C

FUNCTIONS

FUNCTIONS

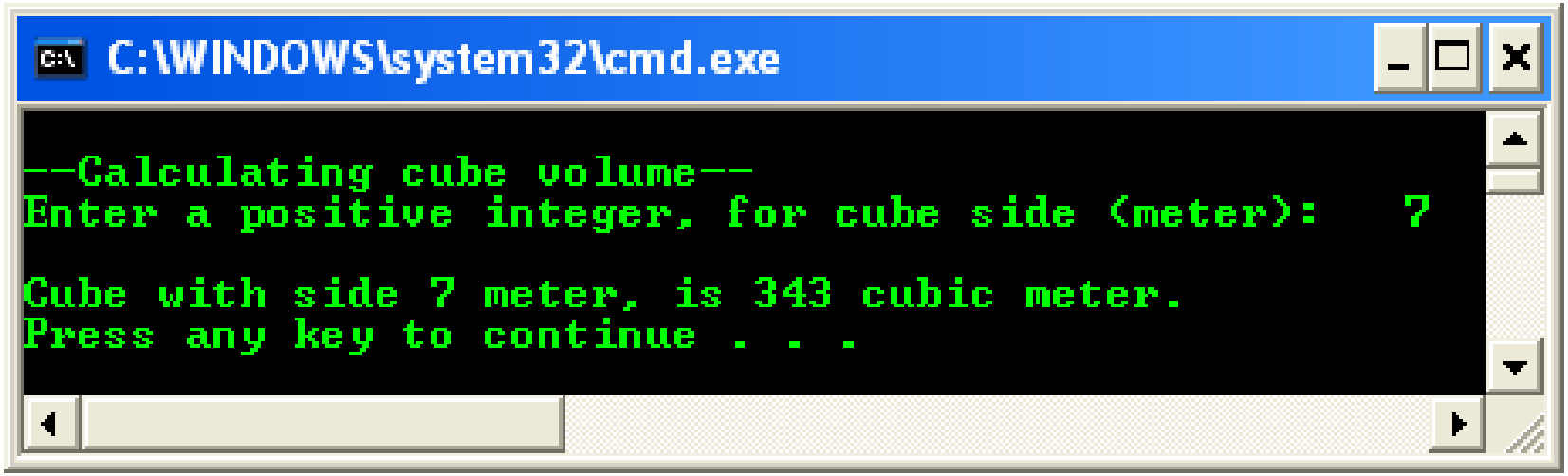
Some definition: A function is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program or/and receives values(s) from the calling program.

- 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.

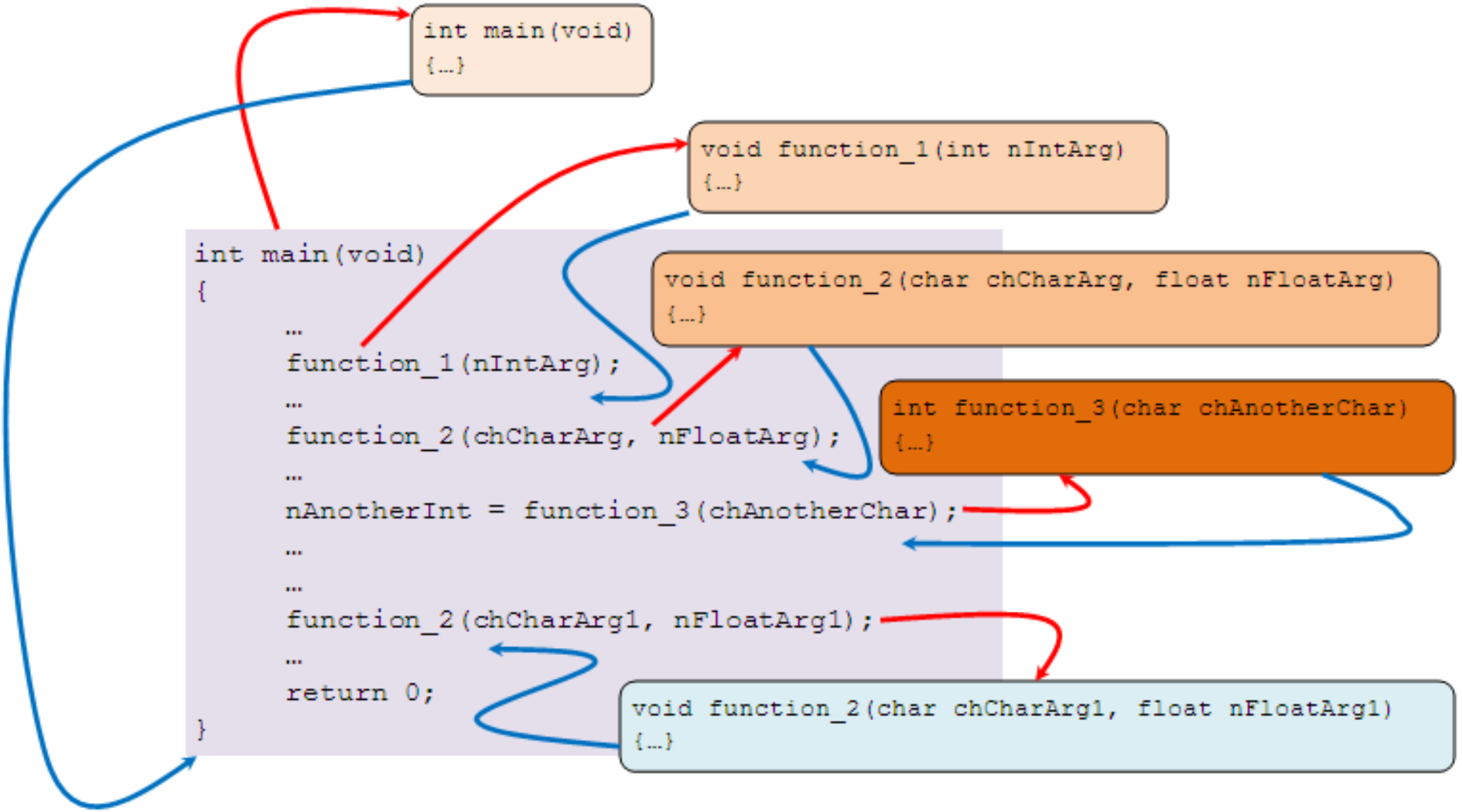
C FUNCTIONS

- 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 refer
5. *May return a value to the calling program* – the called function may pass something back to the calling program.

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

- 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

- 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;
}
```

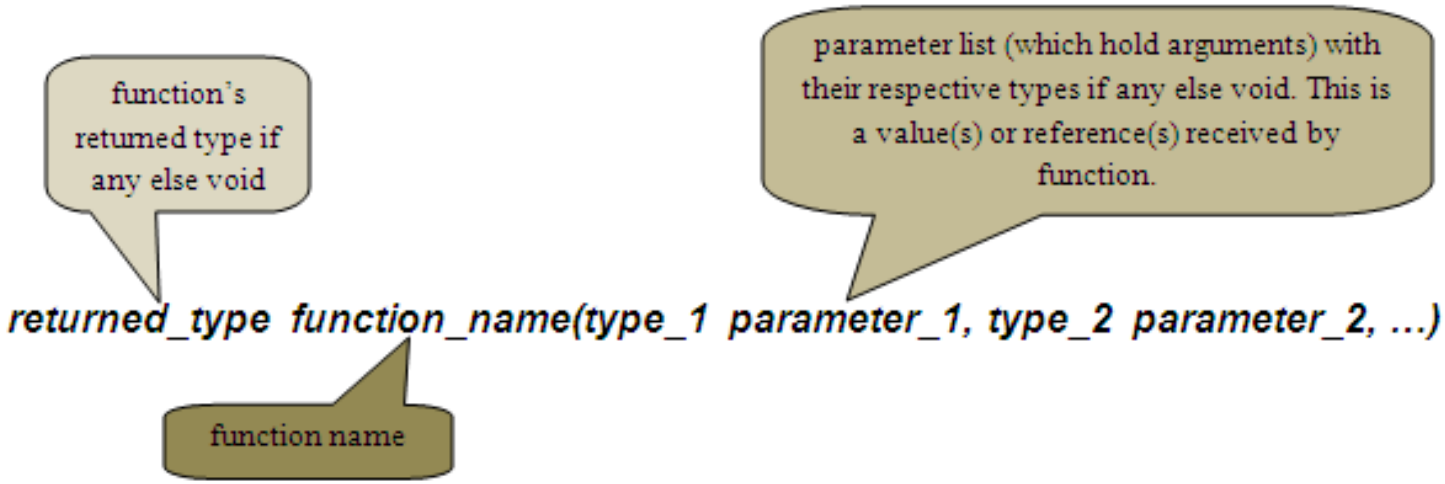
C FUNCTIONS

- 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
(`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,

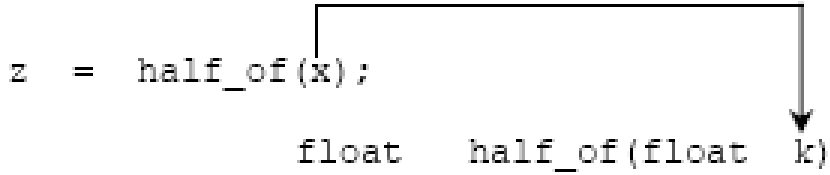
```
void myfunction(int x, float y, char z)
void yourfunction(float myfloat, char mychar)
int  ourfunction(long size)
```

- 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

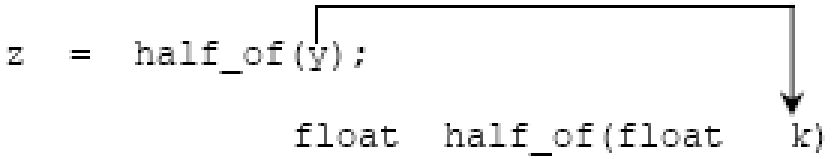
```
long thefunction(void)
void testfunct(void)
int  zerofunct()
```

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`.

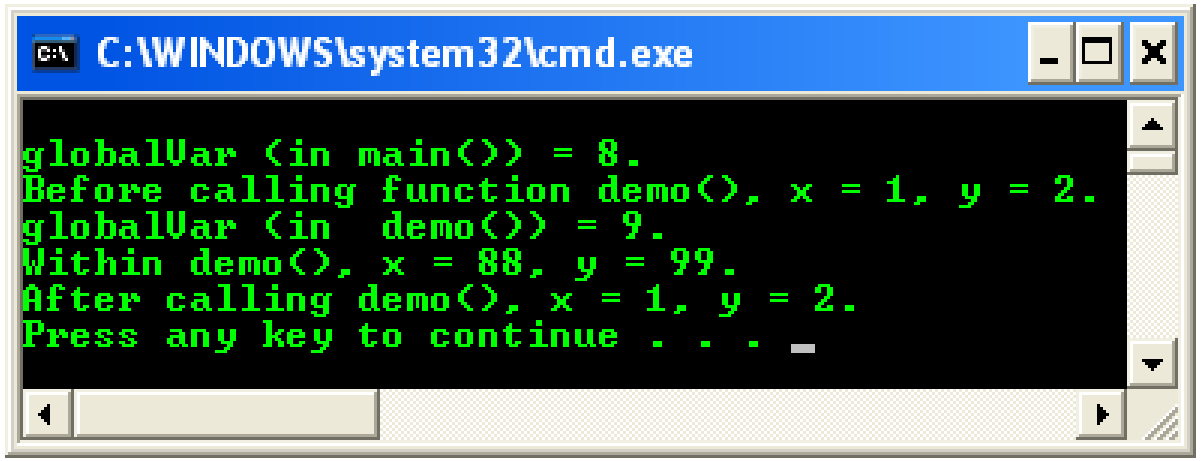
C FUNCTIONS

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 used like any other variable.
- Outside of any functions, those variables are called global variables.

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 [illegible] 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:

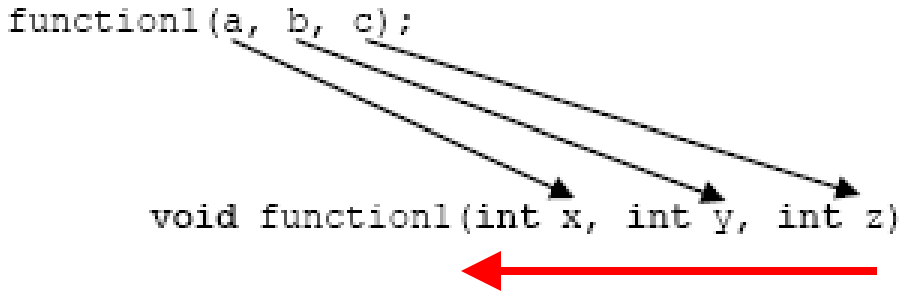
1. To use a variable in a function, we must declare it in the function header or the function body.
2. For a function to obtain a value from the calling program (caller), the value must be passed as an argument (the actual value).
3. For a calling program (caller) to obtain a value from function, the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

C FUNCTIONS

- 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

- 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

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 [illegible] 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

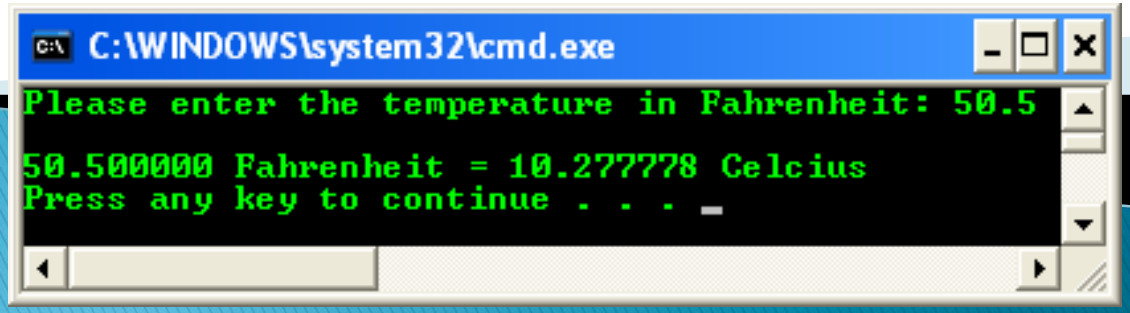
Using Predefined Functions from Header File

- 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



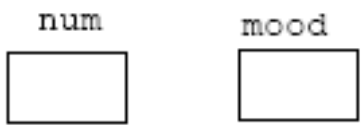
C FUNCTIONS

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

```
#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

- 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()`, 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

- 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

	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