C FUNCTIONS

-INDEPENDENT ROUTINE WHICH DO A SPECIFIC TASK(S)-

C FUNCTIONS

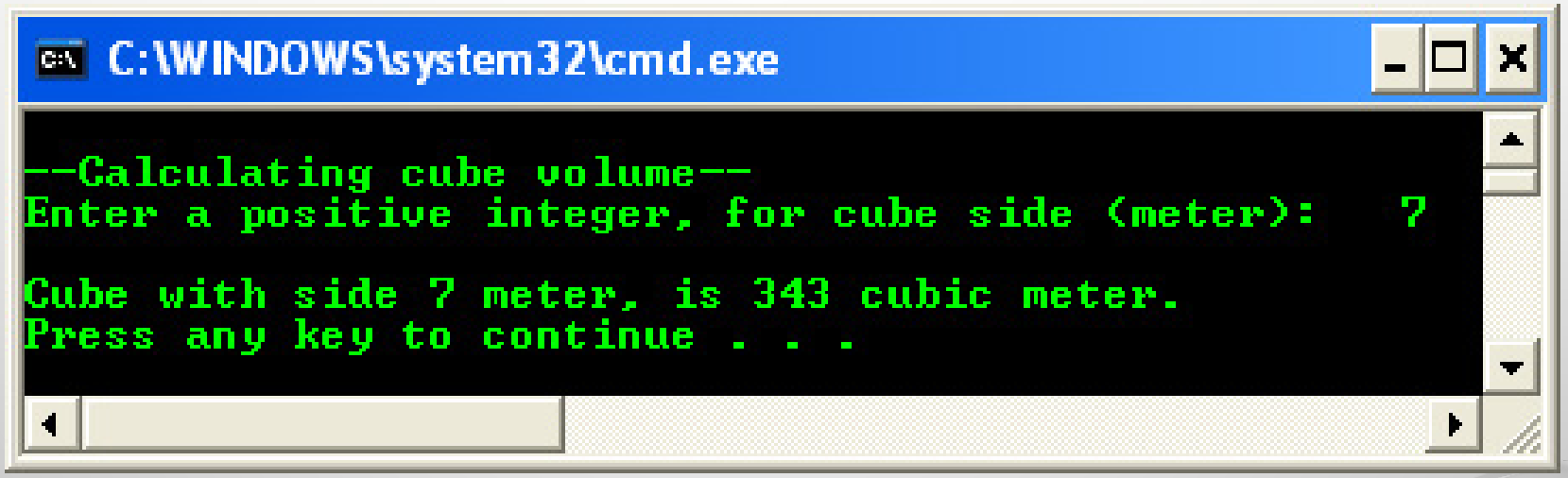
Some definition: A function is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program or/and receives values(s) from the calling program.

- Basically there are two categories of function:

1. Predefined functions: available in C / C++ standard library such as stdio.h, math.h, string.h etc.
2. User-defined functions: functions that programmers create for specialized tasks such as graphic and multimedia libraries, implementation extensions or dependent etc.

C FUNCTIONS

- Let try a simple program example that using a simple user defined function,



C FUNCTIONS

- The following statement call `cube()` function, bringing along the value assigned to the fInput variable.

```
fAnswer  =  cube(fInput);
```

- When this statement is executed, program jump to the `cube()` function definition.
- After the execution completed, the `cube()` function returns to the caller program (`main()`), assigning the returned value, `fCubeVolume` to `fAnswer` variable for further processing (if any).
- In this program the `scanf()` and `print()` are examples of the standard predefined functions.

C FUNCTIONS

- Basically a function has the following characteristics:

1. *Named with unique name* .
2. *Performs a specific task* - Task is a discrete job that the program must perform as part of its overall operation, such as sending a line of text to the printer, sorting an array into numerical order, or calculating a cube root, etc.
3. *Independent* - A function can perform its task without interference from or interfering with other parts of the program.
4. *May receive values from the calling program (caller)* - Calling program can pass values to function for processing whether directly or indirectly (by reference).
5. *May return a value to the calling program* – the called function may pass something back to the calling program.

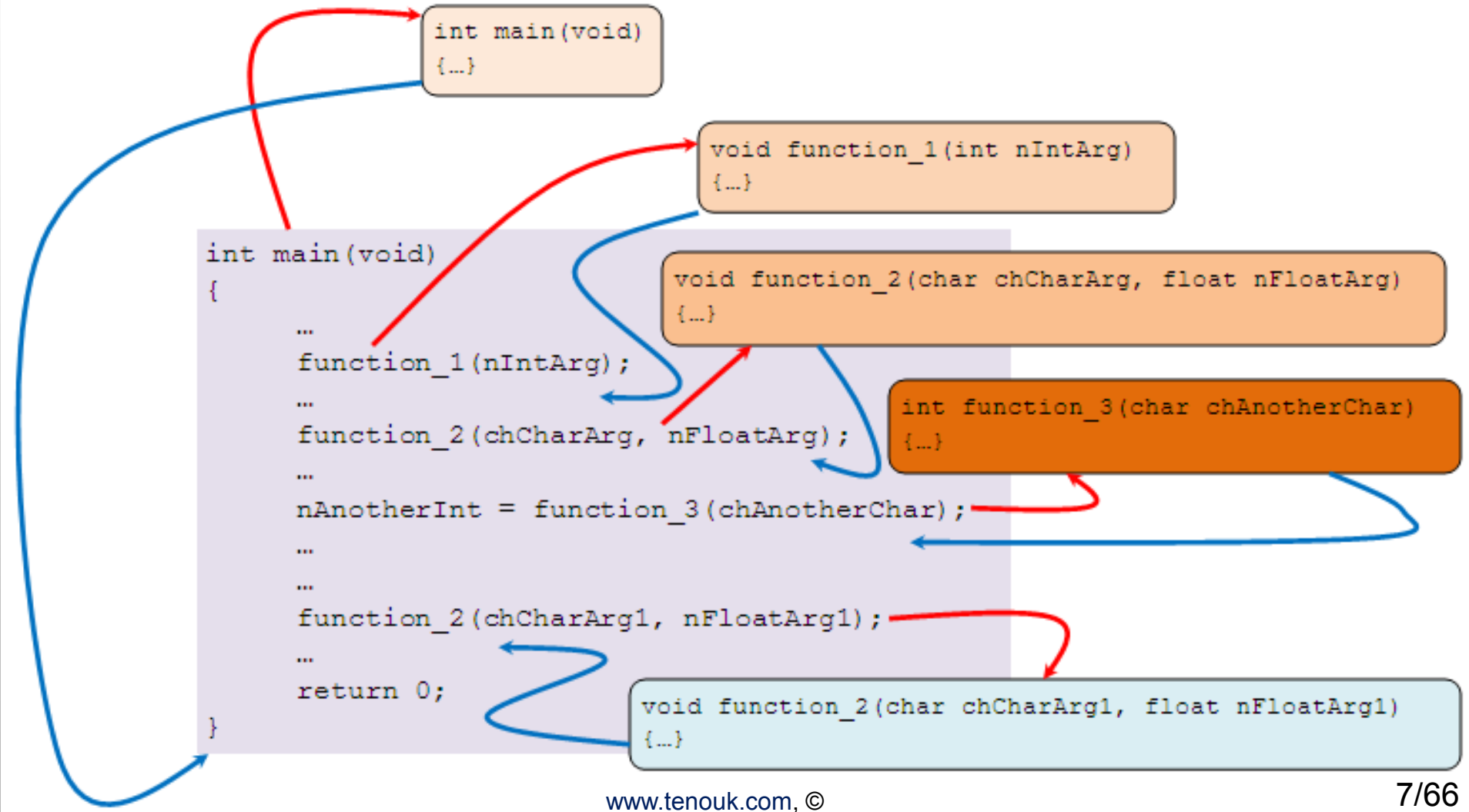
C FUNCTIONS

Function Mechanism

- C program does not execute the statements in a function until the function is called.
- When it is called, the program can send information to the function in the form of one or more arguments although it is not a mandatory.
- Argument is a program data needed by the function to perform its task.
- When the function finished processing, program returns to the same location which called the function.

C FUNCTIONS

- The following figure illustrates function calls (also the memory's stack record activation – construction & destruction.



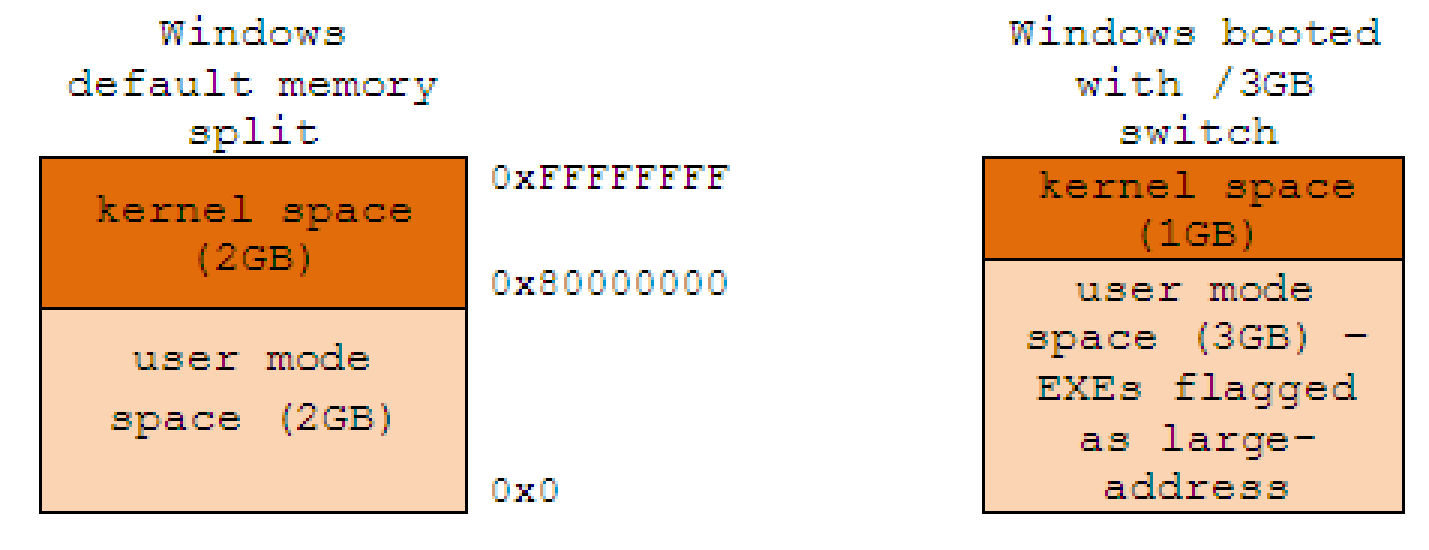
C FUNCTIONS

- Function can be called as many times as needed as shown for `function_2(…)`.
- Can be called in any order provided that it has been declared (as a prototype) and defined.

C FUNCTIONS

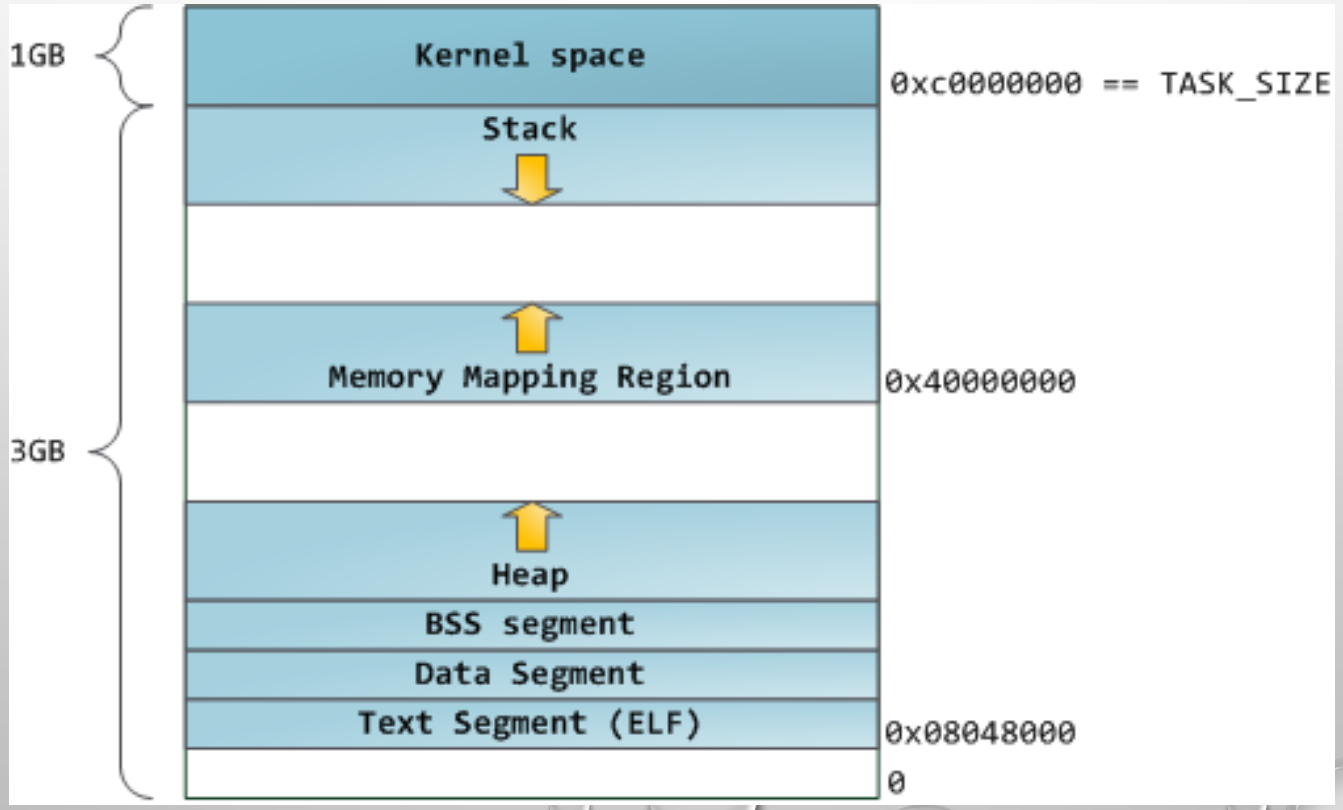
Stack and Heap

- Each process in a multi-tasking OS runs in its own memory area.
- This area is the virtual address space, which in 32-bit mode, is always a 4GB block of memory addresses.
- These virtual addresses are mapped to physical memory by page tables, which are maintained by the OS's kernel and consulted by the processor.
- Each process has its own set of page tables.
- More info at: Anatomy of program in memory.



C FUNCTIONS

- As an example, the standard program's/executable's/binary's segment layout in a Linux process is shown in the following figure.
- The bands in the address space are correspond to memory segments like the heap, stack, and so on.

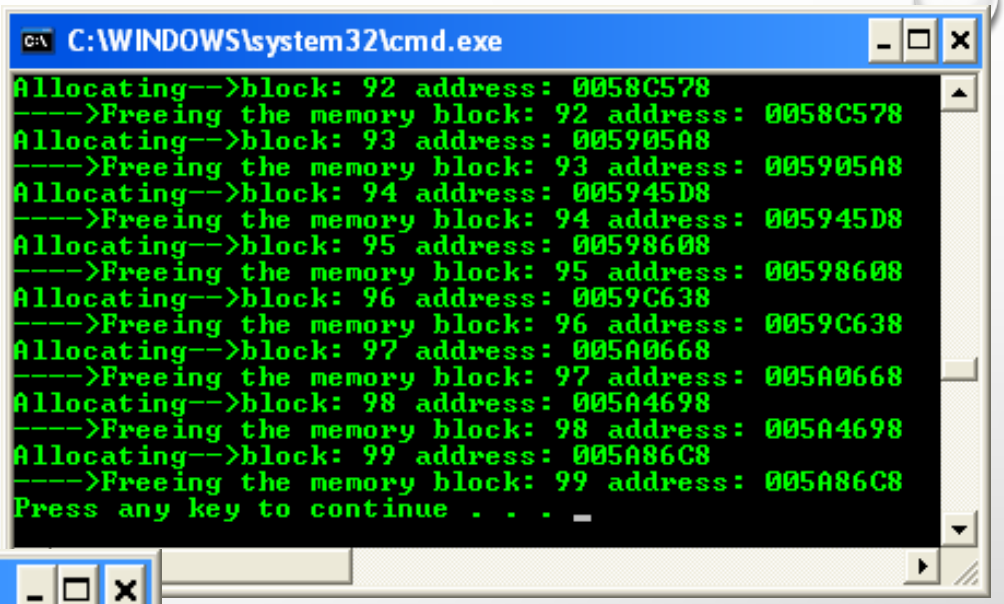


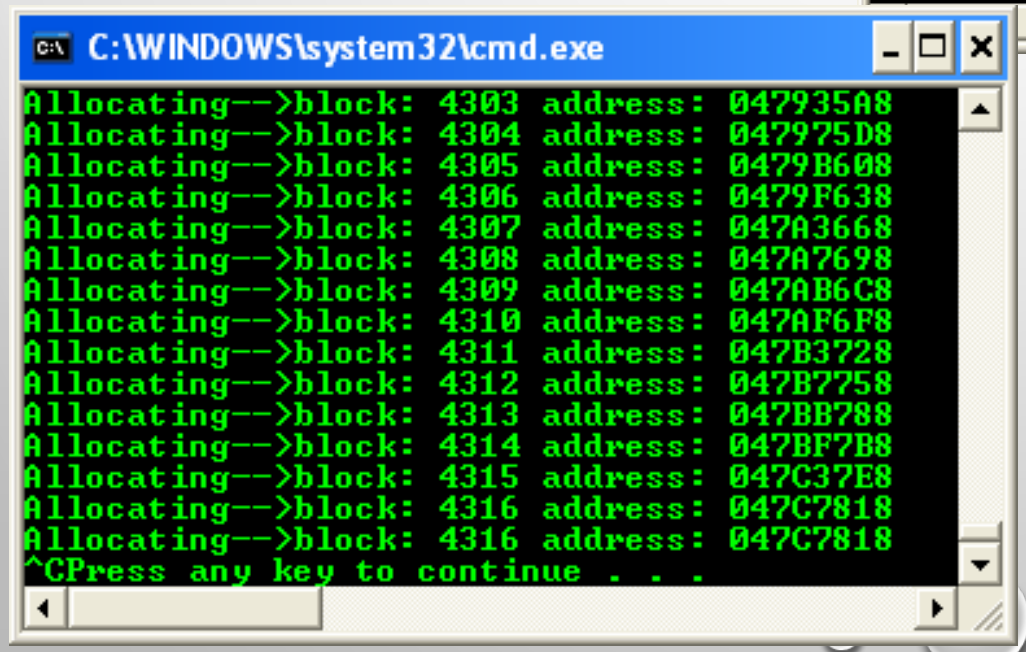
C FUNCTIONS

- Stack and heap are two memory sections in the user mode space.
- Stack will be allocated automatically for function call.
- It grows downward to the lower address.
- It is Last-in First-out (LIFO) mechanism (tally with the assembly language's push and pop instructions)
- Meanwhile, heap will be allocated by demand or request using C memory management functions such as `malloc()`, `memset()`, `realloc()` etc.
- It is dynamic allocation, grows upward to the higher memory address.
- By request means we need to release the allocation manually using C functions such as `free()` and `delete` (if using `new` keyword).
- In a multi-threaded environment each thread will have its own completely independent stack but they will share the heap as needed.

C FUNCTIONS

- malloc() & free() example

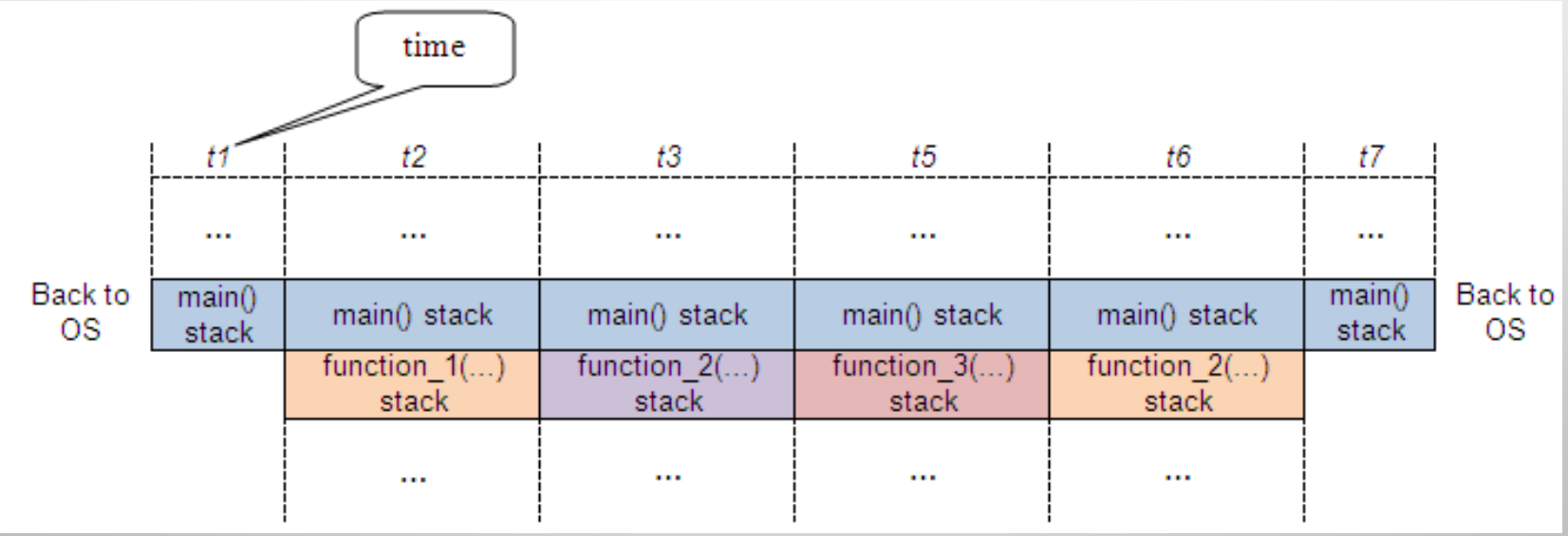




- malloc() without free() example

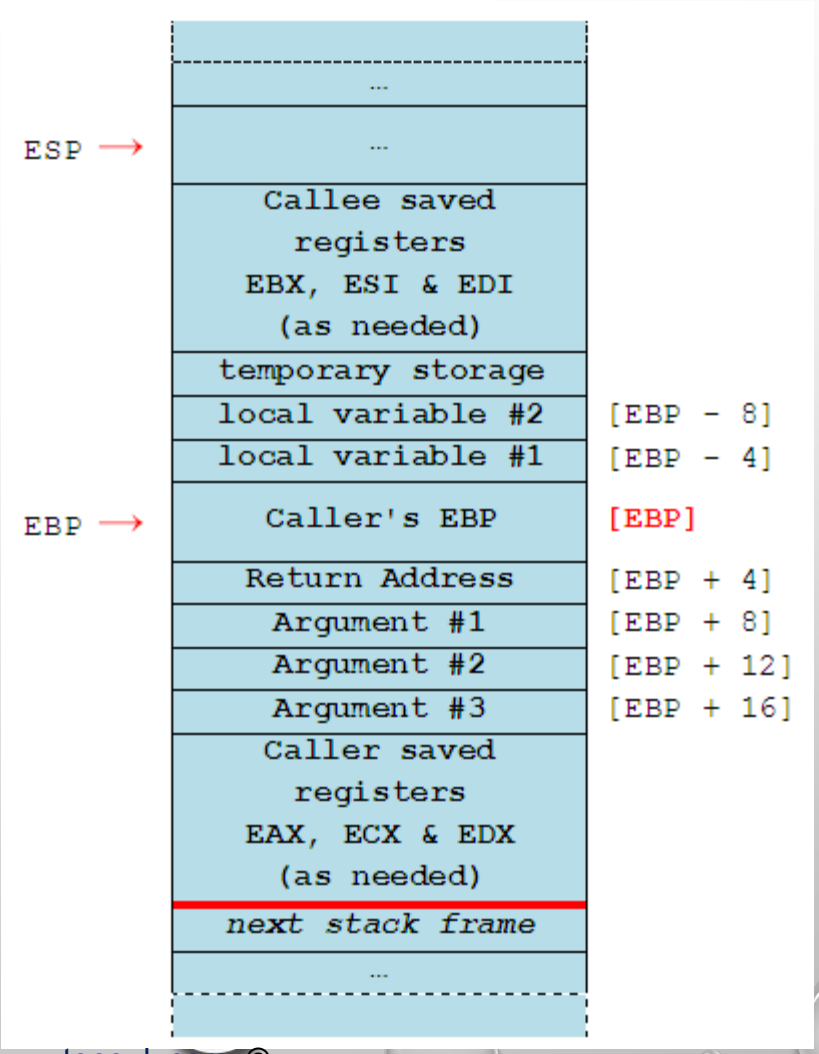
C FUNCTIONS

- By referring the previous function calls example, the stack should be in the equilibrium condition.
- By considering the stack area of memory, the following figure illustrates the condition of stack for function call.
- In this case, `main()` function has 3 function calls: `function_1()`, `function_2()` and `function_3()`.



C FUNCTIONS

- The following is what a typical stack frame might look like.
- Smaller numbered/lower memory addresses are on top.



Legends:

ESP	stack pointer
EBP	base pointer
EAX, EBX, ECX, EDX, ESI, EDI	processors' registers

C FUNCTIONS

- This would be the contents of the stack if we have a function `MyFunct()` with the prototype,

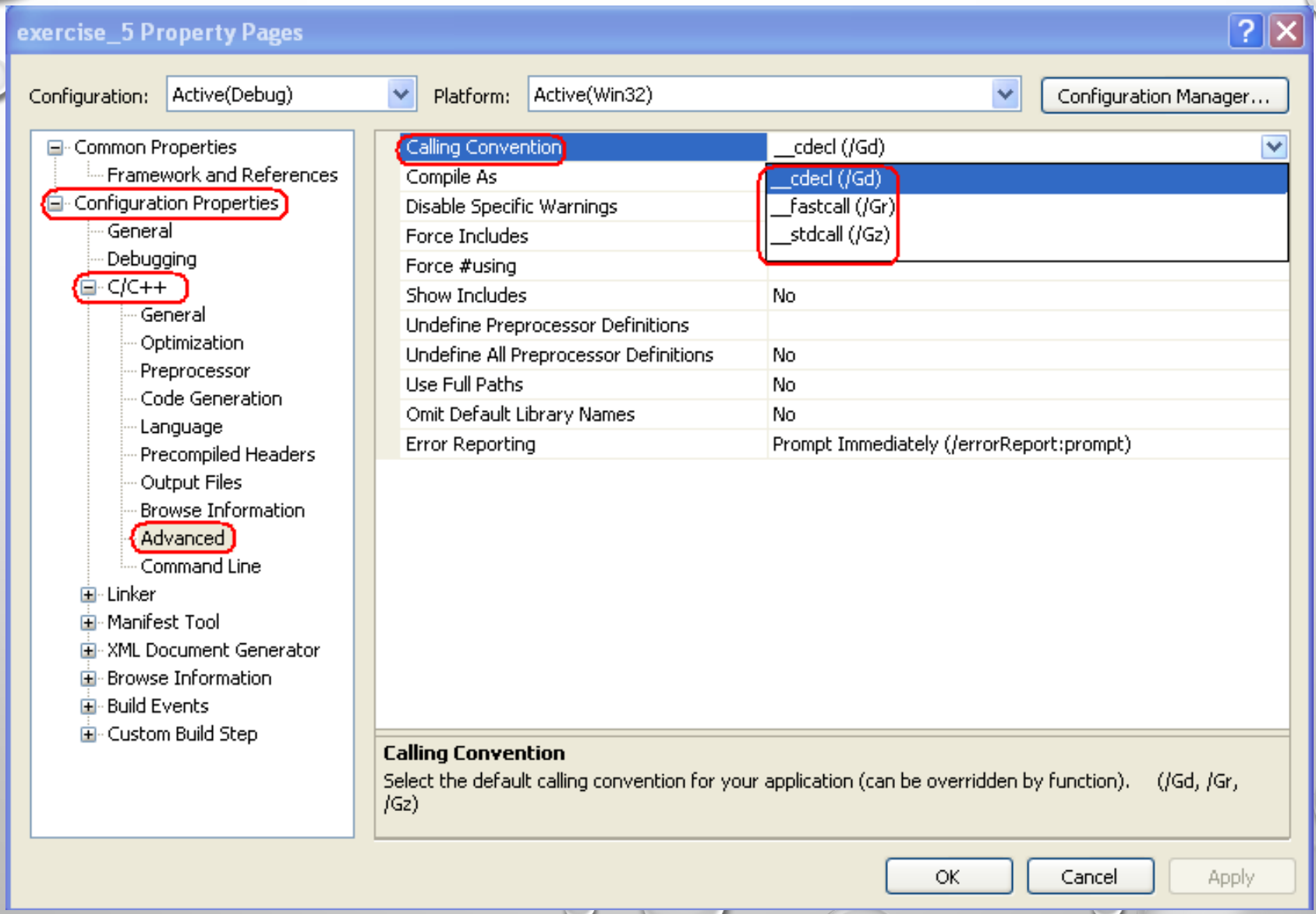
```
int MyFunct(int arg1, int arg2, int arg3) ;
```

- and in this case, `MyFunct()` has two local int variables. (We are assuming here that `sizeof(int)` is 4 bytes).
- The stack would look like this if the `main()` function called `MyFunct()` and control of the program is still inside the function `MyFunct()`.
- `main()` is the "caller" and `MyFunct()` is the "callee".
- The `ESP` register is being used by `MyFunct()` to point to the top of the stack.
- The `EBP` register is acting as a "base pointer".

C FUNCTIONS

- The arguments passed by `main()` to `MyFunct()` and the local variables in `MyFunct()` can all be referenced as an offset from the base pointer.
- The convention used here (`cdecl`) is that the callee is allowed to mess up the values of the `EAX`, `ECX` and `EDX` registers before returning.
- Depend on the calling convention used: `cdecl`, `stdcall` or `fastcall`
- So, if the caller wants to preserve the values of `EAX`, `ECX` and `EDX`, the caller must explicitly save them on the stack before making the subroutine call.
- On the other hand, the callee must restore the values of the `EBX`, `ESI` and `EDI` registers.
- If the callee makes changes to these registers, the callee must save the affected registers on the stack and restore the original values before returning.
- Parameters passed to `MyFunct()` are pushed on the stack.
- The last argument is pushed first so in the end the first argument is on top.
- Local variables declared in `MyFunct()` as well as temporary variables are all stored on the stack.

C FUNCTIONS


exercise_5 Property Pages
Configuration: Active(Debug)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Framework and References
Configuration Properties
General
Debugging
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
Advanced
Command Line
Linker
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Calling Convention
__cdecl (/Gd)
Compile As
Disable Specific Warnings
Force Includes
Force #using
Show Includes
No
Undefine Preprocessor Definitions
Undefine All Preprocessor Definitions
No
Use Full Paths
No
Omit Default Library Names
No
Error Reporting
Prompt Immediately (/errorReport:prompt)
__cdecl (/Gd)
__fastcall (/Gr)
__stdcall (/Gz)
Calling Convention
Select the default calling convention for your application (can be overridden by function). (/Gd, /Gr, /Gz)
OK
Cancel
Apply

C FUNCTIONS

- Return values of 4 bytes or less are stored in the `EAX` register.
- If a return value with more than 4 bytes is needed, then the caller passes an "extra" first argument to the callee.
- This extra argument is address of the location where the return value should be stored. i.e., in C jargon the function call,

```
x = MyFunct(a, b, c);
```

- is transformed into the call,

```
MyFunct(&x, a, b, c);
```

- Note that this only happens for function calls that return more than 4 bytes.

C FUNCTIONS

Function Definition

- Is the actual function body, which contains the code that will be executed as shown below (previous example).

```
int  cube(int fCubeSide)
{
        // local scope (local to this function)
        // only effective in this function 'body'
        int     fCubeVolume;

        // calculate volume
        fCubeVolume  =  fCubeSide * fCubeSide * fCubeSide;
        // return the result to caller
        return     fCubeVolume;
}
```

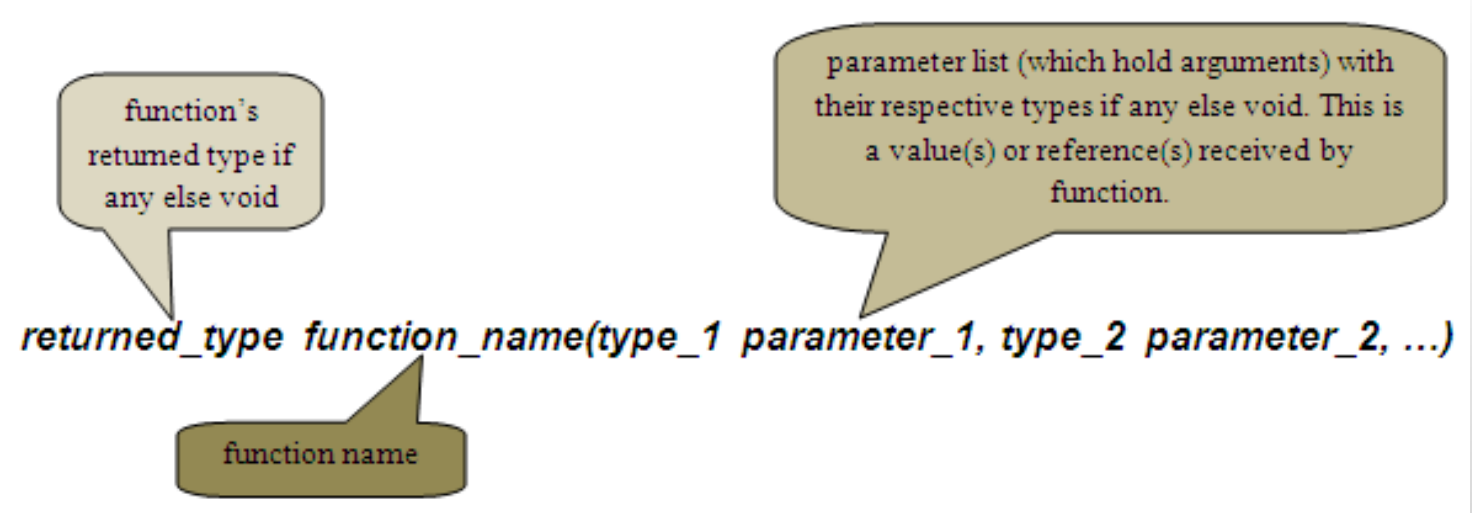
C FUNCTIONS

- First line of a function definition is called the function header, should be identical to the function prototype, except the semicolon.
- Although the argument variable names (`fCubeSide` in this case) were optional in the prototype, they must be included in the function header.
- Function body, containing the statements, which the function will perform, should begin with an opening brace and end with a closing brace.
- If the function returns data type is anything other than `void` (nothing to be returned), a `return` statement should be included, returning a value matching the return data type (`int` in this case).

C FUNCTIONS

The Function header

- The first line of every function definition is called function header. It has 3 components, as shown below,



1. Function return type - Specifies the data type that the function should returns to the caller program. Can be any of C data types: `char`, `float`, `int`, `long`, `double`, pointers etc. If there is no return value, specify a return type of `void`. For example,

```
int     calculate_yield(…)   // returns an int type
float   mark(…)              // returns a float type
void    calculate_interest(…)  // returns nothing
```

C FUNCTIONS

1. *Function name* - Can have any name as long as the rules for C / C++ variable names are followed and must be unique.
2. *Parameter list* - Many functions use arguments, the value passed to the function when it is called. A function needs to know the data type of each argument. Argument type is provided in the function header by the parameter list. Parameter list acts as a placeholder.

C FUNCTIONS

- For each argument that is passed to the function, the parameter list must contain one entry, which specifies the type and the name.
- For example,

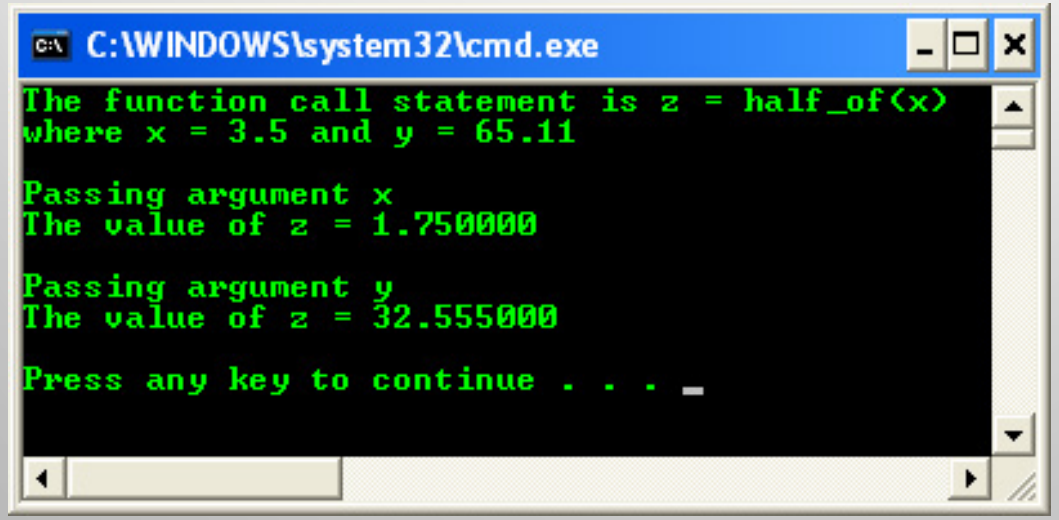
```
void myfunction(int x, float y, char z)
void yourfunction(float myfloat, char mychar)
int  ourfunction(long size)
```

- The first line specifies a function with three arguments: type `int` named `x`, type `float` named `y` and type `char` named `z`.
- Some functions take no arguments, so the parameter list should be `void` or empty such as,

```
long thefunction(void)
void testfunct(void)
int  zerofunct()
```

C FUNCTIONS

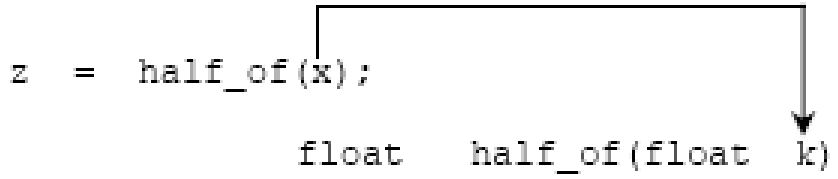
- Parameter is an entry in a function header. It serves as a placeholder for an argument. It is fixed, that is, do not change during execution.
- The argument is an actual value passed to the function by the caller program. Each time a function is called, it can be passed with different arguments.
- A function must be passed with the same number and type of arguments each time it is called, but the argument values can be different.



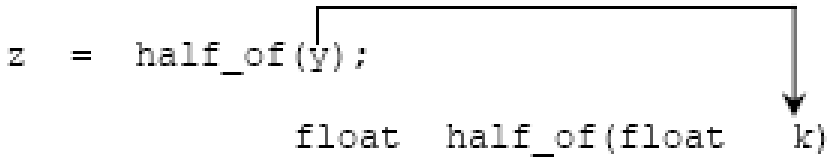
Function example: parameter and argument

C FUNCTIONS

For the first function call:



Then, the second function call:



- Each time a function is called, the different arguments are passed to the function's parameter.
- `z = half_of(y)` and `z = half_of(x)`, each send a different argument to `half_of()` through the `k` parameter.
- The first call send `x`, which is `3.5`, then the second call send `y`, which is `65.11`.
- The value of `x` and `y` are passed (copied) into the parameter `k` of `half_of()`.
- Same effect as copying the values from `x` to `k`, and then `y` to `k`.
- `half_of()` then returns this value after dividing it by `2`.

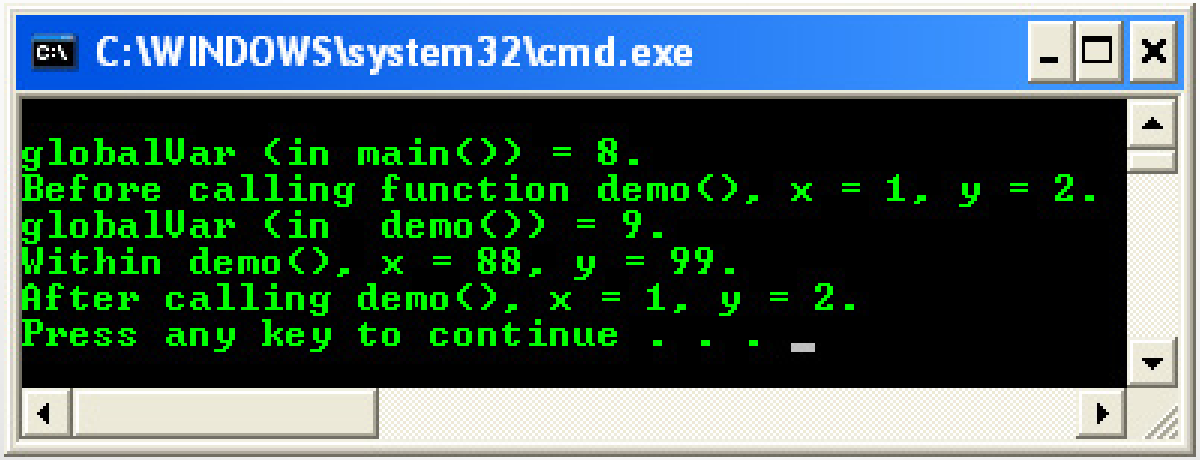
C FUNCTIONS

The Function Body

- Enclosed in curly braces, immediately follows the function header.
- Real work in the program is done here.
- When a function is called execution begins at the start of the function body and terminates (returns to the calling program) when a `return` statement is encountered or when execution reaches the closing braces (`}`).
- Variable declaration can be made within the body of a function.
- Which are called local variables. The scope, that is the visibility and validity of the variables are local.
- Local variables are the variables apply only to that particular function, are distinct from other variables of the same name (if any) declared elsewhere in the program outside the function.
- It is declared, initialized and use like any other variable.
- Outside of any functions, those variables are called global variables.

C FUNCTIONS

- Function program example: local and global variable



- The function parameters are considered to be variable declarations.
- Function prototype normally placed before `main()` and your function definition after `main()` as shown below.
- For C++, the standard said that we must include the prototype but not for C.

C FUNCTIONS

```
#include …

/* function prototype
*/
int funct1(int);

int main()
{
     /* function call
*/
     int y =
funct1(3);
     …
}

/* Function
definition */
int funct1(int x)
{…}
```

But it is OK if we directly declare and define the function before `main()` as shown below.

```
#include …

/* declare and define */
int funct1(int x)
{
  …
}

int main()
{
  /* function call */
  int y = funct1(3);
  …
}
```

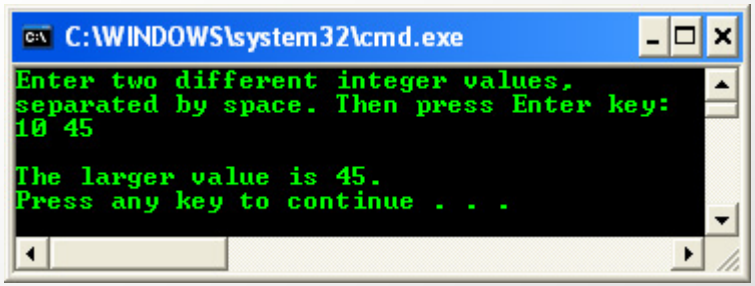
Three rules govern the use of variables in functions:

1. To use a variable in a function, we must declare it in the function header or the function body.
2. For a function to obtain a value from the calling program (caller), the value must be passed as an argument (the actual value).
3. For a calling program (caller) to obtain a value from function, the value must be explicitly returned from the called function (callee).

C FUNCTIONS

The Function Statements

- Any statements can be included within a function, however a function may not contain the definition of another function.
- For examples: if statements, loop, assignments etc are valid statements.



Returning a Value

- A function may or may not return a value.
- If function does not return a value, then the function return type is said to be of type `void`.
- To return a value from a function, use `return` keyword, followed by C expression.
- The value is passed back to the caller.
- The return value must match the return data type.
- A function can contain multiple return statements.

- Program example: multiple return statement

C FUNCTIONS

The Function Prototype

- Must be included for each function that will be defined, (required by Standards for C++ but optional for C) if not directly defined before `main()`.
- In most cases it is recommended to include a function prototype in your C program to avoid ambiguity.
- Identical to the function header, with semicolon (`;`) added at the end.
- Function prototype includes information about the function's return type, name and parameters' list and type.
- The general form of the function prototype is shown below,

```
function_return_type    function_name(type parameter1, type parameter2,…,
    type parameterN)
```

- An example of function prototype,

```
long cube(long);
```

C FUNCTIONS

- Function prototype provides the C compiler the name and arguments of the functions and must appear before the function is used or defined.
- It is a model for a function that will appear later, somewhere in the program.
- From the previous prototype example, 'we' know the function is named `cube`, it requires a variable of the type `long`, and it will return a value of type `long`.
- Then, the compiler can check every time the source code calls the function, verify that the correct number and type of arguments are being passed to the function and check that the return value is returned correctly.
- If mismatch occurs, the compiler generates an error message enabling programmers to trap errors.
- A function prototype need not exactly match the function header.
- The optional parameter names can be different, as long as they are the same data type, number and in the same order.
- But, having the name identical for prototype and the function header makes source code easier to understand.

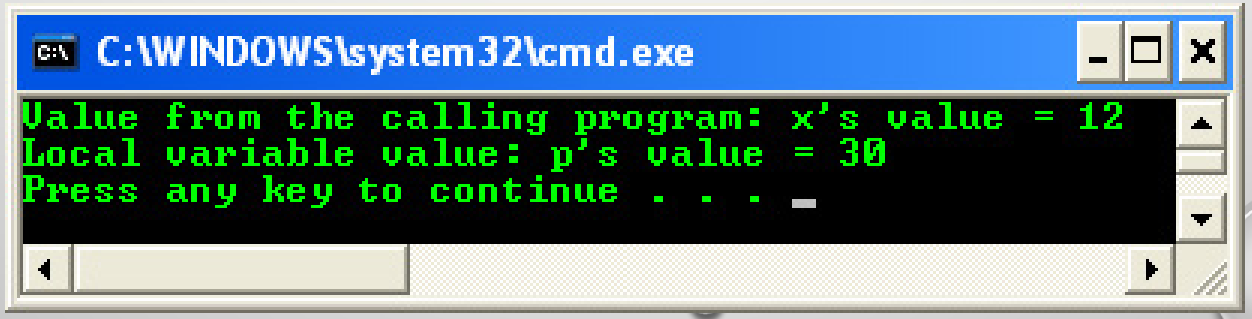
C FUNCTIONS

- Normally placed before the start of `main()` but must be before the function definition.
- Provides the compiler with the description of a function that will be defined at a later point in the program.
- Includes a return type which indicates the type of variable that the function will return.
- And function name, which normally describes what the function does.
- Also contains the variable types of the arguments that will be passed to the function.
- Optionally, it can contain the names of the variables that will be returned by the function.
- A prototype should always end with a semicolon (;).

C FUNCTIONS

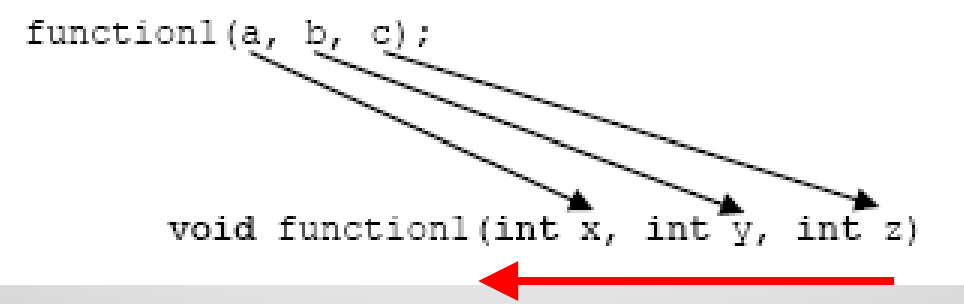
Passing Arguments to a Function

- In order function to interact with another functions or codes, the function passes arguments.
- The called function receives the values passed to it and stores them in its parameters.
- List them in parentheses following the function name.
- Program example: passing arguments



C FUNCTIONS

- The number of arguments and the type of each argument must match the parameters in the function header and prototype.
- If the function takes multiple arguments, the arguments listed in the function call are assigned to the function parameters in order.
- The first argument to the first parameter, the second argument to the second parameter and so on as illustrated below.



- Basically, there are two ways how we can pass something to function parameters,
  1. Passing by value.
  2. Passing by reference using array and pointer.

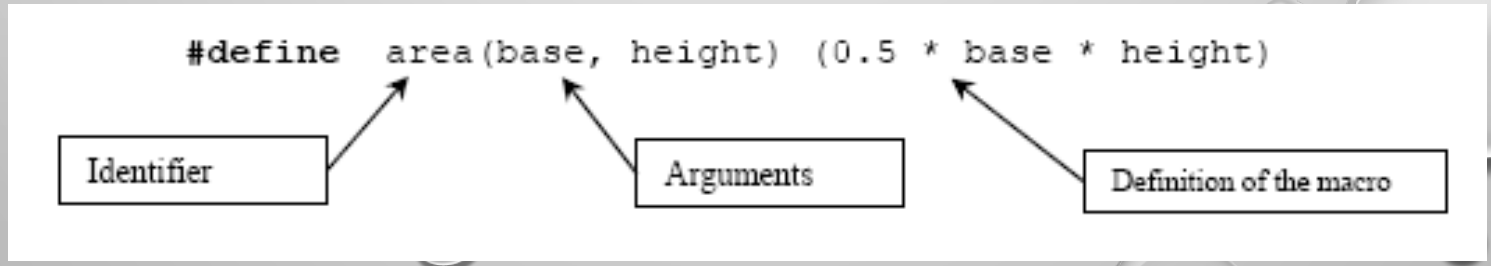
C FUNCTIONS

Macros and Inline Functions

- If the same sequence of steps or instructions is required in several different places in a program, you will normally write a function for the steps and call the function whenever these steps are required. But this involves time overhead.
- Also can place the actual sequence of steps wherever they are needed in the program, but this increase the program size and memory required to store the program. Also need retyping process or copying a block of program.
- Use function if the sequence of steps is long. If small, use macros or inline function, to eliminate the need for retyping and time overhead.

Macros

- Need `#define` compiler directive. For example, to obtain just the area of a triangle, we could create a macro,



C FUNCTIONS

- Then, we can use it anywhere in the program e.g.,

```
printf("\nArea = %f\n", area(4.0, 6.0));
```

- Example for finding an average of 4 numbers (a, b, c and d),

```
#define avg(x, y) (x + y)/2.0
```

- Then in the program we can define something like this,

```
avg1 = avg(avg(a, b), avg(c, d))
```

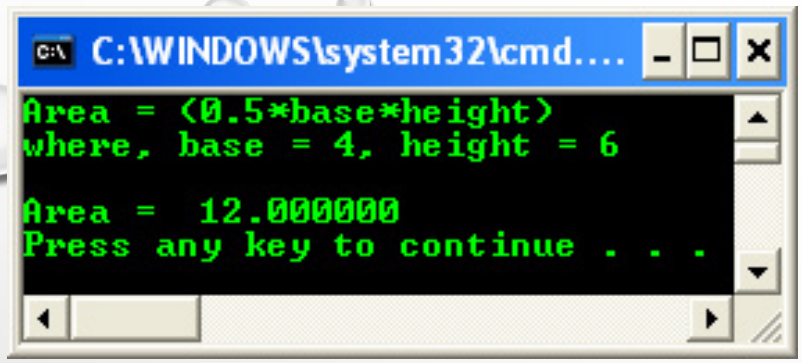
- Doing the substitution,

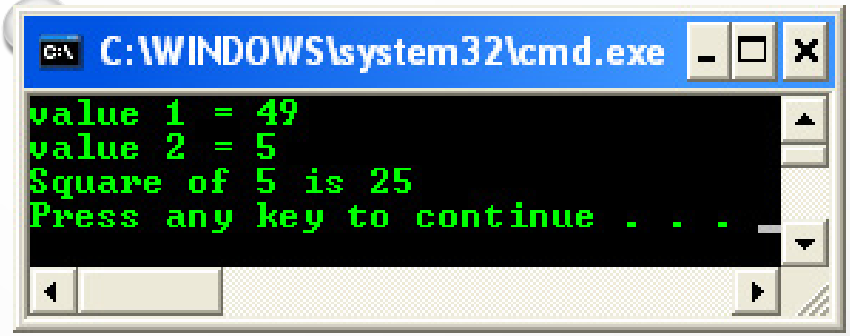
```
avg4 = ((a + b)/2.0 + (c + d)/2.0) / 2.0
```

- The drawback: nesting of macros may result in code that difficult to read.

C FUNCTIONS

Program example: macro





Inline Function

- Is preferred alternative to the macro since it provides most of the features of the macro without its disadvantages.
- Same as macro, the compiler will substitute the code for the inline function wherever the function is called in the program.
- Inline function is a true function whereas a macro is not.
- The best time to use inline functions is when:
  1. There is a time critical function
  2. That is called often
  3. And is respectfully small
- Use keyword `__inline` which is placed before the function.

Program example: inline function

C FUNCTIONS

Header Files and Functions

- Header files contain numerous frequently used functions that programmers can use without having to write codes for them.
- Programmers can also write their own declarations and functions and store them in header files which they can include in any program that may require them (these are called user-defined header file which contains user defined functions).

Standard Header File

- To simplify and reduce program development time and cycle, C provides numerous predefined functions.
- These functions are normally defined for most frequently used routines.
- These functions are stored in what are known as standard library which consist of header files (with extension `.h`, `.hh` etc).
- In the wider scope, each header file stores functions, macros, `enum`, structures (`struct`) , types etc. that are related to a particular application or task.

C FUNCTIONS

- We need to know which functions that are going to use, how to write the syntax to call the functions and which header files to be included in your program.
- Before any function contained in a header file can be used, you have to include the header file in your program. You do this by writing,

```
#include <header_filename.h>
```

- This is called preprocessor directive, normally placed at the top of your program.
- You should be familiar with these preprocessor directives, encountered many times in the program examples previously discussed.

C FUNCTIONS

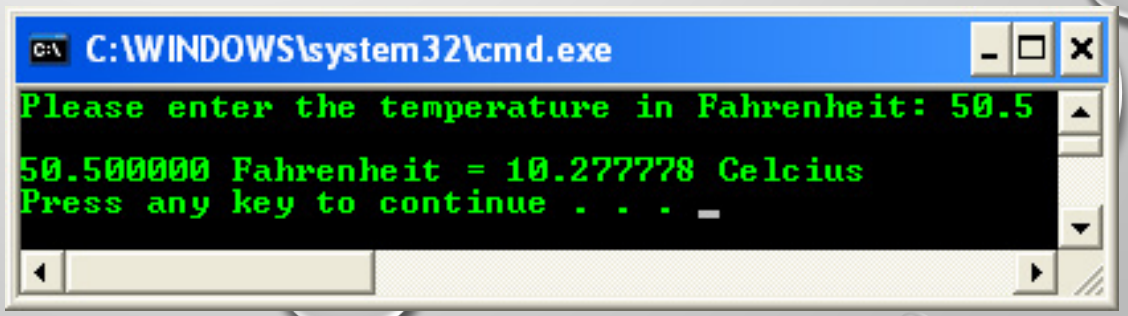
Using Predefined Functions from Header File

- Complete information about the functions and the header file normally provided by the compiler's documentation.
- For your quick reference: C standard library reference.

User-defined Header Files

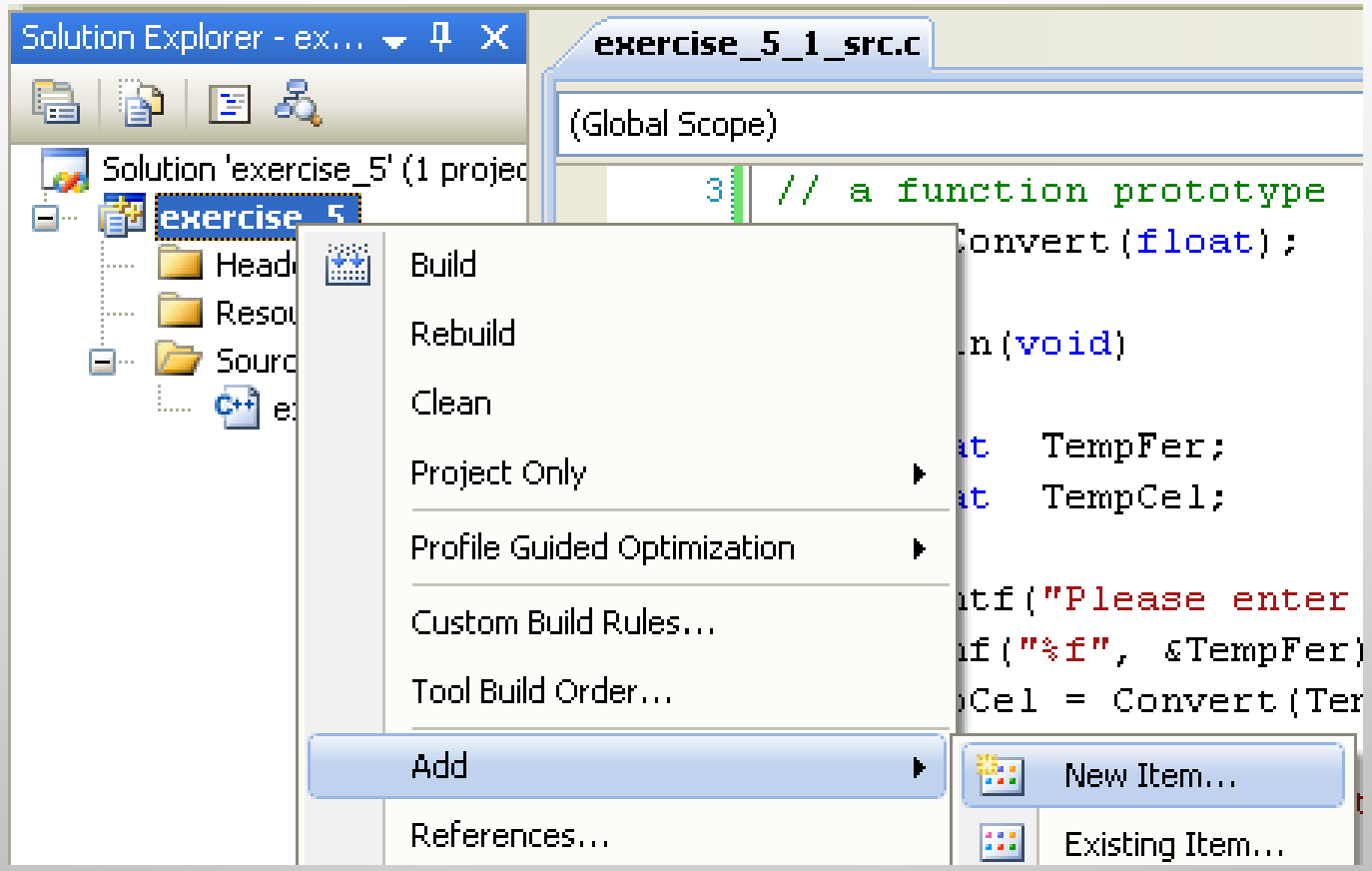
- We can define program segments (including functions) and store them in files.
- Then, we can include these files just like any standard header file in our programs.

Program example: user defined function



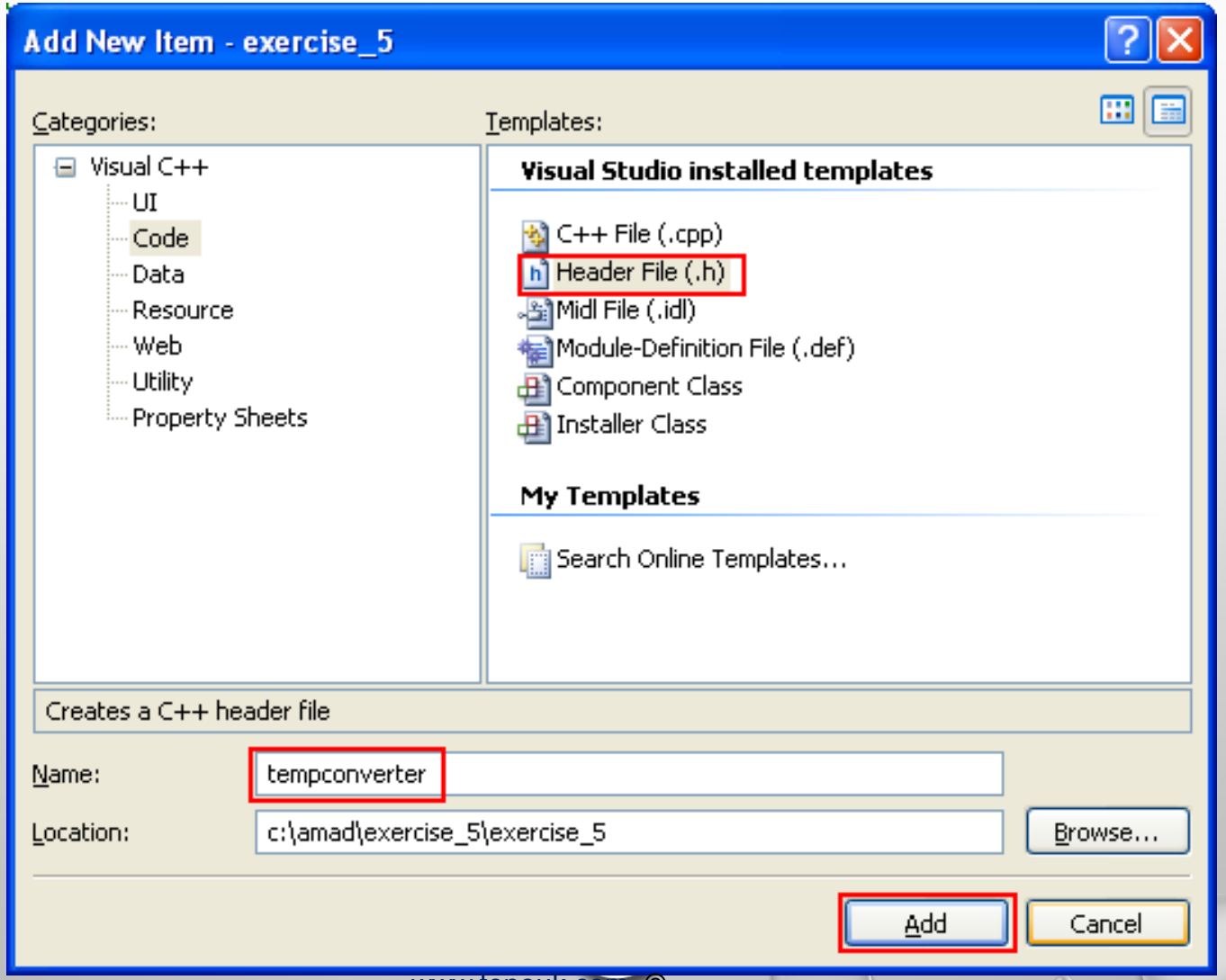
C FUNCTIONS

- Next, let convert the user defined function to header file.
- Step: Add new header file to the existing project.



C FUNCTIONS

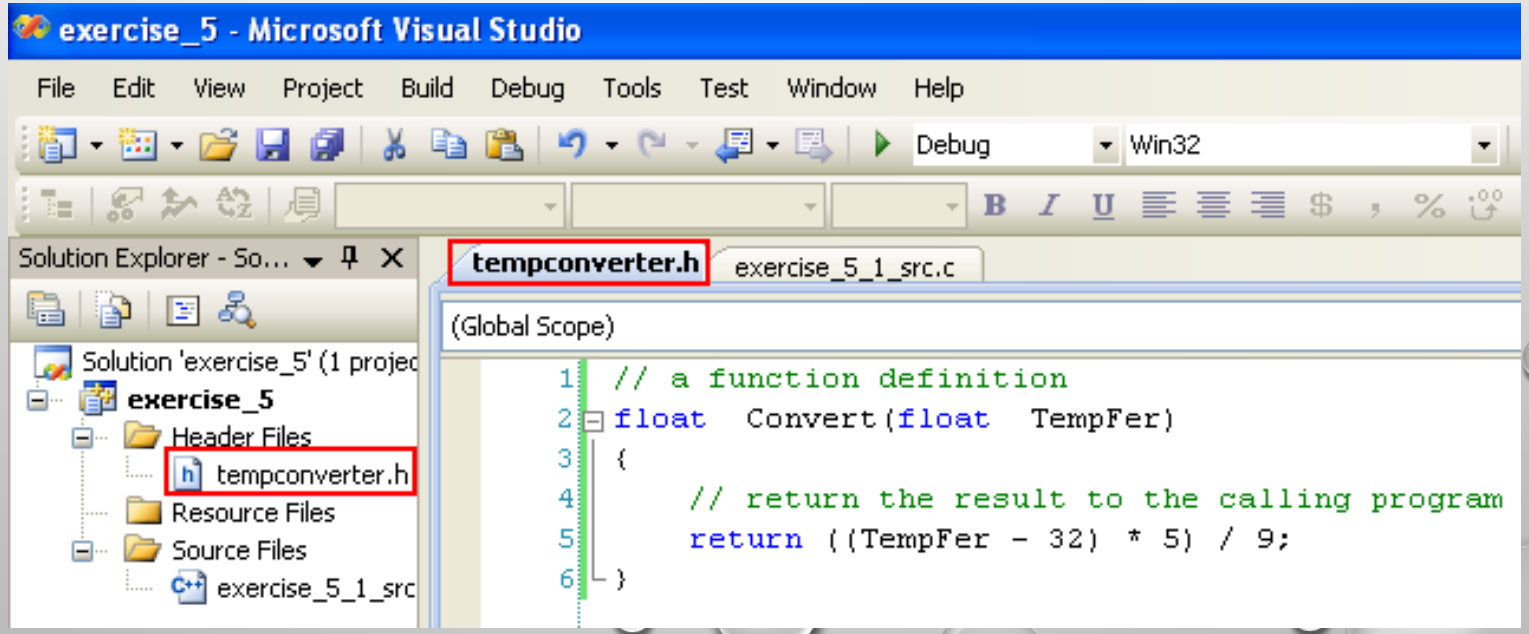
- Step: Put the header file name.



C FUNCTIONS

- Step: Cut the function definition from the source file and paste it into the header file.

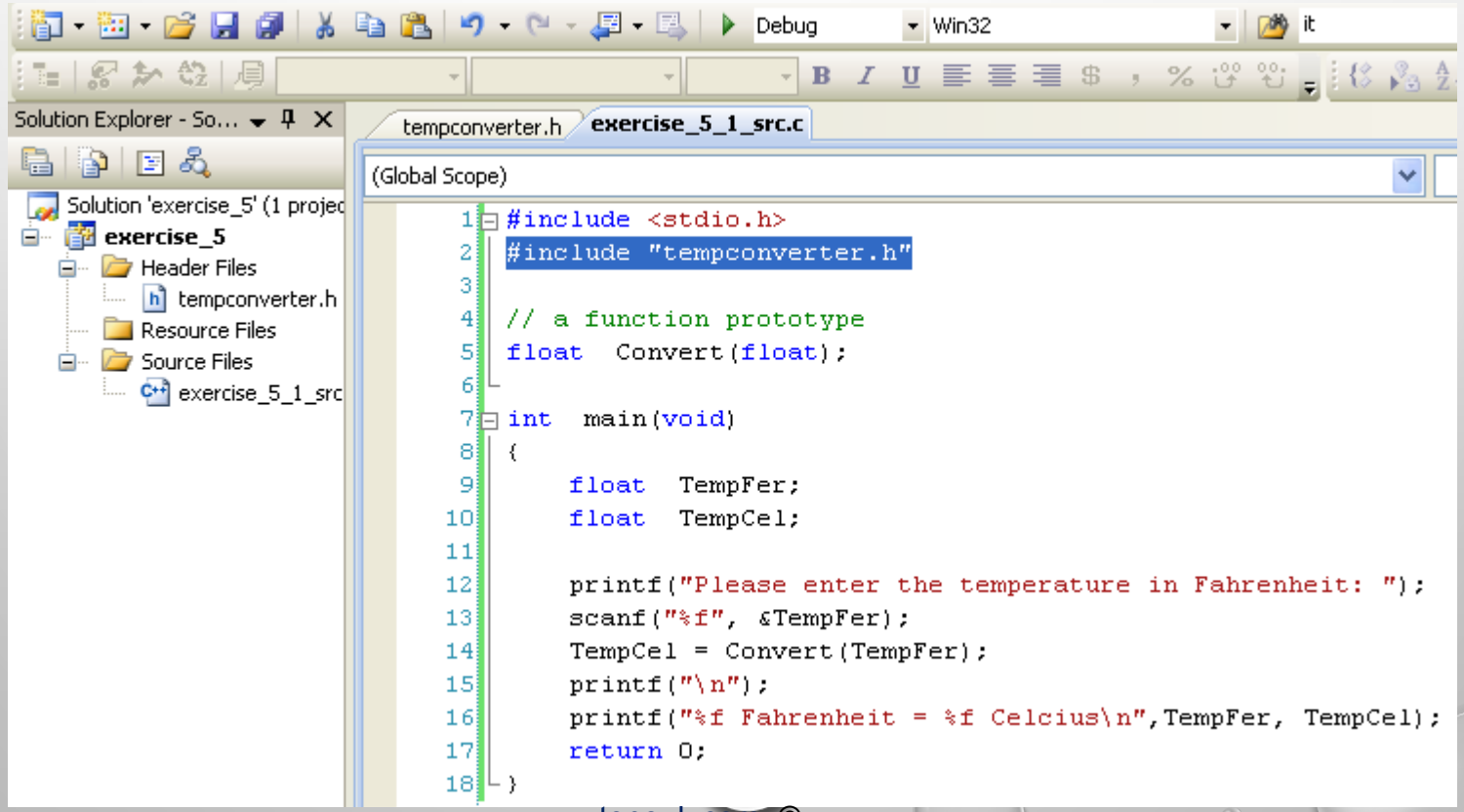
```
// a function definition
float  Convert(float  TempFer)
{
        // return the result to the calling program
        return ((TempFer - 32) * 5) / 9;
}
```



C FUNCTIONS

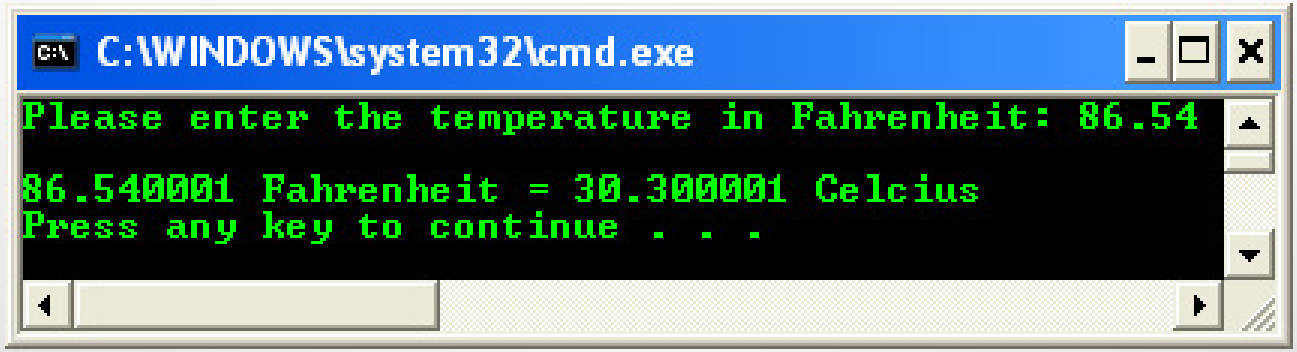
- Step: Next, include the header file at the top, just after the `#include <stdio.h>`.
- Use double quotes because the header file is under the project folder/sub folder instead of the default or VC++ `..\include` sub folder.

```
#include "tempconverter.h"
```



C FUNCTIONS

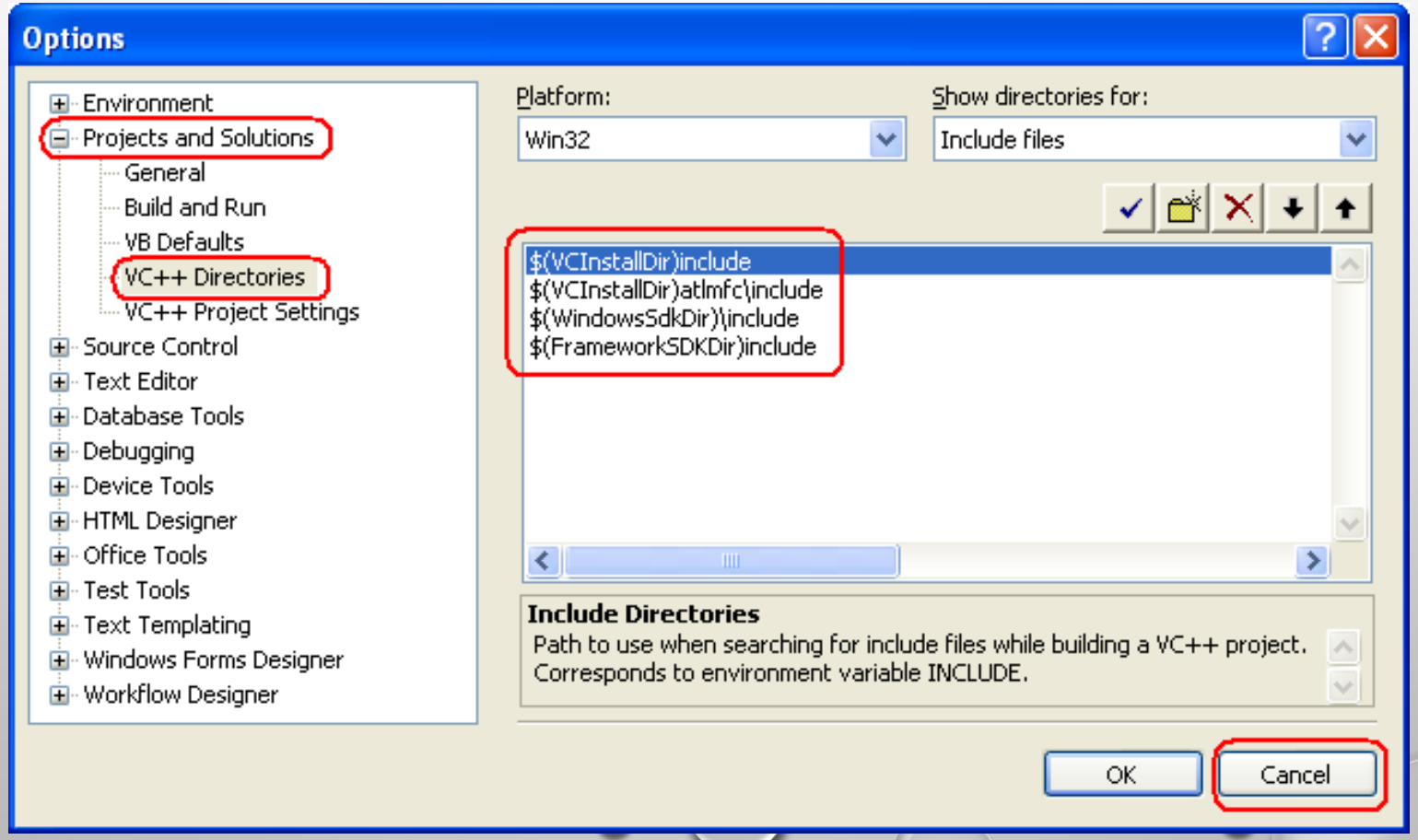
- Step: Re-build and re-run the project.



- Next, we are going to use < > instead of " ".
- To accomplish this, we need to put the header file under the VC++ `include` folder/sub folder.

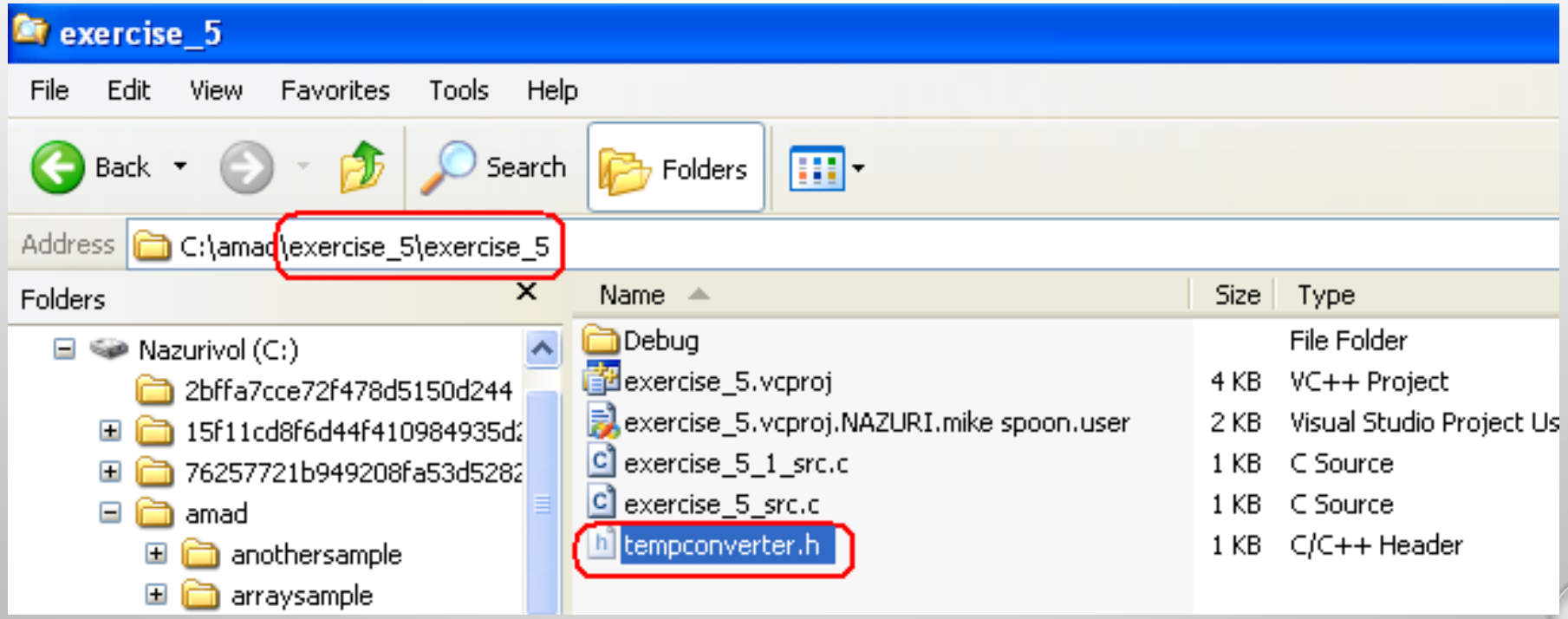
C FUNCTIONS

- The VC++ include sub folder can be found in *VC++ Directories* `Options` settings.
- Step: Select `Tools` menu > Select `Options` sub-menu.

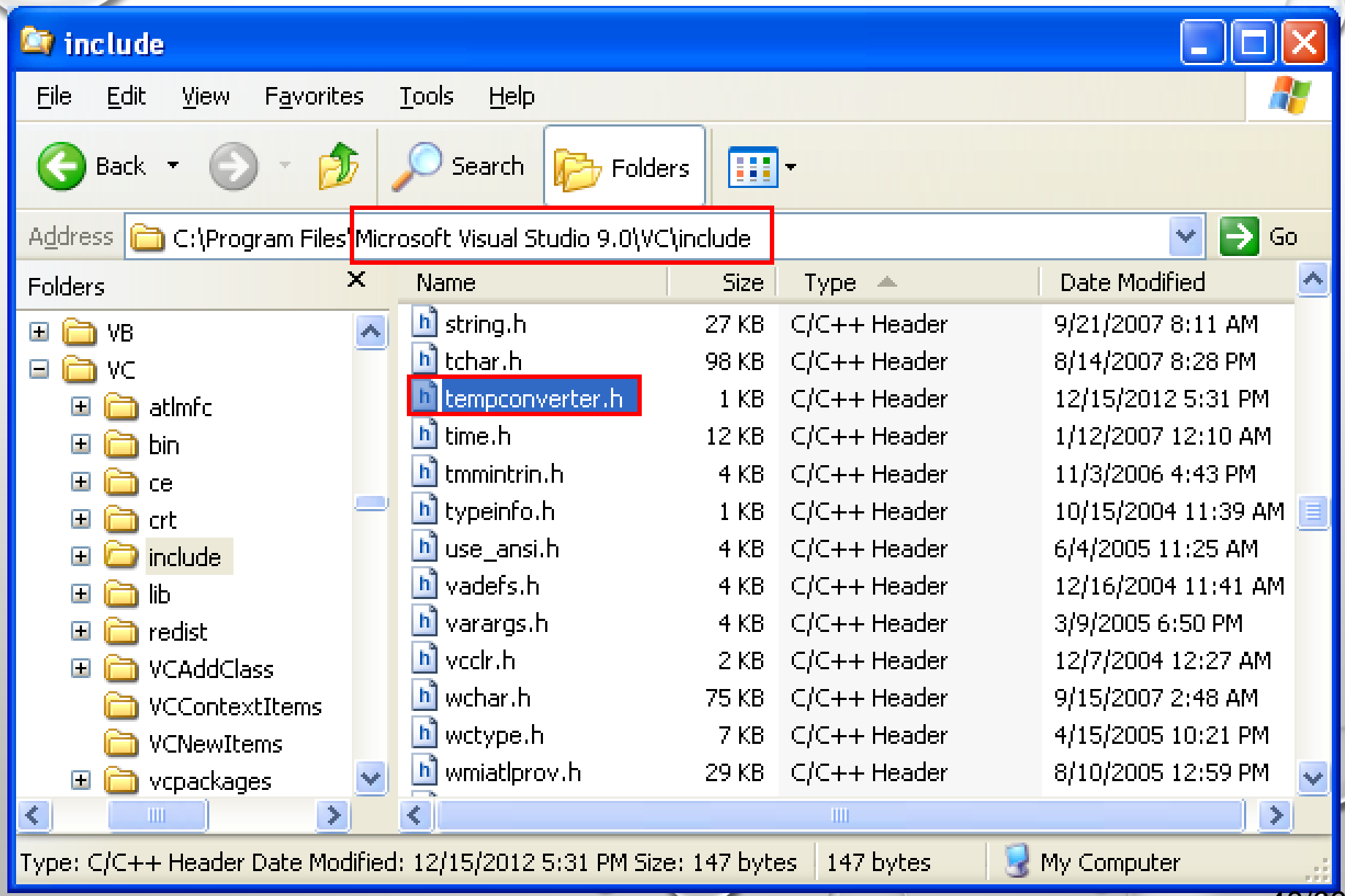


C FUNCTIONS

- Step: When you have found your VC++ include sub folder, copy the header file (`tempconverter.h`) into the VC++ include sub folder.
- Step: The header file can be found under the project folder/sub folder.



C FUNCTIONS


include
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Program Files\Microsoft Visual Studio 9.0\VC\include
Go
Folders
VB
VC
atlmfc
bin
ce
crt
include
lib
redist
VCAddClass
VCContextItems
VCNewItems
vcpackages
Name Size Type Date Modified
string.h 27 KB C/C++ Header 9/21/2007 8:11 AM
tchar.h 98 KB C/C++ Header 8/14/2007 8:28 PM
tempconverter.h 1 KB C/C++ Header 12/15/2012 5:31 PM
time.h 12 KB C/C++ Header 1/12/2007 12:10 AM
tmmintrin.h 4 KB C/C++ Header 11/3/2006 4:43 PM
typeinfo.h 1 KB C/C++ Header 10/15/2004 11:39 AM
use_ansi.h 4 KB C/C++ Header 6/4/2005 11:25 AM
vadefs.h 4 KB C/C++ Header 12/16/2004 11:41 AM
varargs.h 4 KB C/C++ Header 3/9/2005 6:50 PM
vcclr.h 2 KB C/C++ Header 12/7/2004 12:27 AM
wchar.h 75 KB C/C++ Header 9/15/2007 2:48 AM
wctype.h 7 KB C/C++ Header 4/15/2005 10:21 PM
wmiatlprov.h 29 KB C/C++ Header 8/10/2005 12:59 PM
Type: C/C++ Header Date Modified: 12/15/2012 5:31 PM Size: 147 bytes
147 bytes
My Computer

C FUNCTIONS

- Step: Now, change the double quotes (" ") to less-than-greater-than brackets (< >) brackets.

```
#include <tempconverter.h>
```

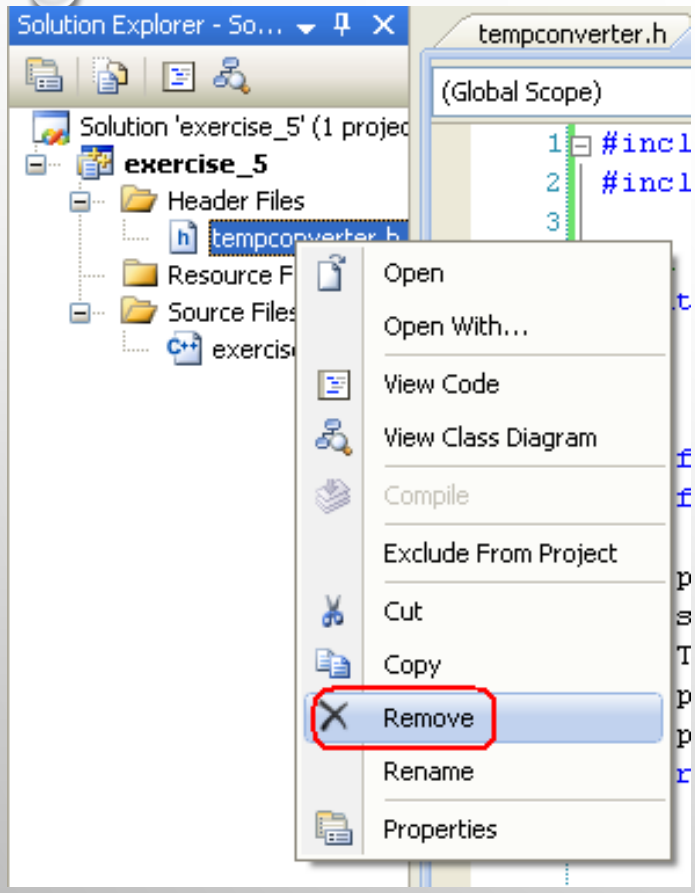
tempconverter.h | exercise_5_1_src.c

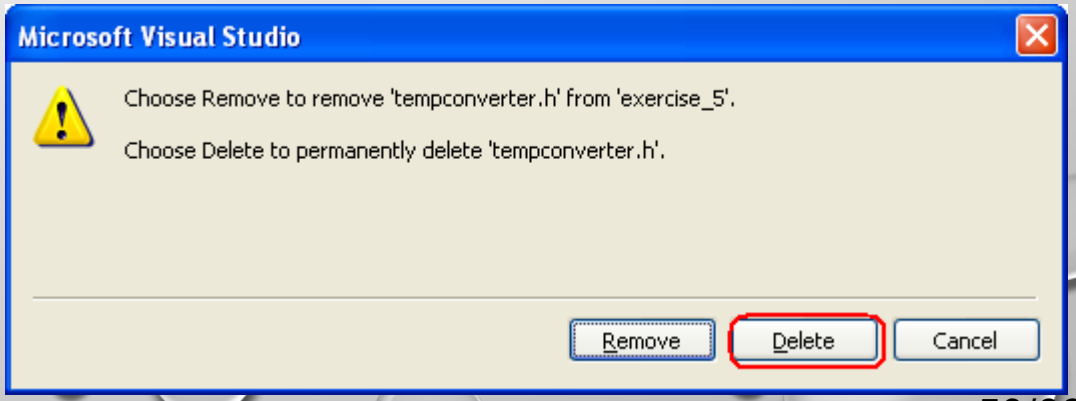
(Global Scope)

```
1  #include <stdio.h>
2  #include <tempconverter.h>
3
4  // a function prototype
5  float  Convert(float);
6
7  int  main(void)
8  {
9      float  TempFer;
10     float  TempCel;
11
12     printf("Please enter the temperature in Fahrenheit: ");
13     scanf("%f", &TempFer);
14     TempCel = Convert(TempFer);
15     printf("\n");
16     printf("%f Fahrenheit = %f Celcius\n",TempFer, TempCel);
17     return 0;
18 }
```

C FUNCTIONS

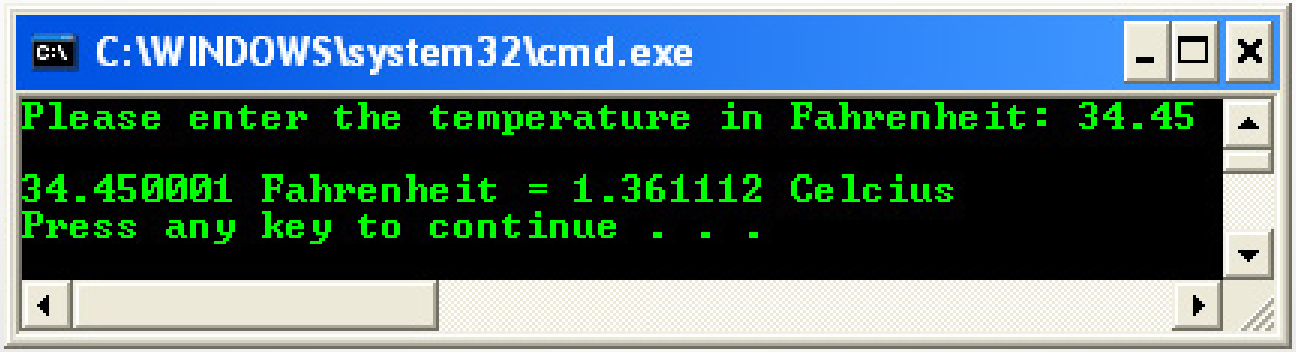
- Step: Then, we can delete the header file in the project.





C FUNCTIONS

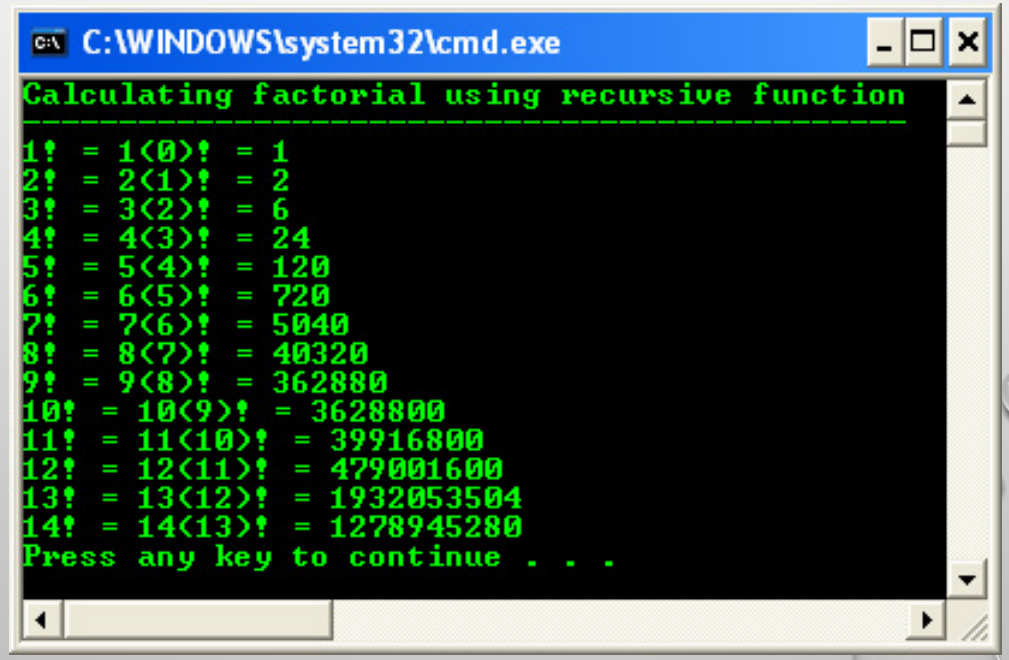
- Step: Re-build and re-run the project.



C FUNCTIONS

Recursive Function

- We cannot define function within function, but we can call the same function within that function.
- A recursive function is a function that calls itself either directly or indirectly through another function.
- Classic example for recursive function is factorial, which used in mathematics.
- Program example: recursive function



C FUNCTIONS

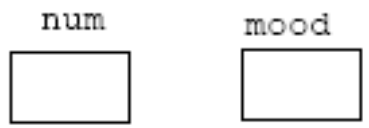
Passing an array to a function

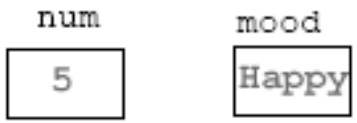
```
#include <stdio.h>

// function prototype
void Wish(int, char[ ]);

void main(void)
{
   Wish(5, "Happy");
}
```

```
// Function definition
void Wish(int num, char mood[])
{
   int i;
   for(i = 1; i <= num; i = i + 1)
         printf("I wish I'm %s\n", mood);
}
```

What are the output and the content of `num` & `mood` variables after program execution was completed?

```
I wish I'm Happy
I wish I'm Happy
I wish I'm Happy
I wish I'm Happy
I wish I'm Happy
Press any key to continue . . .
```





C FUNCTIONS

1. What are the two values passed to `Wish()`?
2. In `Wish()`, what becomes the value of `num`? What becomes the value of `mood`? Write them in the provided boxes.
3. Notice that in both the prototype and the definition of `Wish()`, there is no number in the brackets, giving the number of cells of `mood[ ]`. Can you think why omitting this number makes the function more flexible for use in other instances?
4. How can we tell that `"Happy"` is a character string? How can we tell that `mood[ ]` should also be a character string?
5. If `Wish()` were called from `main()` with this statement, `Wish(3, "Excited");` , then what would be the output?

1. An integer 5 and a string "Happy".
2. num is an integer of 5 and mood is a string of "Happy".
3. This unsized array make the system decide the actual size needed for storage.
4. "Happy" is a character string because it is enclosed in the double quotes and an array mood[ ] has been declared as a char type.
5. Only the first 3 alphabets from "**Exc**ited" will be displayed that is "I wish I'm Exc".

C FUNCTIONS

```
#include <stdio.h>

void Rusted(char[ ]);

void main(void)
{
      // all work done in function Rusted()...
      Rusted("Test Test");
      printf("\n");
}

void Rusted(char x[ ])
{
      int j;
      printf("Enter an integer: ");
      scanf_s("%d", &j);
      for(; j != 0; --j)
            printf("In Rusted(), x = %s\n", x);
}
```

Build this program, show the output & what it do?

```
Enter an integer: 4
In Rusted(), x = Test Test
In Rusted(), x = Test Test
In Rusted(), x = Test Test
In Rusted(), x = Test Test

Press any key to continue . .
```

A function call from `main()` that passes a character string and callee will print the number of character string based on the user input.

C FUNCTIONS

Function, Array and Pointers

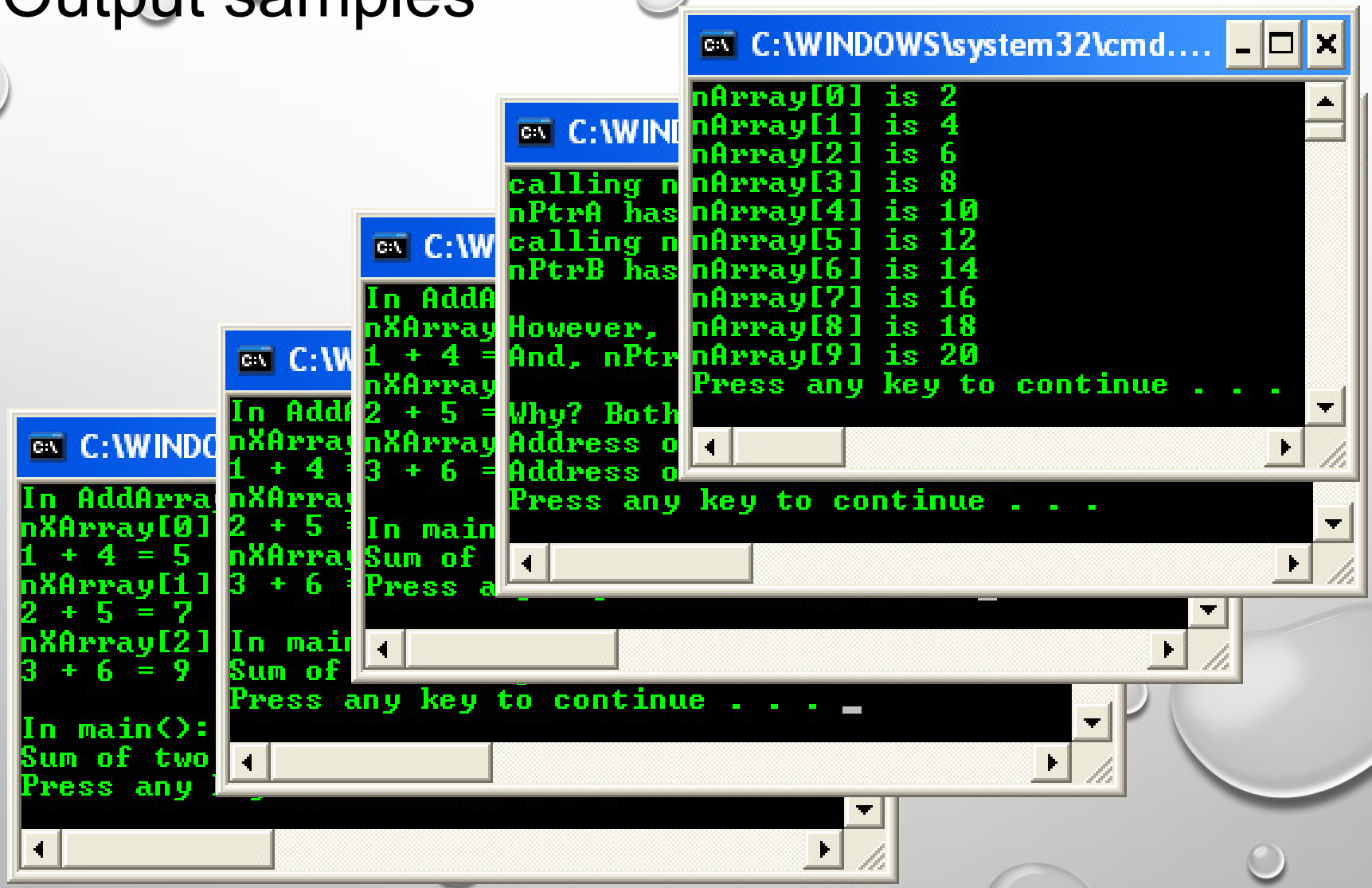
- Functions in C cannot return array types however they can return pointers to arrays or a reference.
- When you pass in the array, you're only passing in a pointer. So when you modify the array's data, you're actually modifying the data that the pointer is pointing at.
- Functions shall not have a return type of type array or function, although they may have a return type of type pointer or reference to such things.
- So, there shall be no arrays of functions, although there can be arrays of pointers to functions.
- Since arrays are not objects, you can't pass them in and out of functions.
- Pointers are objects, so you can pass a pointer to the first element of an object.

C FUNCTIONS

- Functions (including `main()`) are constructed automatically on the 'stack' memory.
- When functions return, the stack will be destroyed/rewound.
- Hence, local variables, including array, will be destroyed, leaving a garbage.
- Program example: passing arrays to a function using array notation.
- Program example: passing arrays to a function using pointer notation.
- Program example: passing arrays to a function & returning a pointer to the first element of the array + global variable.
- Program example: passing arrays to a function & returning a pointer to the first element of the array + `static` keyword.
- Program example: passing arrays to a function using pointer & return the array's content.

C FUNCTIONS

Output samples



C FUNCTIONS

Function and Pointers (Function Pointers)

- We can use pointers to point to C functions because C function have its memory address.
- When we can point to it, then it is another way to invoke it.
- Function pointers are pointer variables which point to functions.
- Function pointers can be declared, assigned values and then used to access the functions they point to.
- The following is an example of function pointer declaration,

```
int   (*functptr)();
```

- Here, `functptr` is declared as a pointer to a function that returns `int` data type.

C FUNCTIONS

- It is a de-referenced value of `functptr`, that is `(*funptr)` followed by `()` which indicates a function, which returns an integer data type.
- The parentheses are essential in the declarations because of the operators' precedence.
- The declaration without the parentheses as the following,

```
int * functptr();
```

- Will declare a function `functptr` that returns an integer pointer that is not our intention in this case.
- In C, the name of a function, which used in an expression by itself, is a pointer to that function.
- For example, if a function, `testfunct()` is declared as follows,

```
int  testfunct(int  xIntArg);
```

C FUNCTIONS

- The name of this function, `testfunct` is a pointer to that function.
- Then, we can assign the function name to a pointer variable `functptr`, something like this:

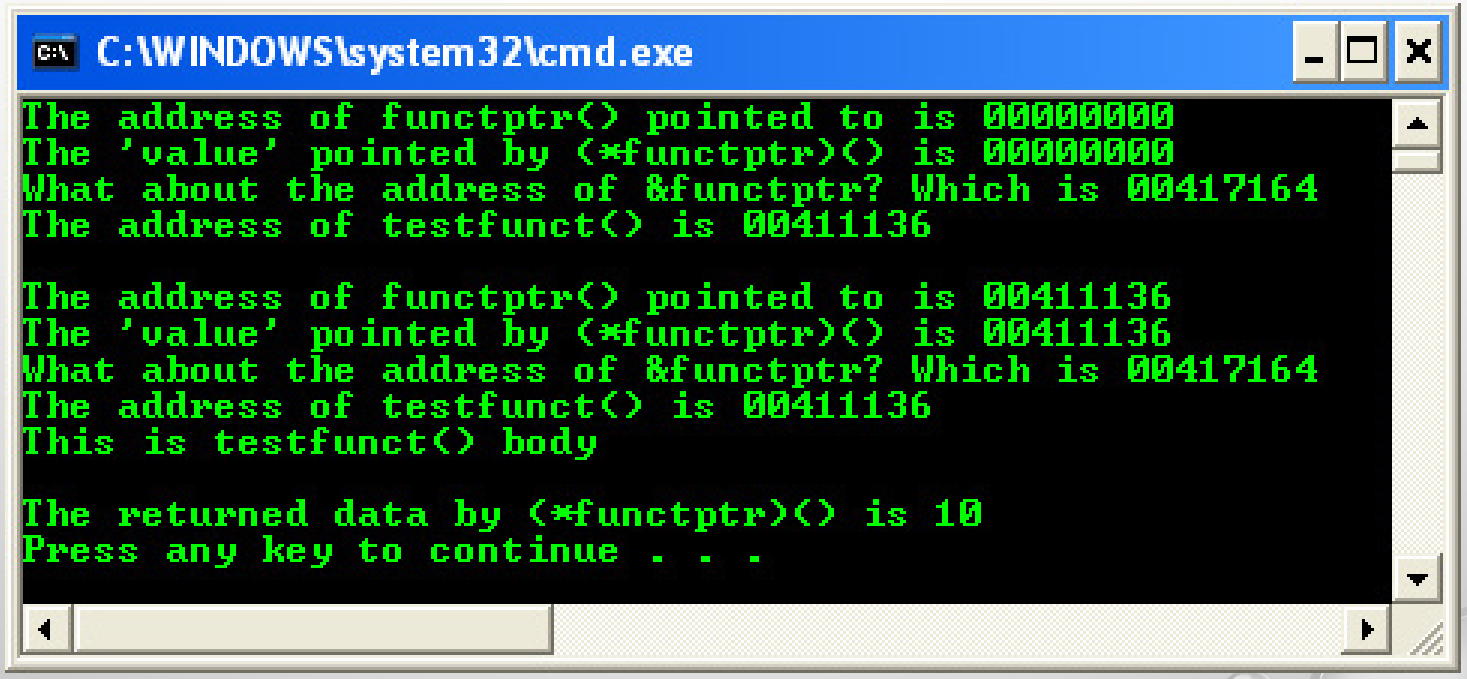
```
functptr = testfunct;
```

- The function can now be accessed or called, by dereferencing the function pointer,

```
/* calls testfunct() with xIntArg as an argument
   then assign the returned value to nRetVal */
nRetVal = (*funptr)( xIntArg);
```

- Program example: function pointers

C FUNCTIONS


C:\WINDOWS\system32\cmd.exe
The address of functptr() pointed to is 00000000
The 'value' pointed by (*functptr)() is 00000000
What about the address of &functptr? Which is 00417164
The address of testfunct() is 00411136
The address of functptr() pointed to is 00411136
The 'value' pointed by (*functptr)() is 00411136
What about the address of &functptr? Which is 00417164
The address of testfunct() is 00411136
This is testfunct() body
The returned data by (*functptr)() is 10
Press any key to continue . . .

C FUNCTIONS

- Function pointers can be passed as parameters in function calls and can be returned as function values.
- It's common to use `typedef` with complex types such as function pointers to simplify the syntax (typing).
- For example, after defining,

```
typedef  int  (*functptr)();
```

- The identifier `functptr` is now a synonym for the type of 'a pointer to a function which takes no arguments and returning int type'.
- Then declaring pointers such as `pTestVar` as shown below, considerably simpler,

```
functptr  pTestVar;
```

- Another example, you can use this type in a `sizeof()` expression or as a function parameter as shown below,

```
/* get the size of a function pointer */
unsigned    pPtrSize = sizeof (int (*functptr)());
/* used as a function parameter */
void    signal(int (*functptr)());
```

C FUNCTIONS

- The following table summarizes the various ways of using functions.
- We can categorize functions by whether or not arguments are passed to them.
- Or we can categorize them by whether or not values are received from them.
- Intersecting these two methods of categorizing functions, we come up with four basic methods of writing and using functions.

C FUNCTIONS

	Do not pass argument	Do pass arguments

No return	`void main(void)`
	`{`
	`  TestFunct();`
	`  ...`
	`}`
	
	`void TestFunct(void)`
	`{`
	`  // receive nothing`
	`  // and nothing to be`
	`  // returned`
	`}`
		`void main(void)`
		`{`
		`  TestFunct(123);`
		`  ...`
		`}`
		
		`void TestFunct(int i)`
		`{`
		`  // receive something and`
		`  // the received/passed`
		`  // value just`
		`  // used here. Nothing`
		`  // to be returned.`
		`}`

With a return	`void main(void)`
	`{`
	`  x = TestFunct();`
	`  ...`
	`}`
	
	`int TestFunct(void)`
	`{`
	`  // received/passed`
	`  // nothing but need to`
	`  // return something`
	`  return 123;`
	`}`
		`void main(void)`
		`{`
		`  x = TestFunct(123);`
		`  ...`
		`}`
		
		`int TestFunct(int x)`
		`{`
		`  // received/passed something`
		`  // and need to return something`
		`  return (x + x);`
		`}`

END of C FUNCTIONS

More exercises can be found at:
tenouk's C lab worksheet