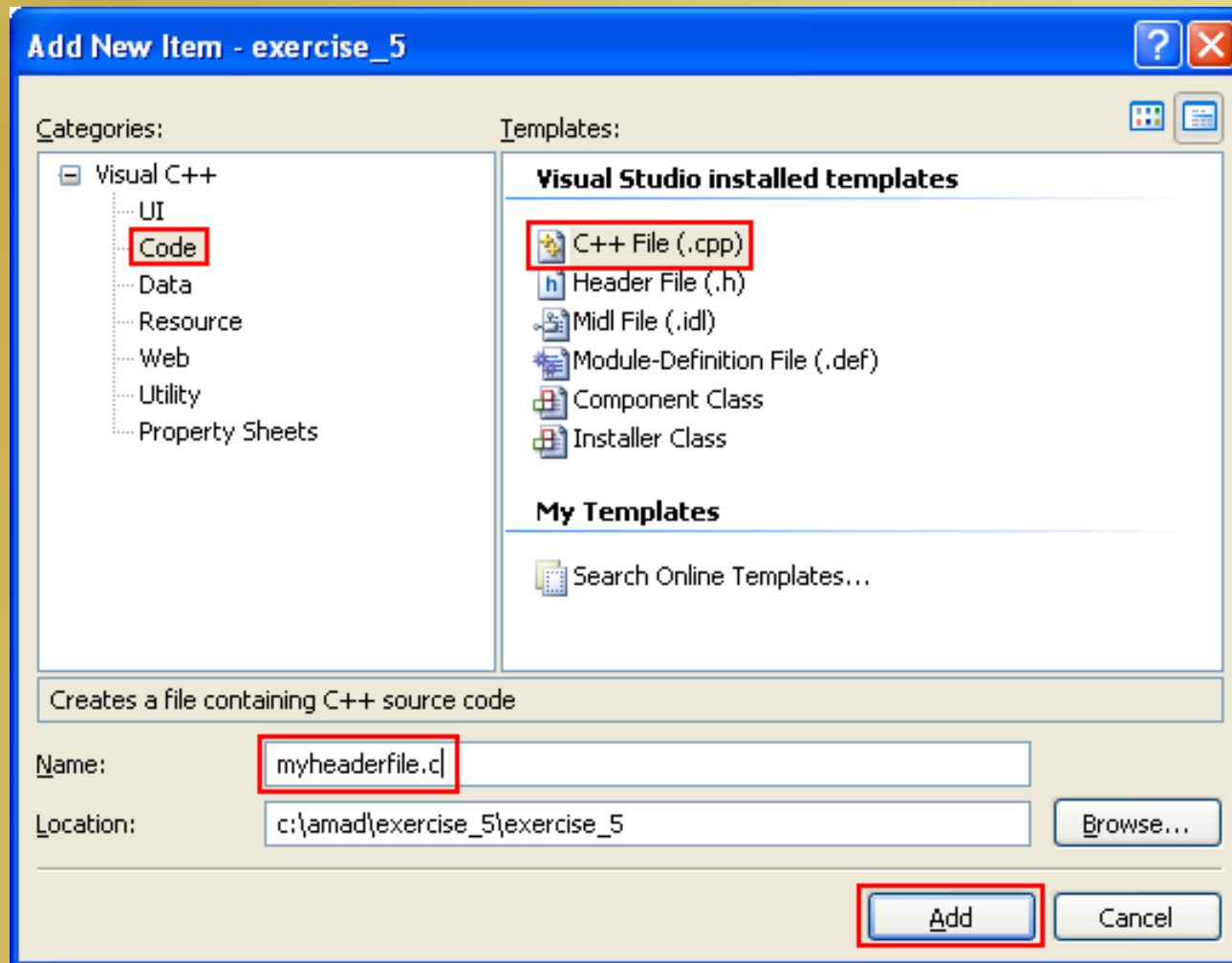
- Step: So, our header file, **myheaderfile.h** content should be something like the following.

```
#ifndef __MYHEADERFILE_H__
#define __MYHEADERFILE_H__
// declare a structure with 3 members
struct inventory{
    int nPartNo;
    float fBuyCost;
    float fSellingPrice;
};
// function prototypes, both will receive a
// pointer
// to inventory structure
void readpart(struct inventory *pPartPtr);
void printpart(struct inventory *pPartPtr);
#endif /* __MYHEADERFILE_H__ */
```

[myheaderfile.txt](#)

STRUCT, TYPEDEF, ENUM & UNION

- Step: Next we create the implementation file for the functions. Add **myheaderfile.c** file to the existing project.



STRUCT, TYPEDEF, ENUM & UNION

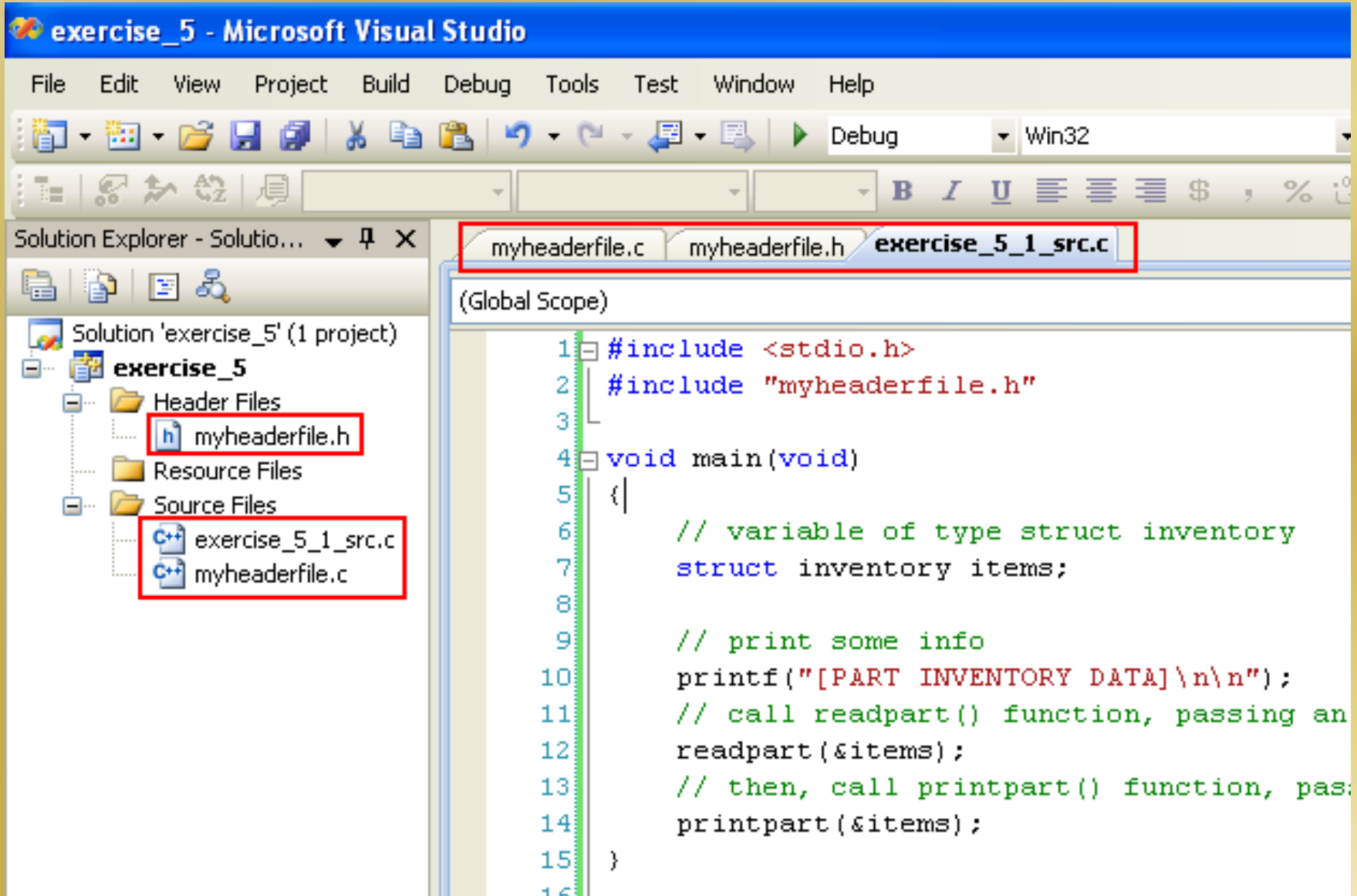
- Step: Add (cut and paste from main source file) the functions implementation part to the file.

[myheaderfile.c in text](#)

- Step: Now our `main()` source file only contain the following code.

[main\(\) in text](#)

STRUCT, TYPEDEF, ENUM & UNION



STRUCT, TYPEDEF, ENUM & UNION

- Step: Re-build and re-run the project.



A screenshot of a Windows command prompt window. The title bar is blue and contains the text 'C:\WINDOWS\system32\cmd.exe'. The window has standard minimize, maximize, and close buttons. The command prompt area has a black background with green text. The output is as follows:

```
[PART INVENTORY DATA]  
  
Part Number: 4356  
Buying Cost: $35.55  
Retail Price: $40.99  
  
[SUMMARY]  
Part no. = 4356, Buying Cost = $ 35.55, Retail price = $ 40.99  
Press any key to continue . . . _
```

At the bottom of the window, there is a scroll bar and a status bar.

- The output should be the same.

STRUCT, TYPEDEF, ENUM & UNION

`#import` and `#pragma once`

- There are two more ways of indicating that a header file should be read only once.
- Neither one is as portable as a wrapper `#ifndef`.
- In Objective-C language, there is a variant of `#include` called `#import` which includes a file, but does so at most once.
- If you use `#import` instead of `#include`, then you don't need the conditionals inside the header file to prevent multiple inclusion of the contents.
- However, it is not in standard C or C++ and should therefore not be used for program portability.
- `#import` is not a well designed feature.
- It requires the users of a header file to know that it should only be included once.

STRUCT, TYPEDEF, ENUM & UNION

- It is much better for the header file's implementor to write the file so that users don't need to know this.
- Using a wrapper `#ifndef` accomplishes this goal.
- Another way to prevent a header file from being included more than once is with the `#pragma once` directive.
- If `#pragma once` is seen when scanning a header file, that file will never be read again, no matter what.
- `#pragma once` does not have the problems that `#import` does, but it is not recognized by all preprocessors, so you cannot rely on it in program portability.
- More real implementation of the source code packaging and organization can found at,
 1. [Reactos C and C++ source code \(browse the left menu\)](#).
 2. [Linux online source code](#).

STRUCT, TYPEDEF, ENUM & UNION

`typedef`

- `typedef` declaration do not introduce new type but introduces new name or creating synonym (or alias) for existing type.
- To construct shorter or more meaningful names for types already defined by the language or for types that you have declared.
- A `typedef` declaration does not reserve storage.
- The name space for a `typedef` name is the same as other ordinary identifiers.
- Therefore, a program can have a `typedef` name and a local-scope identifier by the same name.
- The exception to this rule is if the `typedef` name specifies a variably modified type. In this case, it has block scope.
- The syntax is,

```
typedef    type-declaration    the_synonym;
```


STRUCT, TYPEDEF, ENUM & UNION

- You cannot use the `typedef` specifier inside a function definition.
- When declaring a local-scope identifier by the same name as a `typedef`, or when declaring a member of a structure or union in the same scope or in an inner scope, the type specifier must be specified.
- For example,

```
typedef float TestType;
```

```
int main(void)  
{ ... }
```

```
// function scope (local)  
int MyFunc(int)  
{  
    // same name with typedef, it is OK  
    int TestType;  
}
```

STRUCT, TYPEDEF, ENUM & UNION

- When declaring a local-scope identifier by the same name as a `typedef`, or when declaring a member of a structure or `union` in the same scope or in an inner scope, the type specifier must be specified.
- For example,

```
// both declarations are different  
typedef float TestType;  
const TestType r;
```

- To reuse the `TestType` name for an identifier, a structure member, or a `union` member, the type must be provided, for example,

```
const int TestType;
```

STRUCT, TYPEDEF, ENUM & UNION

- Typedef names share the name space with ordinary identifiers.
- Therefore, a program can have a typedef name and a local-scope identifier by the same name.

```
// a typedef specifier  
typedef char FlagType;
```

```
void main(void)  
{ ... }
```

```
void myproc(int)  
{  
    int FlagType;  
}
```

STRUCT, TYPEDEF, ENUM & UNION

- The following paragraphs illustrate other typedef declaration examples,

```
// a char type using capital letter  
typedef char CHAR;
```

```
// a pointer to a string (char *)  
typedef CHAR * THESTR;  
// then use it as function parameter  
THESTR strchr(THESTR source, CHAR destination);
```

```
typedef unsigned int uint;  
// equivalent to 'unsigned int ui';  
uint ui;
```

STRUCT, TYPEDEF, ENUM & UNION

- To use `typedef` to specify fundamental and derived types in the same declaration, you can separate declarators with comma. For example,

```
typedef char CHAR, *THESTR;
```

- The following example provides the type `Funct ()` for a function returning no value and taking two `int` arguments,

```
typedef void Funct(int, int);
```

- After the above `typedef` statement, the following is a valid declaration,

```
Funct test;
```

- And equivalent to the following declaration,

```
void test(int, int);
```

STRUCT, TYPEDEF, ENUM & UNION

- Names for structure types are often defined with `typedef` to create shorter and simpler type name.
- For example, the following statements,

```
typedef struct Card MyCard;  
typedef unsigned short USHORT;
```

- Defines the new type name `MyCard` as a synonym for type `struct Card` and `USHORT` as a synonym for type `unsigned short`.
- Programmers usually use `typedef` to define a structure (also union, enum and class) type so a structure tag is not required.
- For example, the following definition.

```
typedef struct{  
    char *face;  
    char *suit;  
} MyCard;
```



No tag

STRUCT, TYPEDEF, ENUM & UNION

- Creates the structure type `MyCard` without the need for a separate `typedef` statement.
- Then `MyCard` can be used to declare variables of type `struct Card` and look more natural as normal variable.
- For example, the following declaration,

```
MyCard deck[ 50 ] ;
```

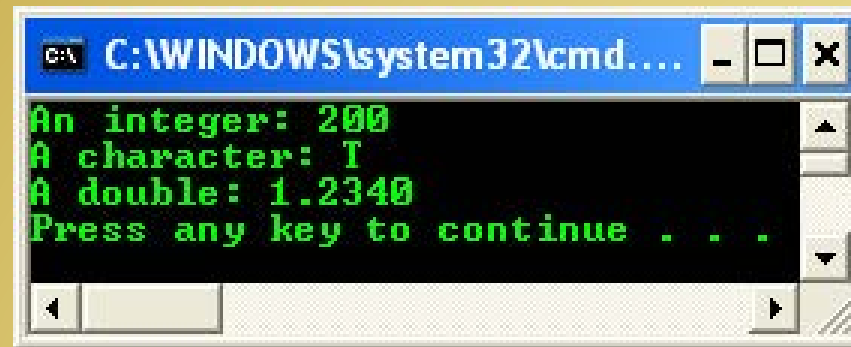
- Declares an array of 50 `MyCard` structures. `typedef` simply creates a new type name which may be used as an alias for an existing type name.

STRUCT, TYPEDEF, ENUM & UNION

- Often, `typedef` is used to create synonyms for the basic data types.
- For example, a program requiring 4-byte integers may use type `int` on one system and type `long` on another system.
- Programs designed for portability often uses `typedef` to create an alias for 4-byte integers such as, let say `Integer`.
- The alias `Integer` can be changed once in the program to make the program work on both systems.
- Microsoft uses this extensively in defining new type name for [Win32 and Windows programming](#).

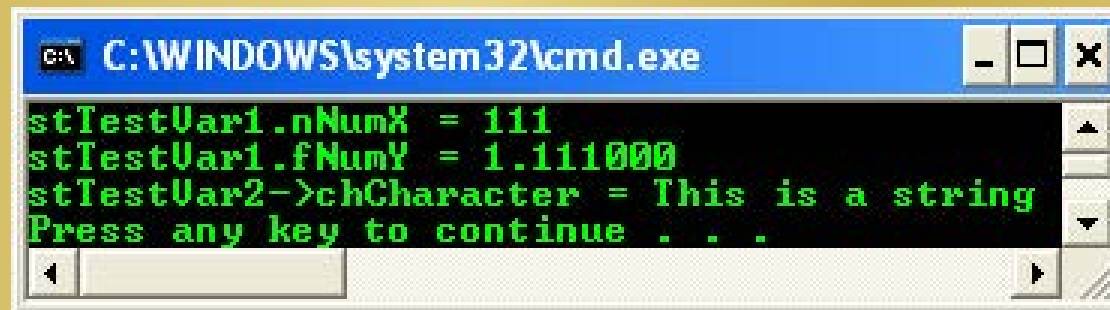
STRUCT, TYPEDEF, ENUM & UNION

- A basic use of typedef program example



```
C:\WINDOWS\system32\cmd.exe
An integer: 200
A character: T
A double: 1.2340
Press any key to continue . . .
```

- Another typedef program example



```
C:\WINDOWS\system32\cmd.exe
stTestVar1.nNumX = 111
stTestVar1.fNumY = 1.111000
stTestVar2->chCharacter = This is a string
Press any key to continue . . .
```

STRUCT, TYPEDEF, ENUM & UNION

- All the structure program examples in the previous structure section can be simplified using typedef. For example,

```
// declare a structure with 3 members
struct inventory{
    int nPartNo;
    float fBuyCost;
    float fSellingPrice;
};
```

- Can be typedef-ed,

```
// declare a structure with 3 members
typedef struct inventory{
    int nPartNo;
    float fBuyCost;
    float fSellingPrice;
}abcinventory;
```

STRUCT, TYPEDEF, ENUM & UNION

Then, the following declaration,

```
// variable of type struct inventory  
struct inventory items;
```

Should become

```
// variable of type struct abcinventory  
abcinventory items;
```

STRUCT, TYPEDEF, ENUM & UNION

enum - Enumeration Constants

- `enum` is another user-defined type consisting of a set of named constants called enumerators.
- Using a keyword `enum`, it is a set of integer constants represented by identifiers.
- The syntax is shown below, ([is optional])

```
// for definition of enumerated type  
enum [tag]  
{  
enum-list  
}  
[declarator];
```

- And

```
// for declaration of variable of type tag  
enum tag declarator;
```

- These enumeration constants are, in effect, symbolic constants whose values can be set automatically.

STRUCT, TYPEDEF, ENUM & UNION

- The values in an `enum` start with 0, unless specified otherwise, and are incremented by 1. For example, the following enumeration,

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

- Creates a new data type, `enum days`, in which the identifiers are set automatically to the integers 0 to 6.
- To number the days 1 to 7, use the following enumeration,

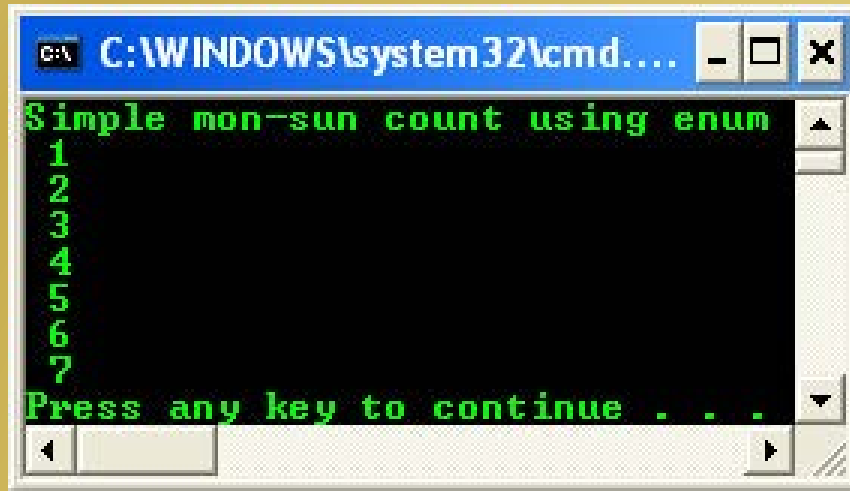
```
enum days {Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun};
```

- Or we can re-arrange the order,

```
enum days {Mon, Tue, Wed, Thu = 7, Fri, Sat, Sun};
```

STRUCT, TYPEDEF, ENUM & UNION

- [A simple enum program example](#)

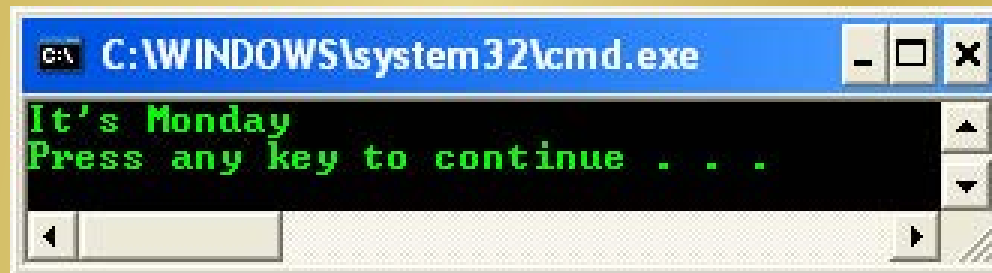


A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd....'. The command prompt displays the text 'Simple mon-sun count using enum' in green. Below this, the numbers 1 through 7 are listed vertically in green. At the bottom, it says 'Press any key to continue . . .'. The window has standard Windows XP-style window controls (minimize, maximize, close) in the title bar.

- As said before, by default, the first enumerator has a value of 0, and each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator.
- Enumerators needn't have unique values within an enumeration.
- The name of each enumerator is treated as a constant and must be unique within the scope where the `enum` is defined.

STRUCT, TYPEDEF, ENUM & UNION

- An enumerator can be promoted to an integer value.
- Converting an integer to an enumerator requires an explicit cast, and the results are not defined if the integer value is outside the range of the defined enumeration.
- Enumerated types are valuable when an object can assume a known and reasonably limited set of values.
- [Another enum program example.](#)



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The command prompt displays the text 'It's Monday' in green, followed by 'Press any key to continue . . .' in green. The window has standard Windows XP-style window controls (minimize, maximize, close) in the top right corner and a scroll bar on the right side.

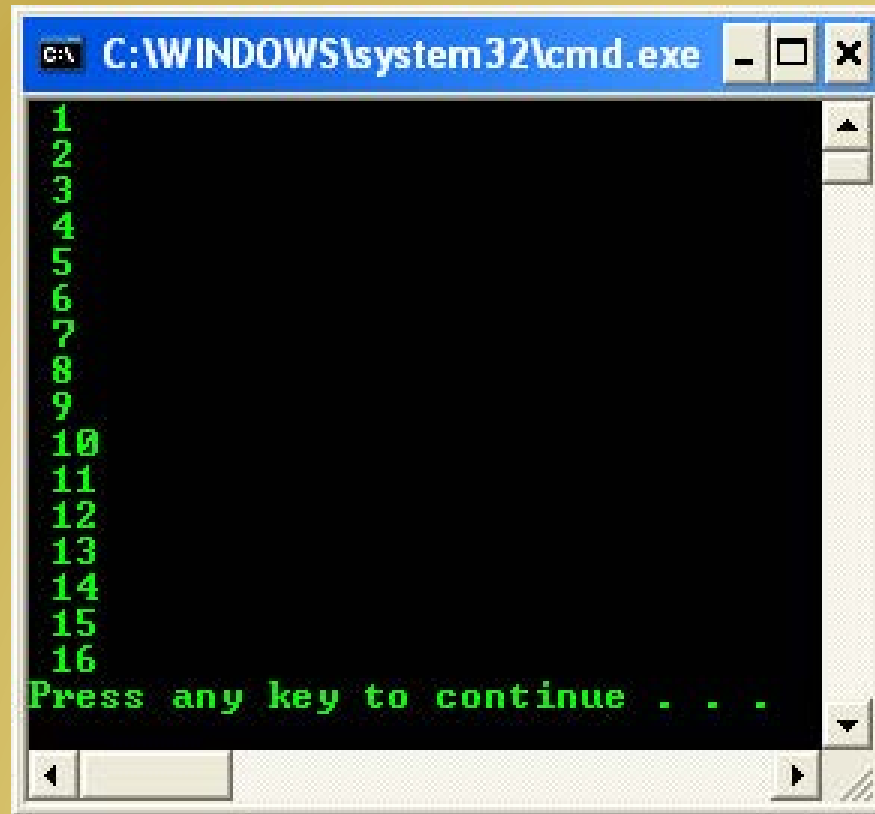
STRUCT, TYPEDEF, ENUM & UNION

- After the `enum` data type has been declared and defined, in C++ it is legal to use the `enum` data type without the keyword `enum`.
- From the previous example, however, the following statement is legal in C++,

```
// is legal in C++  
Days WhatDay = tuesday;
```

- Enumerators are considered defined immediately after their initializers; therefore, they can be used to initialize succeeding enumerators.
- The following example defines an enumerated type that ensures any two enumerators can be combined with the OR operator.

STRUCT, TYPEDEF, ENUM & UNION

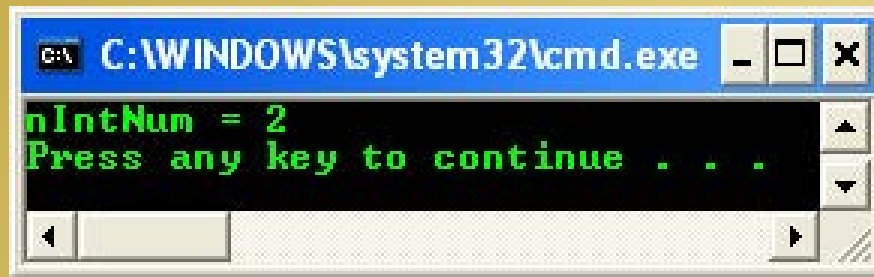


```
C:\WINDOWS\system32\cmd.exe
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Press any key to continue . . .
```

- In this example, the preceding enumerator initializes each succeeding enumerator.

STRUCT, TYPEDEF, ENUM & UNION

- Enumerated types are integral types; any enumerator can be converted to another integral type by integral promotion.
- Consider the following enum example,



- However, there is no implicit conversion from any integral type to an enumerated type.
- Therefore from the previous example, the following statement is an error,

```
// erroneous attempt to set enDays to
// Saturday
enDays = 5;
```


STRUCT, TYPEDEF, ENUM & UNION

- The assignment `enDays = 5`, where no implicit conversion exists, must use casting to perform the conversion,

```
// explicit cast-style conversion to type Days
enDays = (Days)5;
// explicit function-style conversion to type Days
enDays = Days(4);
```

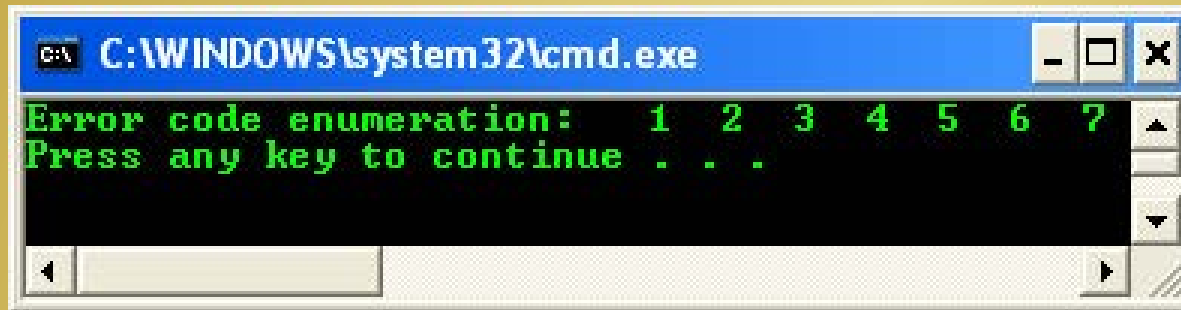
- The preceding example shows conversions of values that coincide with the enumerators.
- There is no mechanism that protects you from converting a value that does not coincide with one of the enumerators. For example,

```
enDays = Days(30);
```

- Some such conversions may work but there is no guarantee the resultant value will be one of the enumerators.

STRUCT, TYPEDEF, ENUM & UNION

- The following program example uses enum and typedef.



```
C:\WINDOWS\system32\cmd.exe
Error code enumeration:  1  2  3  4  5  6  7
Press any key to continue . . .
```

STRUCT, TYPEDEF, ENUM & UNION

Union

- A `union` is a user-defined data or class type that, at any given time, contains only one object from its list of members (although that object can be an array or a class type).

```
union [tag] { member-list } [declarators];
```

- Then, in program we can declare `union` type variable as,

```
[union] tag declarators;
```

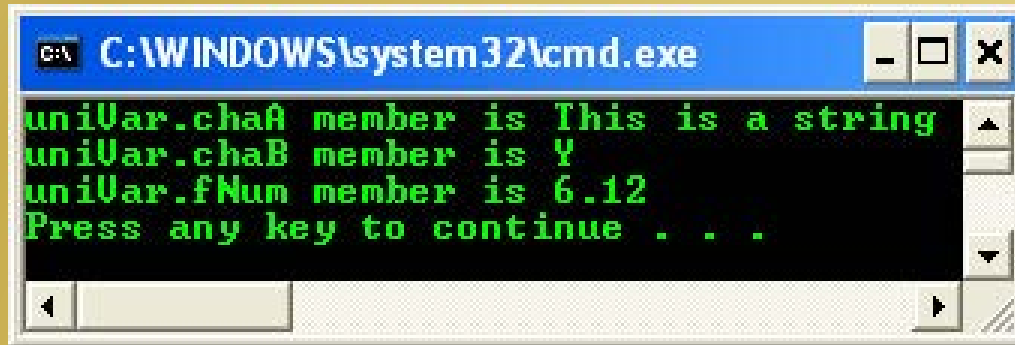
- In C++ the `union` keyword is optional.

STRUCT, TYPEDEF, ENUM & UNION

- The storage associated with a union variable is the storage required for the largest member of the union.
- When a smaller member is stored, the union variable can contain unused memory space.
- All members are stored in the same memory space and start at the same address.
- Hence, the stored value is overwritten each time a value is assigned to a different member.
- Begin the declaration of a union with the `union` keyword, and enclose the member list in curly braces.

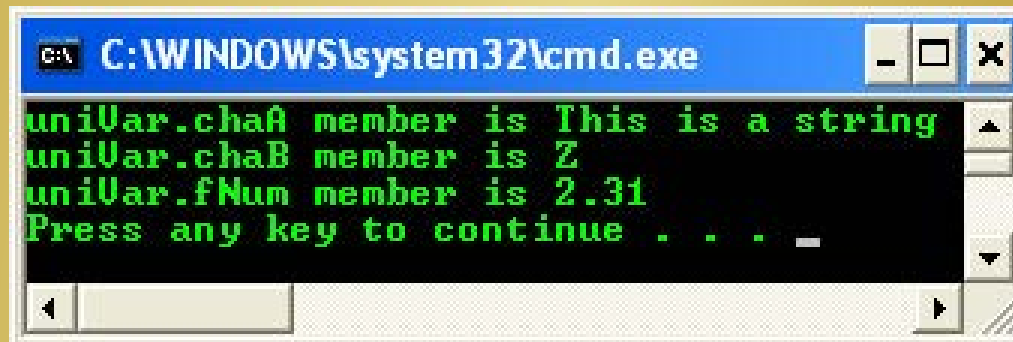
STRUCT, TYPEDEF, ENUM & UNION

- Union with array and pointer example.



```
C:\WINDOWS\system32\cmd.exe
uniVar.chaA member is This is a string
uniVar.chaB member is Y
uniVar.fNum member is 6.12
Press any key to continue . . .
```

- Another variation of the previous union example.



```
C:\WINDOWS\system32\cmd.exe
uniVar.chaA member is This is a string
uniVar.chaB member is Z
uniVar.fNum member is 2.31
Press any key to continue . . .
```

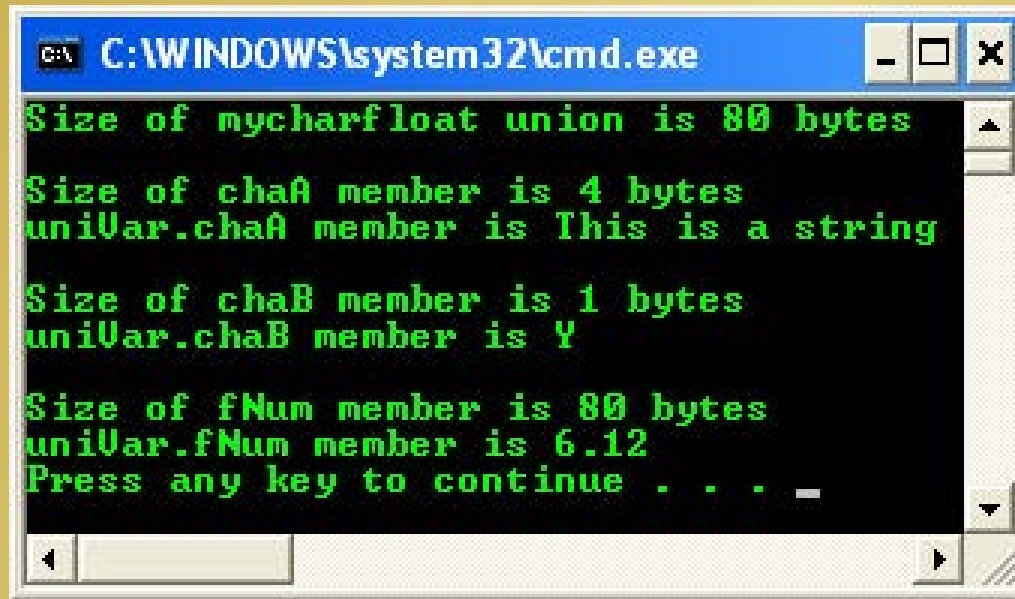
STRUCT, TYPEDEF, ENUM & UNION

- The members of the `myuniontype` union are, in order of their declaration, a pointer to a `char`, a `char`, and an array of `float` values.
- The storage allocated for `myuniontype` is the storage required for the 20-element array `fNum`, since `fNum` is the longest member of the union.
- If the tag (`mycharfloat`) associated with the union is omitted, the union is said to be unnamed or anonymous.
- A union may only be initialized with a value of the type of its first member; thus union `myuniontype` described above can only be initialized with a pointer to a string value.

STRUCT, TYPEDEF, ENUM & UNION

union vs struct Storage Allocation

- Let check the `union`'s size and its biggest member.
- In the following program examples, we are using stack memory instead of heap.
- Hence, no need 'manual' memory allocation using function such as `malloc()` etc.
- [Union, array and pointers example: storage size](#)

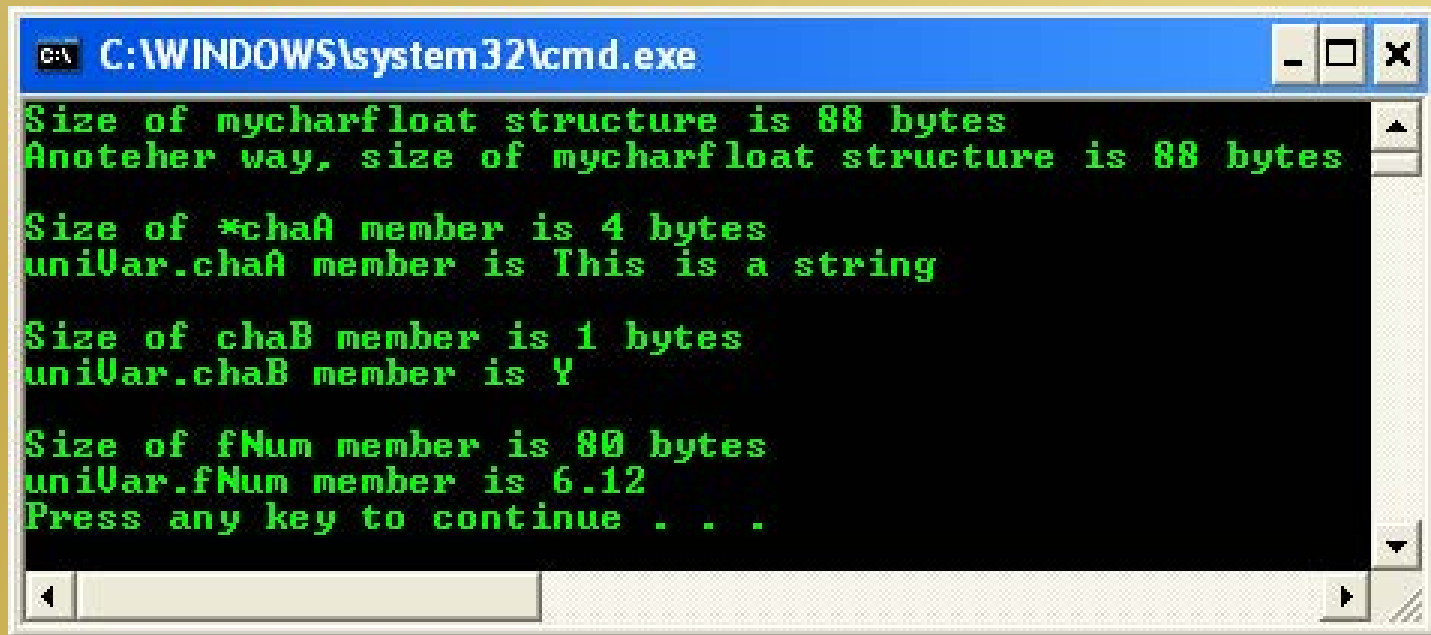


```
C:\WINDOWS\system32\cmd.exe
Size of mycharfloat union is 80 bytes
Size of chaA member is 4 bytes
uniVar.chaA member is This is a string
Size of chaB member is 1 bytes
uniVar.chaB member is Y
Size of fNum member is 80 bytes
uniVar.fNum member is 6.12
Press any key to continue . . .
```

- The size is the size of the biggest element which is an array of float (20 x 4 bytes = 80 bytes).

STRUCT, TYPEDEF, ENUM & UNION

- Let change the union to structure.
- [Structure, array and pointers example: storage size](#)



```
C:\WINDOWS\system32\cmd.exe
Size of mycharfloat structure is 88 bytes
Anotehar way, size of mycharfloat structure is 88 bytes

Size of *chaA member is 4 bytes
uniVar.chaA member is This is a string

Size of chaB member is 1 bytes
uniVar.chaB member is Y

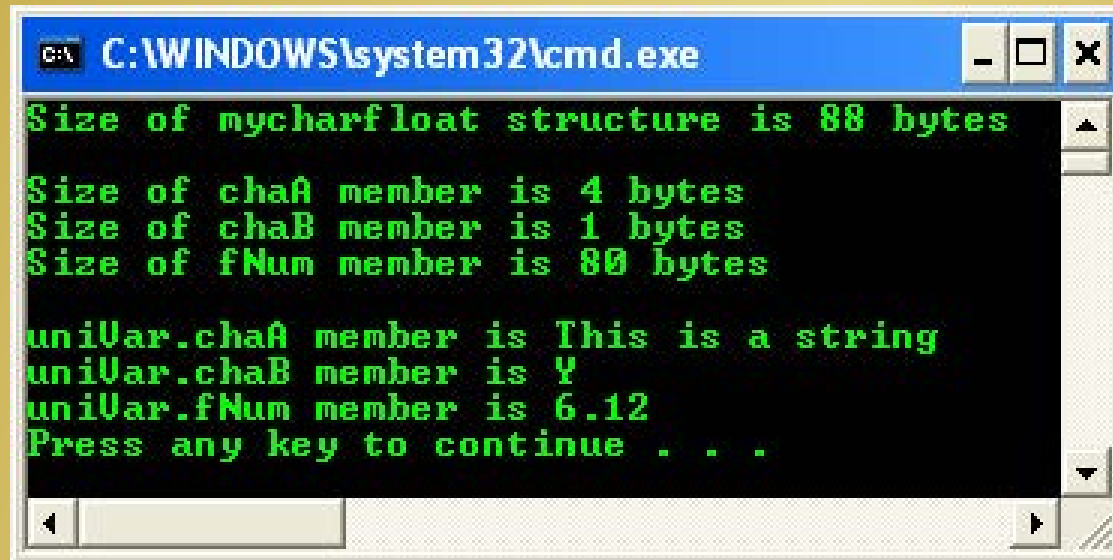
Size of fNum member is 80 bytes
uniVar.fNum member is 6.12
Press any key to continue . . .
```

- Well the size is 88 bytes but the biggest element is 80 bytes.
- Why 88 bytes? The `char` pointer is 4 bytes on 32 bits system, `char` is 1 byte and an array of 20 floats is 80 bytes which makes the total 85 bytes.
- We lack another 3 bytes which used for 32-bits system memory alignment of the `char` ($4 - 1 = 3$ bytes).

STRUCT, TYPEDEF, ENUM & UNION

Accessing union Member

- If we put the some of the `printf()` at the end of source code, structure will generate the expected output but not for union, because the last `printf()` access will overwrite the previous stored data or at least the result is unreliable.
- [Union, array and pointers program example: structure member access](#)



```
C:\WINDOWS\system32\cmd.exe
Size of mycharfloat structure is 88 bytes
Size of chaA member is 4 bytes
Size of chaB member is 1 bytes
Size of fNum member is 80 bytes
uniVar.chaA member is This is a string
uniVar.chaB member is Y
uniVar.fNum member is 6.12
Press any key to continue . . .
```

- The string pointed to by `chaA` pointer was displayed properly.

STRUCT, TYPEDEF, ENUM & UNION

- [Let change the same program to union.](#)

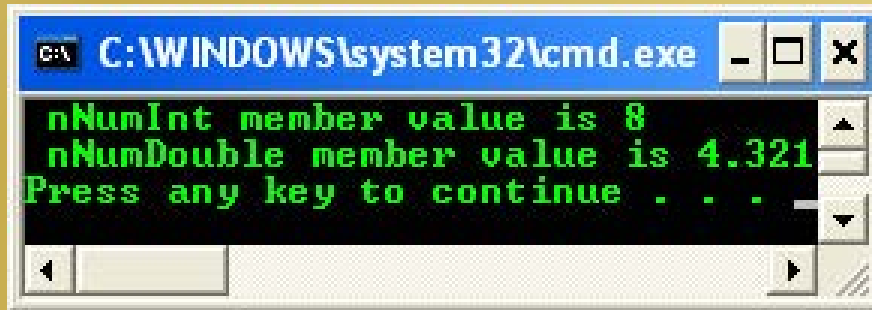


```
C:\WINDOWS\system32\cmd.exe
Size of mycharfloat structure is 80 bytes
Size of chaA member is 4 bytes
Size of chaB member is 1 bytes
Size of fNum member is 80 bytes
uniVar.chaA member is
uniVar.chaB member is Y
uniVar.fNum member is 6.12
Press any key to continue . . .
```

- The string pointed by chaA pointer is nowhere can be seen.
- If a union of two types is declared and one value is stored, but the union is accessed with the other type, the results are unreliable.
- For example, a union of float and int is declared. A float value is stored, but the program later accesses the value as an int.
- In such a situation, the value would depend on the internal storage of float values.
- The integer value would not be reliable.

STRUCT, TYPEDEF, ENUM & UNION

- [Let try another example](#)



```
C:\WINDOWS\system32\cmd.exe
nNumInt member value is 8
nNumDouble member value is 4.321
Press any key to continue . . .
```

- Default access of members in a union is `public`.
- A C `union` type can contain only data members.
- In C, you must use the `union` keyword to declare a union variable. In C++, the `union` keyword is unnecessary.
- A variable of a `union` type can hold one value of any type declared in the `union`. Use the member-selection operator (`.`) to access a member of a `union`.

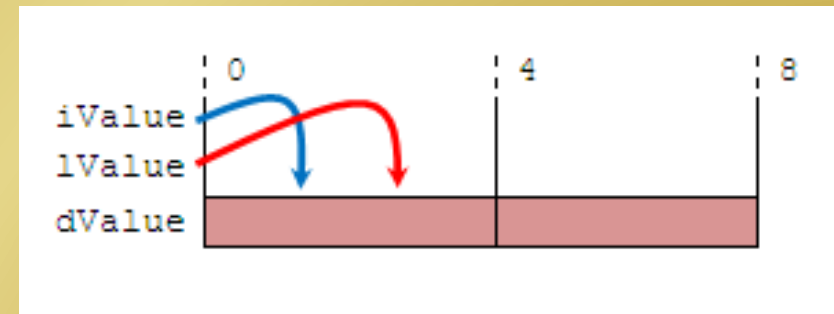
STRUCT, TYPEDEF, ENUM & UNION

Initializing union member

- You can declare and initialize a `union` in the same statement by assigning an expression enclosed in braces.
- The expression is evaluated and assigned to the first field of the `union`.
- For example: [union initialization program example](#).

```
C:\WINDOWS\system32\cmd.exe
Size of iValue member is 4 bytes.
Size of lValue member is 4 bytes.
Size of dValue member is 8 bytes.

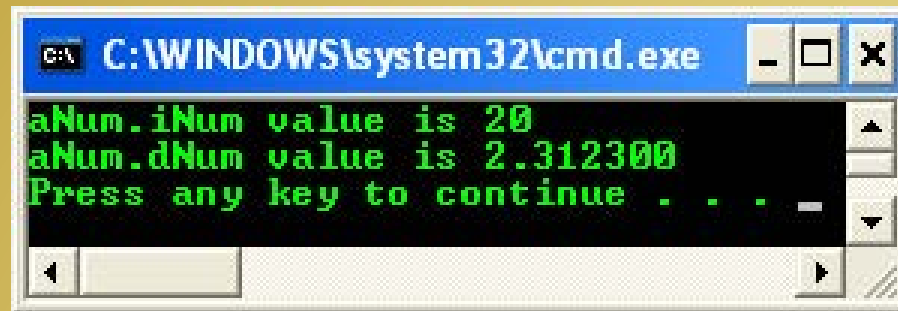
iValue member is 10
dValue member is 3.141600
Press any key to continue . . .
```



- The `NumericType` union is arranged in memory (conceptually) as shown in the following figure.
- The memory spaces of `union` members are overlaps.

STRUCT, TYPEDEF, ENUM & UNION

- Another example to initialize a union.



```
C:\WINDOWS\system32\cmd.exe
aNum.iNum value is 20
aNum.dNum value is 2.312300
Press any key to continue . . .
```

- Unions can contain most types in their member lists, except for the following,
 1. Class types that have constructors or destructors.
 2. Class types that have user-defined assignment operators.
 3. Static data members.

STRUCT, TYPEDEF, ENUM & UNION

Anonymous union

- Anonymous unions are unions that are declared without a *class-name* or *declarator-list*.

```
union { member-list }
```

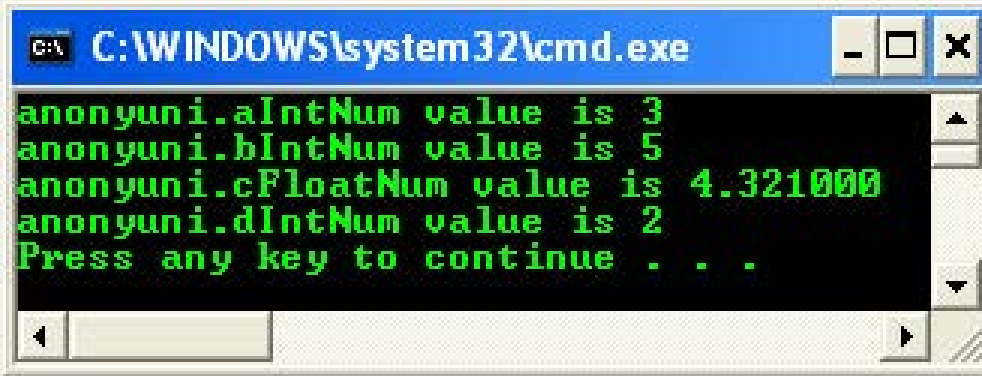
- Such union declarations do not declare types but they declare objects.
- The names declared in an anonymous union cannot conflict with other names declared in the same scope.
- In C, an anonymous union can have a tag; it cannot have declarators.
- Names declared in an anonymous `union` are used directly, like non-member variables.

STRUCT, TYPEDEF, ENUM & UNION

1. In addition to the restrictions mentioned in 'Union can contain most types...', anonymous unions are subject to additional restrictions,
 - a) They must also be declared as `static` if declared in file scope. If declared in local scope, they must be `static` or `automatic`.
 - b) They can have only public members; `private` and `protected` members in anonymous unions generate errors (compared to C++ class).
 - c) They cannot have function members.
2. Simply omitting the *class-name* portion of the syntax does not make a `union` an anonymous union.
3. For a `union` to qualify as an anonymous union, the declaration must not declare an object.

STRUCT, TYPEDEF, ENUM & UNION

- Let try a program example: anonymous union



```
C:\WINDOWS\system32\cmd.exe
anonyuni.aIntNum value is 3
anonyuni.bIntNum value is 5
anonyuni.cFloatNum value is 4.321000
anonyuni.dIntNum value is 2
Press any key to continue . . .
```

The screenshot shows a Windows command prompt window with the title bar "C:\WINDOWS\system32\cmd.exe". The window contains the following text in green on a black background: "anonyuni.aIntNum value is 3", "anonyuni.bIntNum value is 5", "anonyuni.cFloatNum value is 4.321000", "anonyuni.dIntNum value is 2", and "Press any key to continue . . .". The window has standard Windows controls (minimize, maximize, close) in the title bar and a scroll bar on the right side.

STRUCT, TYPEDEF, ENUM & UNION

- Anonymous `union` (and `struct`) is not in the C99 standard, but they are in the C11 standard.
- GCC and clang already support this.
- The C11 standard seems have been removed from Microsoft, and GCC has provided support for some MSFT extensions for some time.
- As in structure, `typedef` can be used to simplify the `union` declaration.

```
// defines a union named myuniontype with
// members consist
// of char pointer, char and array of 20 floats
typedef union mycharfloat
{
    char *chaA, chaB;
    float fNum[20];
} myuniontype;
```

- Later, in program we just write like the following,

```
myuniontype myUnionVar;
```

STRUCT, TYPEDEF, ENUM & UNION

- Union are particularly useful in embedded programming or in situations where direct access to the hardware/memory is needed depending on the *endianism* and processor architecture.
- For example, `union` can be used to breakdown hardware registers into the component bits. So, you can access an 8-bit register bit-by-bit.
- Unions are used when you want to model `struct` defined by hardware, devices or network protocols, or when you are creating a large number of objects and want to save space.
- In most occasion of the program running, you really don't need them 90% of the time. (just an assumption!)
- Most of the cases the `union` is wrapped in a `struct` and one member of the `struct` tells which element in the `union` to access.
- In effect, a `union` is a structure in which all members have offset zero from the base.
- The same operations are permitted on `union` as on `struct`: assignment to or copying as a unit, taking the address, and accessing a member.

C STRUCT, TYPEDEF, ENUM & UNION

-end of the session-