C Fundamental

Functions

Objectives

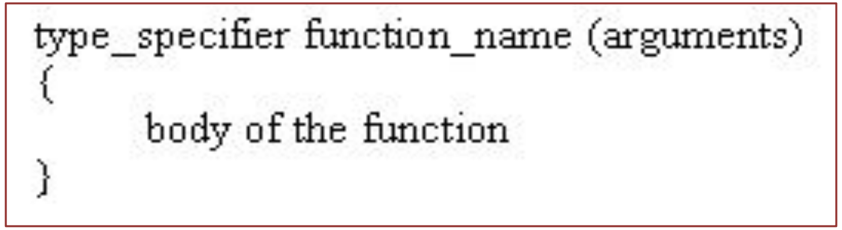
- The structure of a function
- Types of function
- Arguments and Parameters
- Call functions
- Function pointers
- Storage classes

Section 1

THE STRUCTURE OF A FUNCTION

The Function Structure

The general syntax of a function in C is :



```
type_specifier function_name (arguments)
{
        body of the function
}
```

- The function contains the set of programming statements enclosed by {}
- The type_specifier specifies the data type of the value, which the function will return.
- A valid function name is to be assigned to identify the function
- Arguments appearing in parentheses are also termed as formal parameters.

Function: Prototypes

- Specifies the data types of the arguments

```
char aFunction(int x, int y);
```

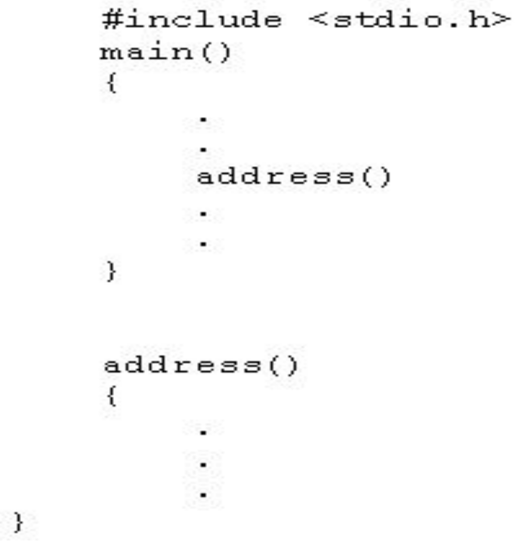
Advantage:

Any illegal type conversions between the arguments used to call a function and the type definition of its parameters is reported

```
char    noParam
();
```

Function: Declaration

- Declaring a function becomes compulsory when the function is being used before its definition
- The address() function is called before it is defined
- Some C compilers return an error, if the function is not declared before calling
- This is sometimes referred to as Implicit declaration



```
#include <stdio.h>
main()
{
        .
        .
        address()
        .
        .
}

address()
{
        .
        .
        .
}
```

Functions: Advantage

- A function is a self-contained program segment that carries out a specific, well-defined task
- Functions are generally used as abbreviations for a series of instructions that are to be executed more than once
- Functions are easy to write and understand
- Debugging the program becomes easier as the structure of the program is more apparent, due to its modular form
- Programs containing functions are also easier to maintain, because modifications, if required, are confined to certain functions within the program

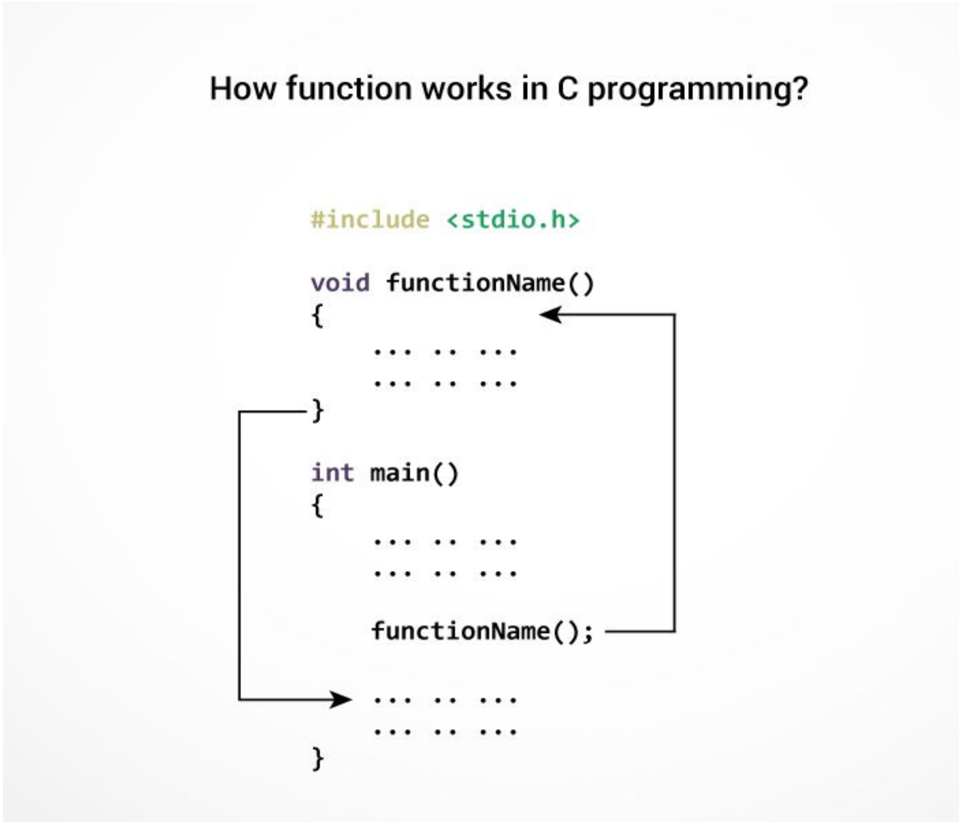
Functions: Example

An example of function:

```
#include <stdio.h>

void functionName()
{
        // lines of code
}

void main()
{
        functionName();
}
```

Function: Working Sample

The control of the program jumps back to the **main()** function once code inside the function definition is executed.



```
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

Section 2

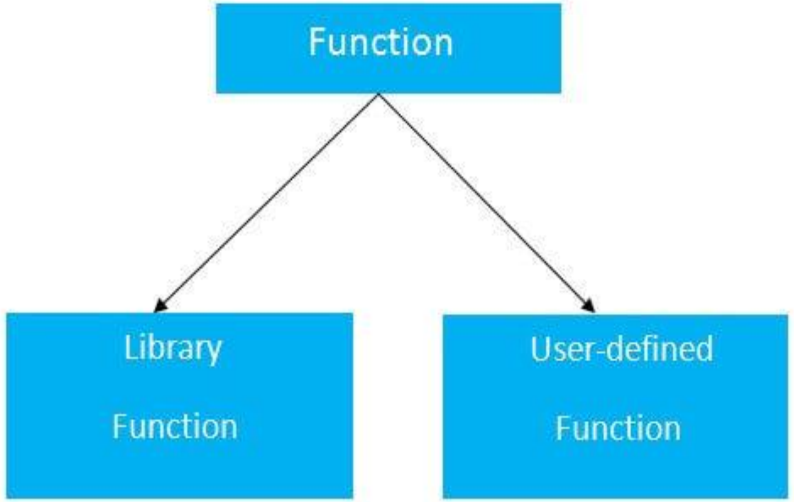
TYPES OF FUNCTION

Types of Function

There are two types of functions in C programming:

<u>Library Functions</u>: are the functions which are declared in the C header files such as **scanf()**, **printf()**, **gets()**, **puts()**, …etc.

<u>User-defined functions</u>: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



Types of Function: Standard library

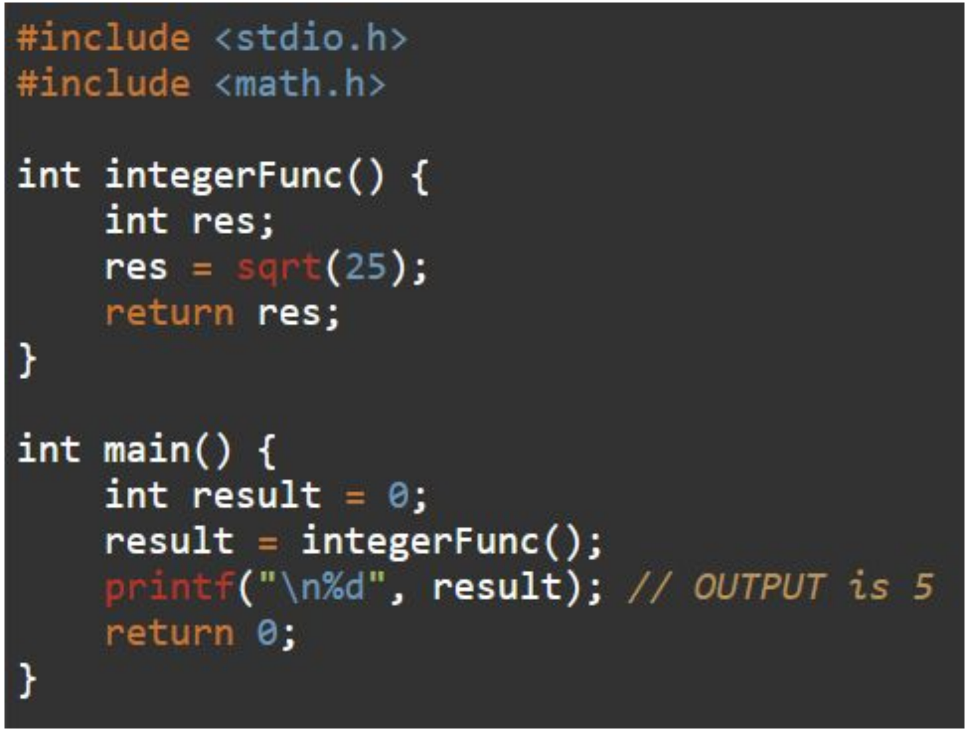
The standard library functions are built-in functions in C programming. These functions are defined in header files. For example:

- The **printf()** is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the **stdio.h** header file.
- Hence, to use the **printf()** function, we need to include the **stdio.h** header file using **#include <stdio.h>**.
- The **sqrt()** function calculates the square root of a number. The function is defined in the **math.h** header file (**#include <math.h>**).

Types of Function: Standard library (1)

About use function in standard library, look at the example. Use function sqrt() in library math.h to calculate square root.

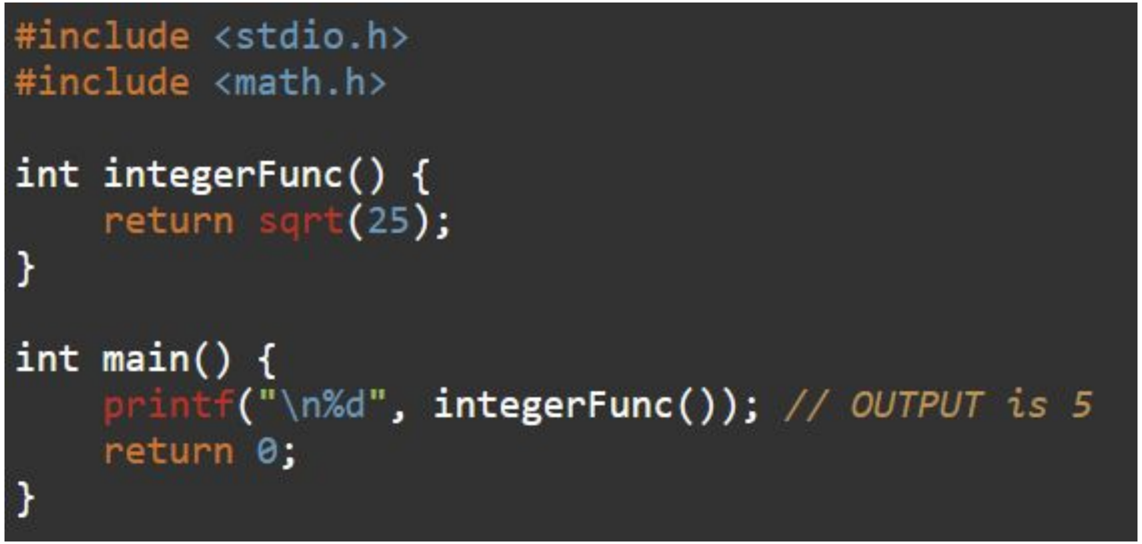
EXAMPLE



```
#include <stdio.h>
#include <math.h>

int integerFunc() {
    int res;
    res = sqrt(25);
    return res;
}

int main() {
    int result = 0;
    result = integerFunc();
    printf("\n%d", result); // OUTPUT is 5
    return 0;
}
```

Types of Function: Standard library (2)

Continue example above. Code will be shortened by removing intermediate variables:



```c
#include <stdio.h>
#include <math.h>

int integerFunc() {
    return sqrt(25);
}

int main() {
    printf("\n%d", integerFunc()); // OUTPUT is 5
    return 0;
}
```

The function's return value is used as a variable

Types of Function: User-defined

You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

```
#include <stdio.h>
void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

The execution of a C program begins from the **main()** function.

When the compiler encounters **functionName();**, control of the program jumps to

void functionName()

And, the compiler starts executing the codes inside **functionName()**.

User-defined Function: Syntax

```
returnType functionName(type1 argument1, type2 argument2, ...);
```

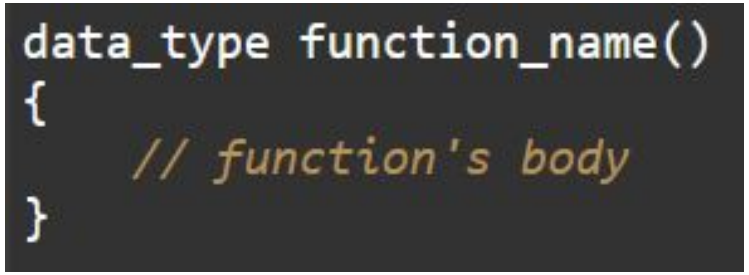
Example: `int addNum(int a, int b);`

In the above example, **int addNum(int a, int b);** is the function prototype which provides the following information to the compiler:

- name of the function is **addNum ()**
- return type of the function is int
- two arguments of type int are passed to the function

The function prototype is not needed if the user-defined function is defined before the **main()** function.

User-defined Function: Data Type



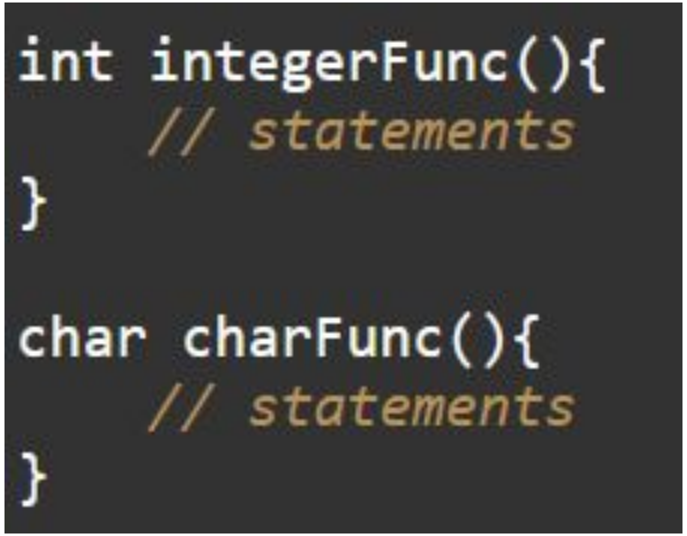
```
data_type function_name()
{
    // function's body
}
```

- The type_specifier is not written prior to the function **squarer**(), because **squarer**() returns an integer type value
- The **type_specifier** is not compulsory if an integer type of value is returned or if no value is returned
- However, to avoid inconsistencies, a data type should be specified

User-defined Function: Data Type (1)

Example **about data type of a function.**

Functions also have data types like variables



```
int integerFunc(){
        // statements
}

char charFunc(){
        // statements
}
```

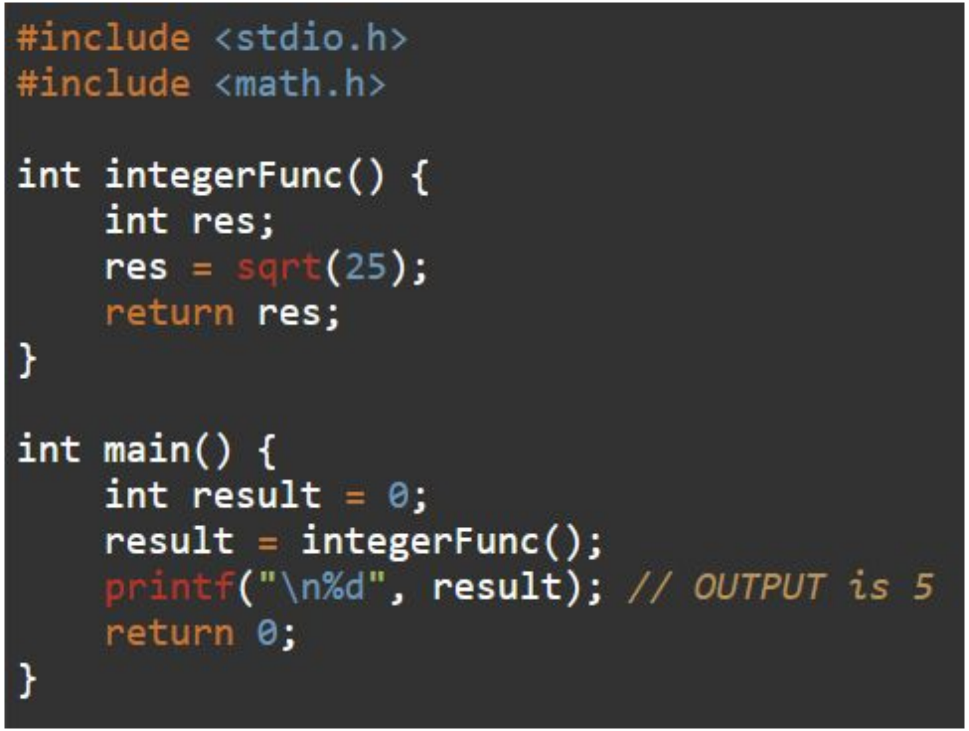
User-defined Function: Calling

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call:

```
functionName(argument1, argument2, ...);
```

User-defined Function: Calling (1)



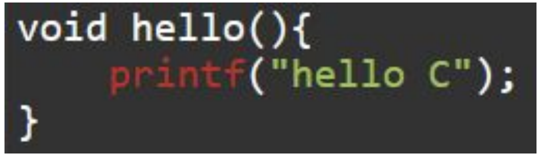
```c
#include <stdio.h>
#include <math.h>

int integerFunc() {
    int res;
    res = sqrt(25);
    return res;
}

int main() {
    int result = 0;
    result = integerFunc();
    printf("\n%d", result); // OUTPUT is 5
    return 0;
}
```

In the example, the function call is made using integerFunc(); statement inside the main() function.

User-defined Function: Return

A C function may or may not return a value from the function. If you don't have to return any value from the function, use **void** for the return type.

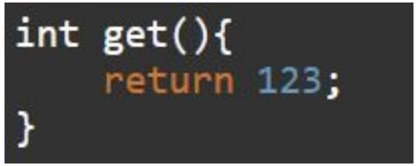
Example without return value:



```
void hello(){
    printf("hello C");
}
```

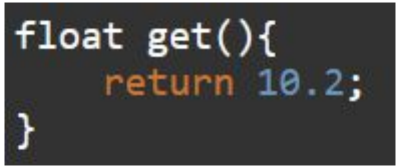
If you want to return any value from the function, you need to use any data type such as **int**, **char**, **float**, etc. The return type depends on the value to be returned from the function.

Example with return value:



```
int get(){
    return 123;
}
```

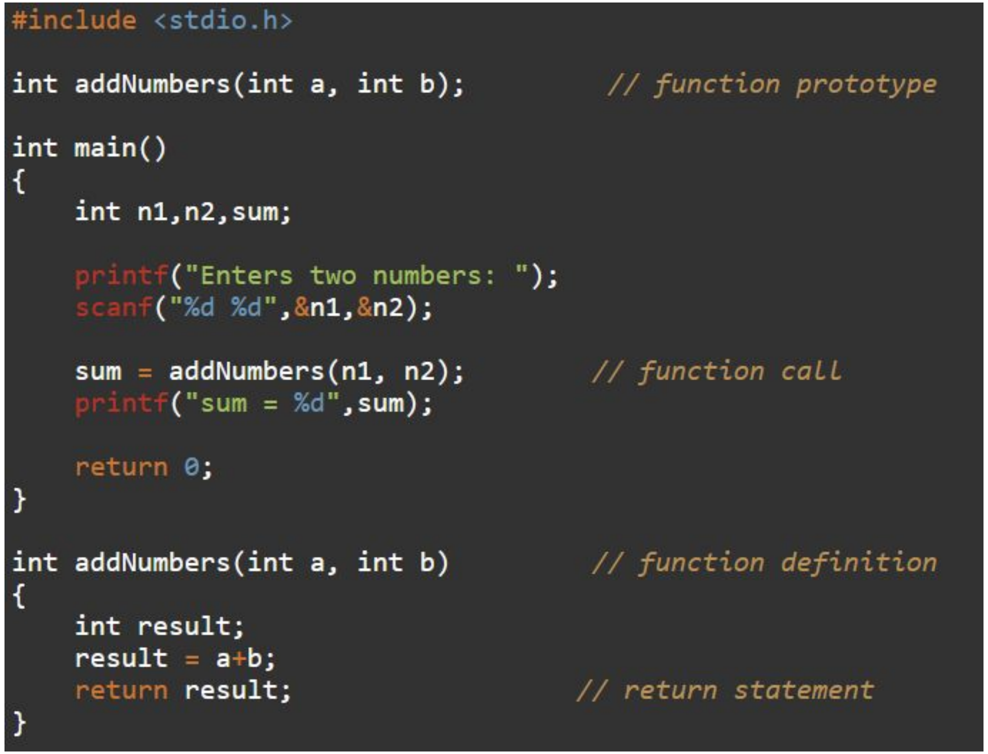
or



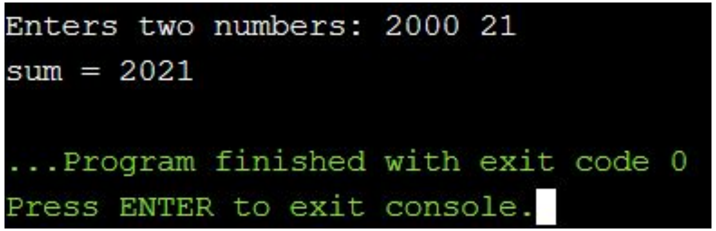
```
float get(){
    return 10.2;
}
```

User-defined Function: Example

Here is an example to add two integers. To perform this task, we have created an user-defined **addNumbers()**.



```c
#include <stdio.h>

int addNumbers(int a, int b);         // function prototype

int main()
{
    int n1,n2,sum;

    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);         // function call
    printf("sum = %d",sum);

    return 0;
}

int addNumbers(int a, int b)          // function definition
{
    int result;
    result = a+b;
    return result;                    // return statement
}
```

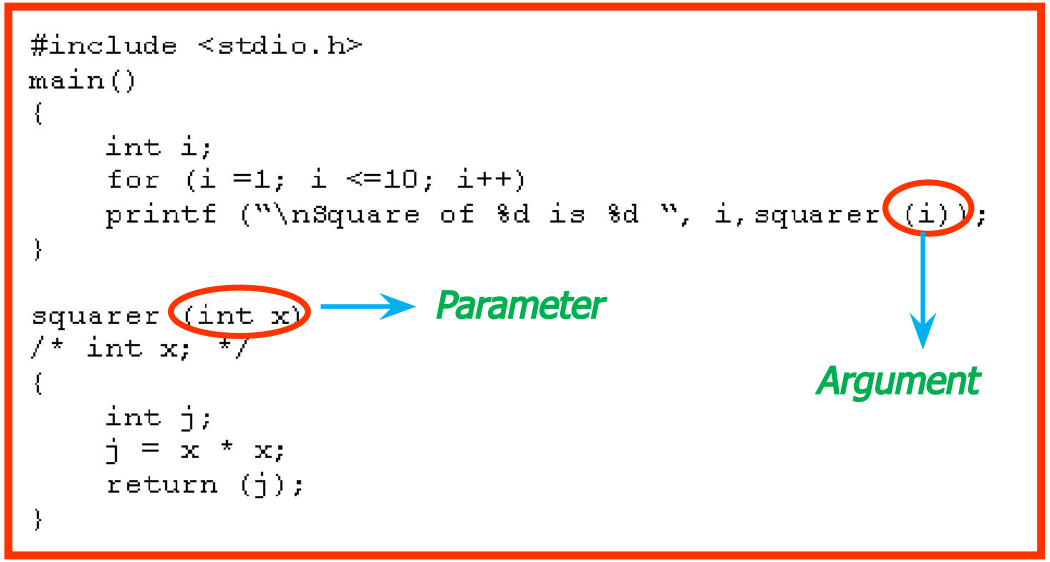


```
Enters two numbers: 2000 21
sum = 2021

...Program finished with exit code 0
Press ENTER to exit console.
```

Section 3

ARGUMENTS AND PARAMETERS

Arguments and Parameters



```
#include <stdio.h>
main()
{
     int i;
     for (i =1; i <=10; i++)
     printf ("\nSquare of %d is %d ", i,squarer (i));
}

squarer (int x)
/* int x; */
{
     int j;
     j = x * x;
     return (j);
}
```

Parameter is variable in the declaration of function.

Argument is the actual value of this variable that gets passed to function.

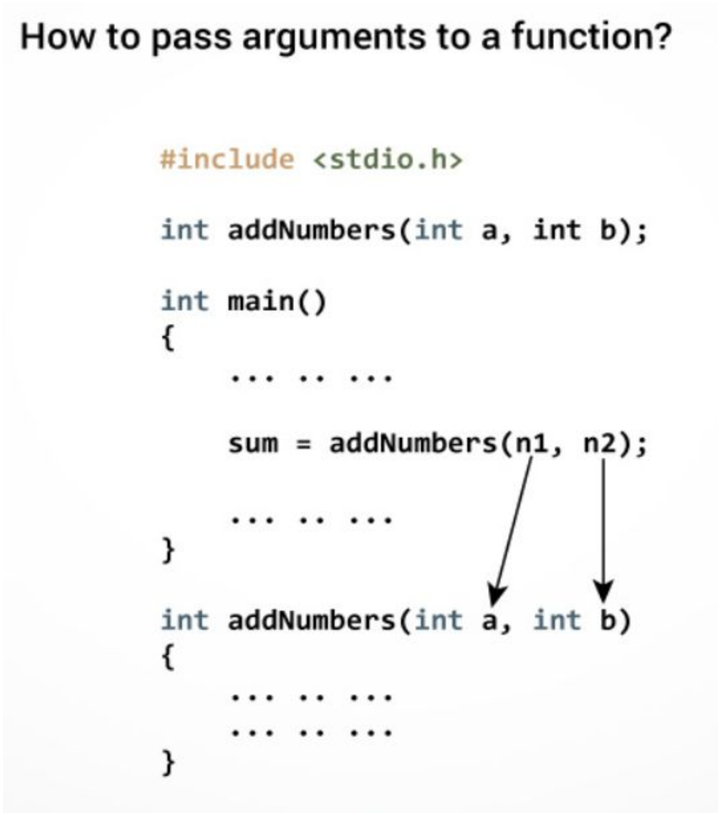
- The program calculates the square of numbers from 1 to 10
- The function works on data using arguments
- The data is passed from the main() to the squarer() function

Arguments and Parameters (1)

- An **argument** is referred to the values that are passed within a function when the function is called. These values are generally the source of the function that require the arguments during the process of execution.

- A **parameter** is referred to as the variables that are defined during a function declaration or definition. These variables are used to receive the arguments that are passed during a function call

Arguments and Parameters: Passing value

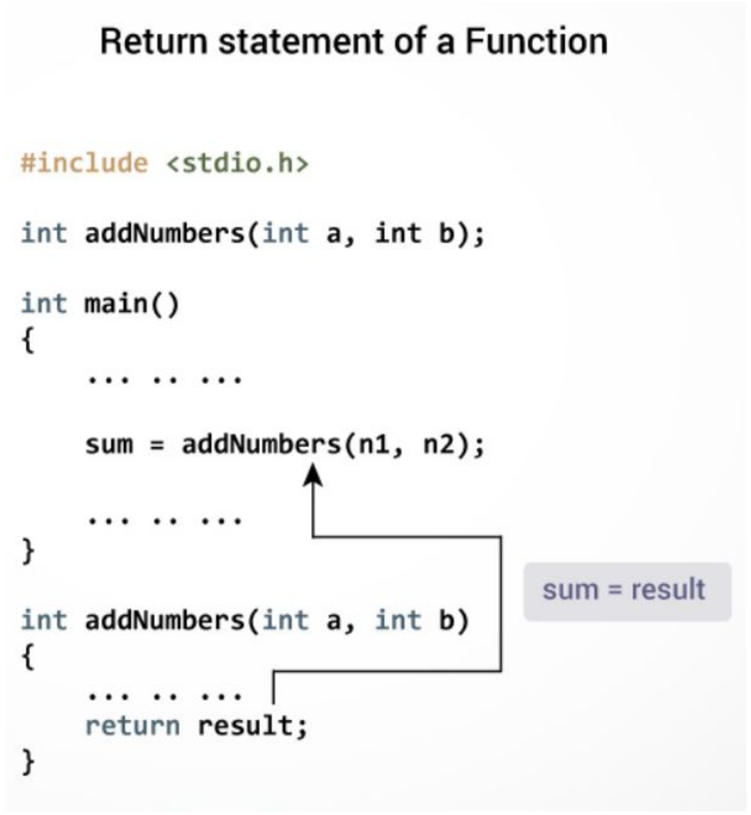


How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

In programming, argument refers to the variable passed to the function. In the example, two variables **n1** and **n2** are passed during the function call. The parameters ***a*** and ***b*** accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

Arguments and Parameters: Return Statement

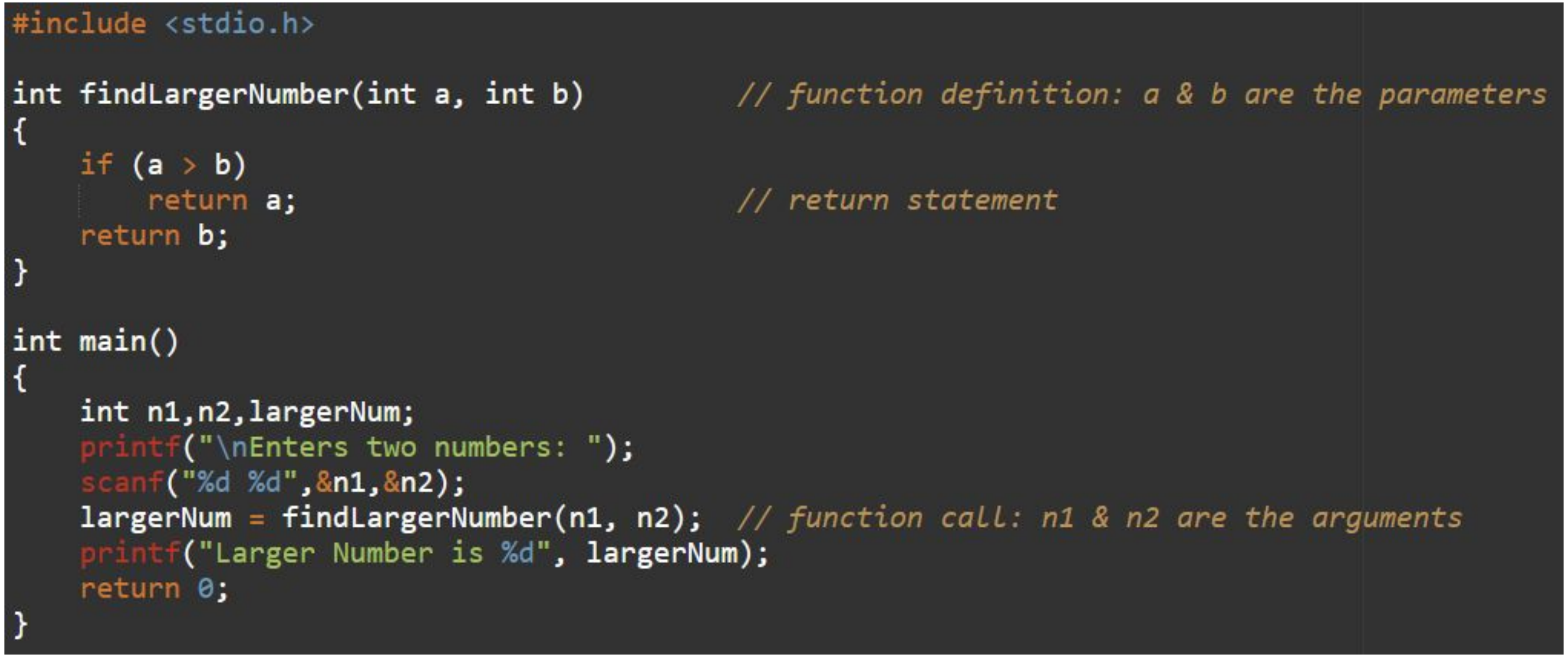

Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```

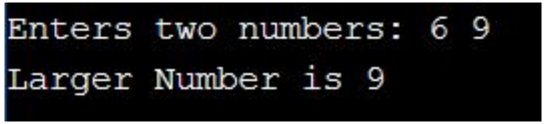
The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the example, the value of the result variable is returned to the main function. The sum variable in the **main()** function is assigned this value.

Arguments and Parameters: Example



```c
#include <stdio.h>

int findLargerNumber(int a, int b)          // function definition: a & b are the parameters
{
    if (a > b)
        return a;                           // return statement
    return b;
}

int main()
{
    int n1,n2,largerNum;
    printf("\nEnters two numbers: ");
    scanf("%d %d",&n1,&n2);
    largerNum = findLargerNumber(n1, n2);  // function call: n1 & n2 are the arguments
    printf("Larger Number is %d", largerNum);
    return 0;
}
```



```
Enters two numbers: 6 9
Larger Number is 9
```

Section 4

CALL FUNCTION

Invoking a Function

- A semicolon is used at the end of the statement when a function is called, but not after the function definition
- Parentheses are compulsory after the function name, irrespective of whether the function has arguments or not
- Only one value can be returned by a function
- The program can have more than one function
- The function that calls another function is known as the calling function/routine
- The function being called is known as the called function/routine

Function Scope rules

- Scope Rules - Rules that govern whether one piece of code knows about or has access to another piece of code or data
- The code within a function is private or local to that function
- Two functions have different scopes
- Two Functions are at the same scope level
- One function cannot be defined within another function

Calling The Functions

We have two types in C:

- Call by value
- Call by reference

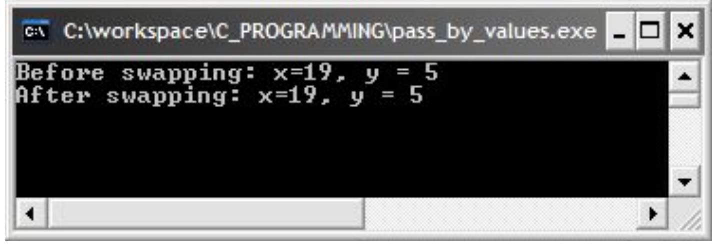
Calling By Value

- In C, by default, all function arguments are passed by value
- When arguments are passed to the called function, the values are passed through temporary variables
- All manipulations are done on these temporary variables only
- The arguments are said to be passed by value when the value of the variable are passed to the called function and any alteration on this value has no effect on the original value of the passed variable

Calling By Value - Example

```c
/* Pass-by-Value example */
#include <stdio.h>
int swap (int a, int b);
int main (){
  int x = 19, y = 5;
  printf("Before swapping: x=%d, y = %d\n",x,y);
  swap(x, y);
  printf("After swapping: x=%d, y = %d",x,y);
  return 0;
}
int swap (int a, int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Output:



```
Before swapping: x=19, y = 5
After swapping: x=19, y = 5
```

Calling By Reference

- In call by reference, the function is allowed access to the actual memory location of the argument and therefore can change the value of the arguments of the calling routine

- Definition

```
getstr(char *ptr_str, int *ptr_int);
```

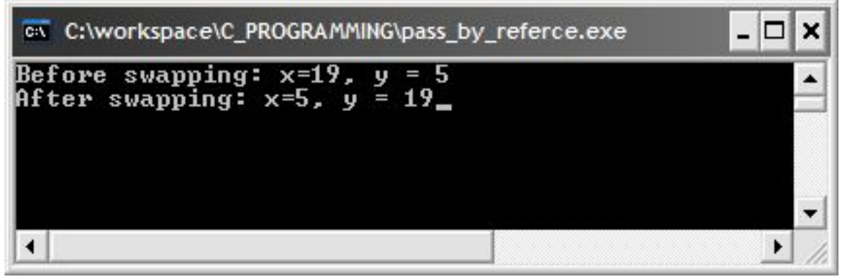
- Call

```
getstr(pstr, &var);
```

Calling By Reference - Example

```c
/* Pass-by-Reference example */
#include <stdio.h>
int swap (int *a, int *b);
int main (){
  int x = 19, y = 5;
  printf("Before swapping: x=%d, y = %d\n",x,y);
  swap(&x, &y);
  printf("After swapping: x=%d, y = %d",x,y);
  return 0;
}
int swap (int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Output



C:\workspace\C_PROGRAMMING\pass_by_referce.exe

```
Before swapping: x=19, y = 5
After swapping: x=5, y = 19_
```

Nesting Function Calls

```
    main()
     {
          .
          .
          palindrome
();
          .
          .
     }
```

```
palindrome()
{
     .
     .
     getstr();
     reverse();
     cmp();
     .
     .
}
```

Functions in Multi-files Programs

- Functions can also be defined as **static** or **external**

- Static functions are recognized only within the program file and their scope does not extend outside the program file

```
static fn _type fn_name (argument list);
```

- External function are recognized through all the files of the program

```
extern fn_type fn_name (argument list);
```

Section 5

FUNCTION POINTER

Function pointer: Syntax

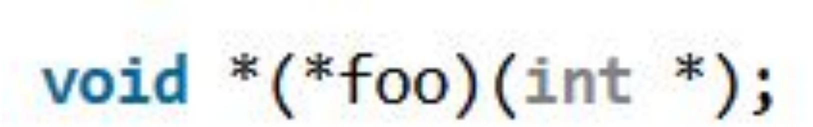
The syntax for declaring a function pointer might seem messy at first, but in most cases it's really quite straight-forward once you **understand** what's going on. Let's look at a simple example:

```
void (*foo)(int);
```

In this example, **foo** is a pointer to a function taking one argument, an integer, and that returns void. It's as if you're declaring a function called "***foo**", which takes an int and returns void; now, if ***foo** is a function, then foo must be a pointer to a function.
The key to writing the declaration for a function pointer is that you're just writing out the declaration of a function but with **(*func_name)** where you'd normally just put **func_name**.

Function pointer: Syntax (1)

Reading Function Pointer Declarations:

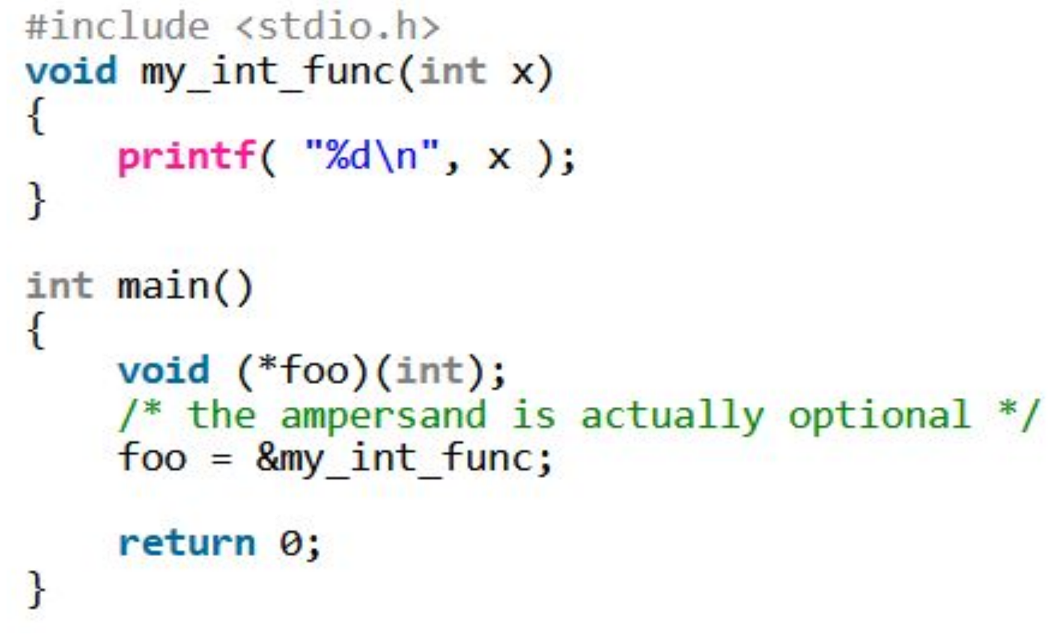


```
void *(*foo)(int *);
```

Here, the key is to read inside-out; notice that the innermost element of the expression is ***foo**, and that otherwise it looks like a normal function declaration. ***foo** should refer to a function that returns a **void *** and takes an **int ***. Consequently, foo is a pointer to just such a function.

Function pointer: Initializing

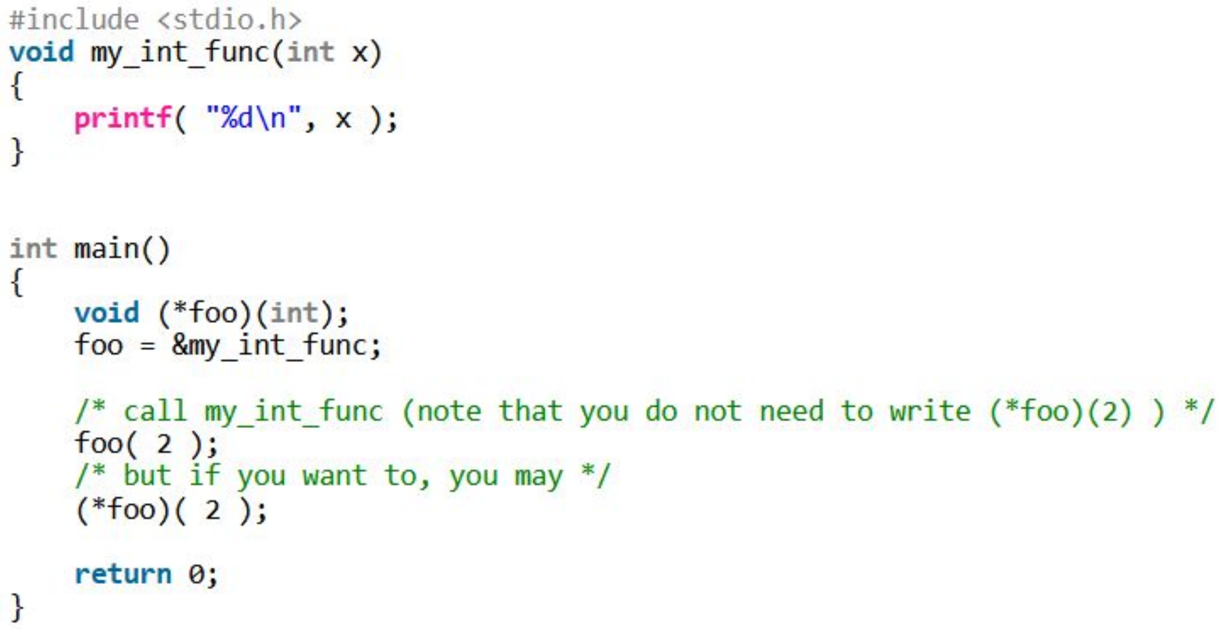
To initialize a function pointer, you must give it the address of a function in your program. The syntax is like any other variable:



```c
#include <stdio.h>
void my_int_func(int x)
{
    printf( "%d\n", x );
}

int main()
{
    void (*foo)(int);
    /* the ampersand is actually optional */
    foo = &my_int_func;

    return 0;
}
```

Function pointer: Using

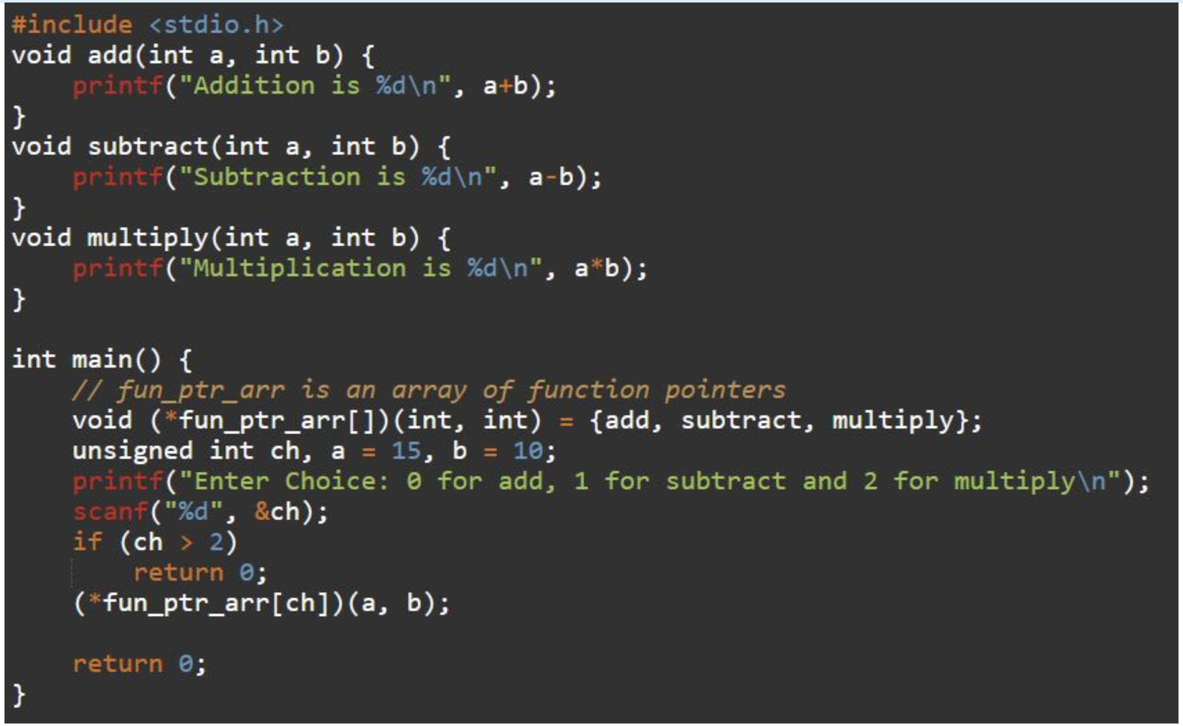
To call the function pointed to by a function pointer, treat the function pointer as though it were the name of the function to call. The act of calling it performs the dereference; there's no need to do it yourself:



```
#include <stdio.h>
void my_int_func(int x)
{
    printf( "%d\n", x );
}


int main()
{
    void (*foo)(int);
    foo = &my_int_func;

    /* call my_int_func (note that you do not need to write (*foo)(2) ) */
    foo( 2 );
    /* but if you want to, you may */
    (*foo)( 2 );

    return 0;
}
```

Function pointer: Example

Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.



```c
#include <stdio.h>
void add(int a, int b) {
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b) {
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b) {
    printf("Multiplication is %d\n", a*b);
}

int main() {
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;
    printf("Enter Choice: 0 for add, 1 for subtract and 2 for multiply\n");
    scanf("%d", &ch);
    if (ch > 2)
        return 0;
    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```

Section 6

STORAGE CLASS

Storage Classes: Location of Variables

- Local Variables
  - ✔ Declared inside a function
  - ✔ Created upon entry into a block and destroyed upon exit from the block
- Formal Parameters
  - ✔ Declared in the definition of function as parameters
  - ✔ Act like any local variable inside a function
- Global Variables
  - ✔ Declared outside all functions
  - ✔ Holds value throughout the execution of the program

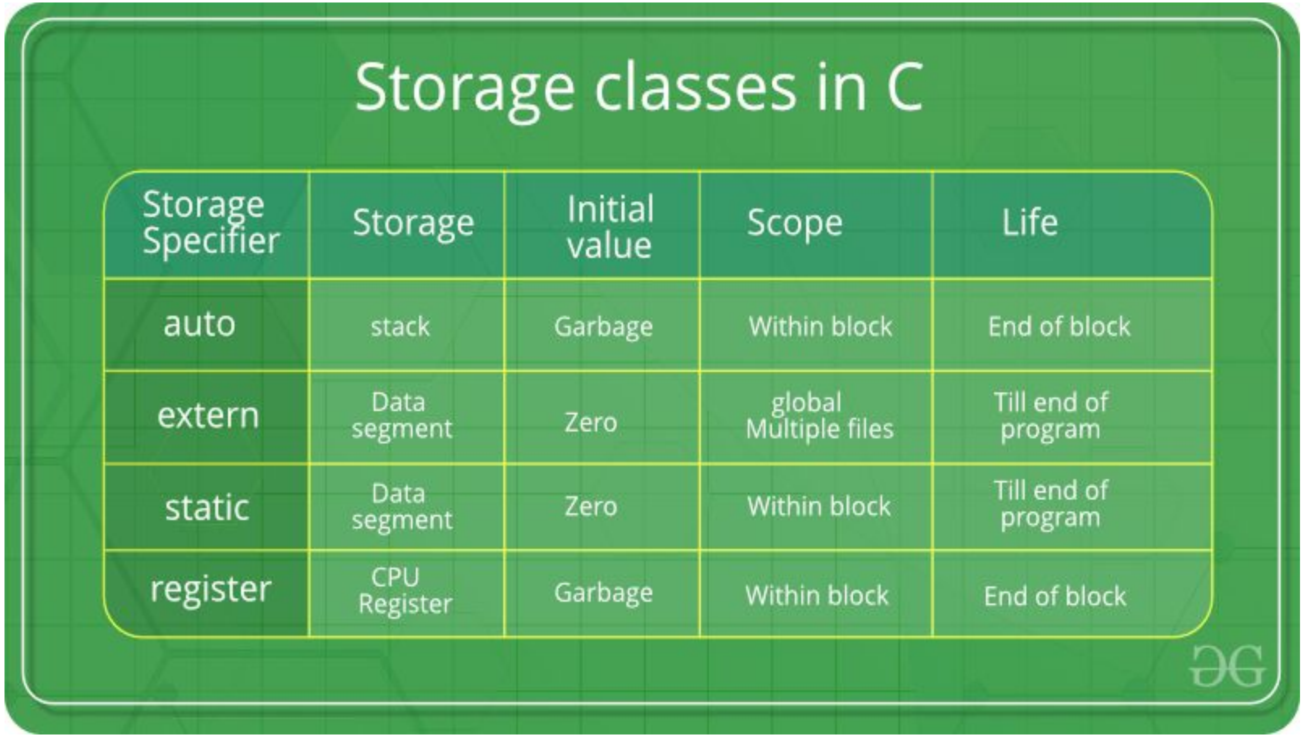
Storage Classes (1)

- Every C variable has a characteristic called as a storage class
- The storage class defines two characteristics of the variable:
  - **Lifetime** – The lifetime of a variable is the length of time it retains a particular value
  - **Visibility** – The visibility of a variable defines the parts of a program that will be able to recognize the variable

Storage Classes (2)

- **auto**matic
- **extern**al
- **static**
- **register**

Storage Classes (3)



Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

Thank you

Q&A