CS100
Introduction to Programming

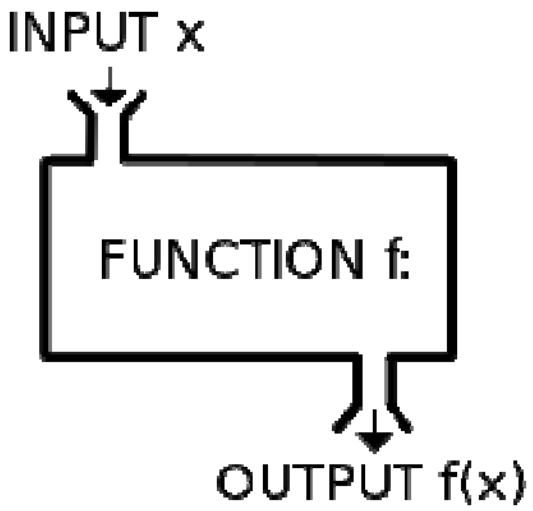
Lecture 3. Functions & Pointers

I. Functions

Function Definition in Mathematics

- **A mapping from a set to another**

$$y = f(x) \qquad \{(x, f(x)) : x \in X\}$$

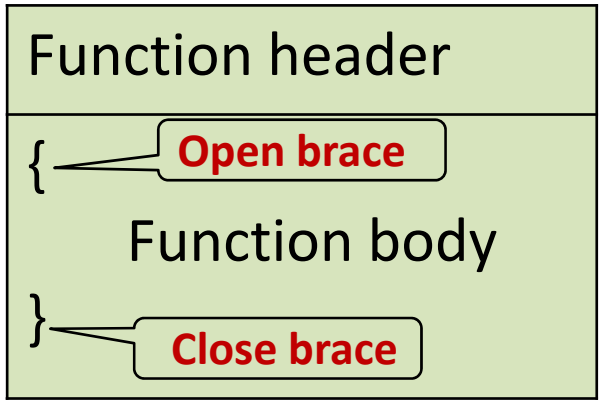


Example:

$$f(x) = \sin(x^2 + 1)$$

Function Definition in C

- A **function** is a self-contained unit of code to carry out a specific task, e.g. printf(…), sqrt(…).
- A function consists of
  - a header
  - an opening curly brace
  - a body
  - a closing curly brace

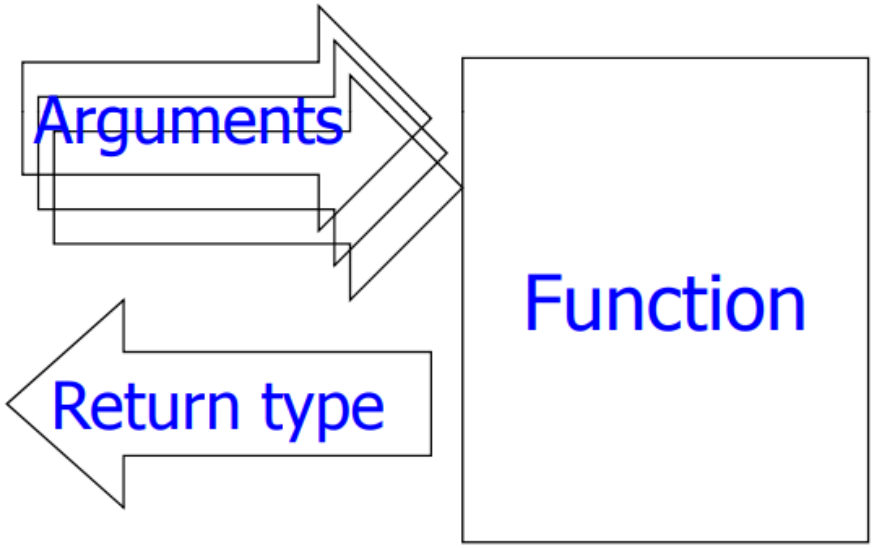



```
float findMaximum(float x, float y)
{
   // variable declaration
   float maxnum;

   // find the max number
   if (x >= y)
      maxnum = x;
   else
      maxnum = y;

   return maxnum;
}
```

Function Header

Format:

Return_type Function_name(Argument_list)



Argument List

- Arguments define the **data** passed into the function.
- Each parameter has a **data type** (e.g. int, char, etc.)
- A function can have no parameter, one argument or many arguments.

type argument_name[, type argument_name]

- The data type for each argument must be declared.
- The function assumes that these inputs will be supplied to the function when it is being called.

Return Type

Return Type is the data type returned from the function. It can be *int*, *float*, *char*, *void*, or nothing.

- **int** – the function will return a value of type int

```
int successor(int num)
{
   return num + 1; /* has a return statement */
}
```

- **float** – the function will return a value of type float
- **void** – the function will not return any value.

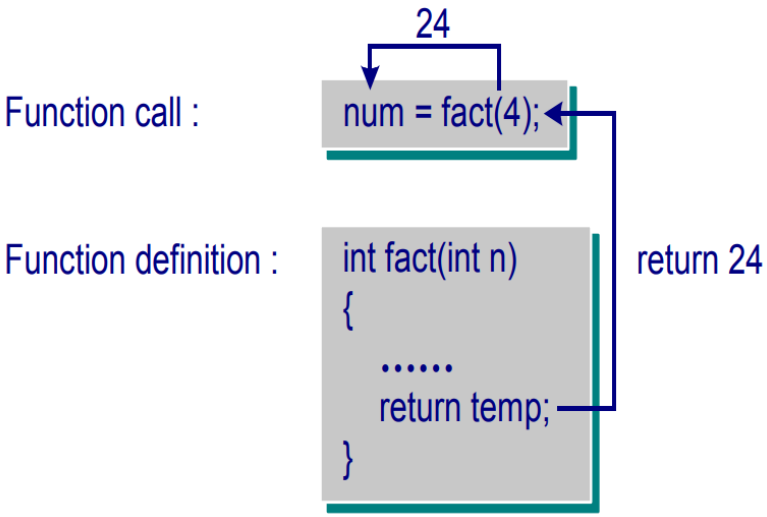
```
void hello_n_times(int n)
{
   int count;
   for (count = 0; count < n; count++)
      printf("hello\n");
   /* no return statement */
}
```

- **nothing** – if defined with no type, the default type is int

The return statement

- It may appear in **any place** and in **more than one place** inside the function body.

```
int fact(int n)
{
   int temp = 1;
   if (n < 0) {
      printf("error: must be positive\n");
      return 0;
   } else if (n == 0) {
      return 1;
   } else {
      for (; n > 0; n--)
         temp *= n;
   }
   return temp;
}
```



Output:

Enter a positive number: ***4***

The factorial of 4 is 24

Function: Examples

```
char findGrade(float marks)
{
   char grade;
   if (marks >= 50)
      grade = ‘P’;
   else
      grade = ‘F’;
   return grade;
}
```

```
float areaOfCircle(float radius)
{
   const float pi = 3.14;
   float area = pi*radius*radius;
   return area;
}
```

It’s only an example, not a real policy.

Function Prototypes

- This is to declare a function. A function declaration is called a **function prototype**. It provides the information about
  - the **type** of the function
  - the **name** of the function
  - the **number and types of the arguments**

- The declaration may be the same as the function header terminated by a semicolon. For example:

  void hello_n_times(int n);

- Or the function is declared without giving the parameter names:

  double distance(double, double);

- The declaration has to be done before the function is called:
  - before the main() header
  - inside the main() body or
  - inside any function which uses it

Function Prototypes: Examples

```
#include <stdio.h>
// before the main()
// function prototype
int factorial(int n);

void main()
{
....
}

/* function definition */
int factorial(int n)
{
....
}
```

```
#include <stdio.h>
// inside the main()
void main()
{
   // function prototype
   int factorial(int);
   // then use the function factorial()
   ....
}

/* function definition */
int factorial(int n)
{
....
}
```

Declaration & Implementation of a Function

- **Usually, a function is declared in a header file**
  - A header file (*.h) can contain declarations of multiple functions
  - A header can also define multiple global variable declarations

```
//define a set of basic calculations

float Add(float a, float b);
float Sub(float a, float b);
float Mul(float a, float b);
float Div(float a, float b);
```

Declaration & Implementation of a Function

- **The implementation of a function is usually put in a *.cpp file**
  - Multiple implementations can be done there

```
float Add(float a, float b)
{
     return a+b;
}
...
float Mul(float a, float b)
{
     return a*b;
}
...
```

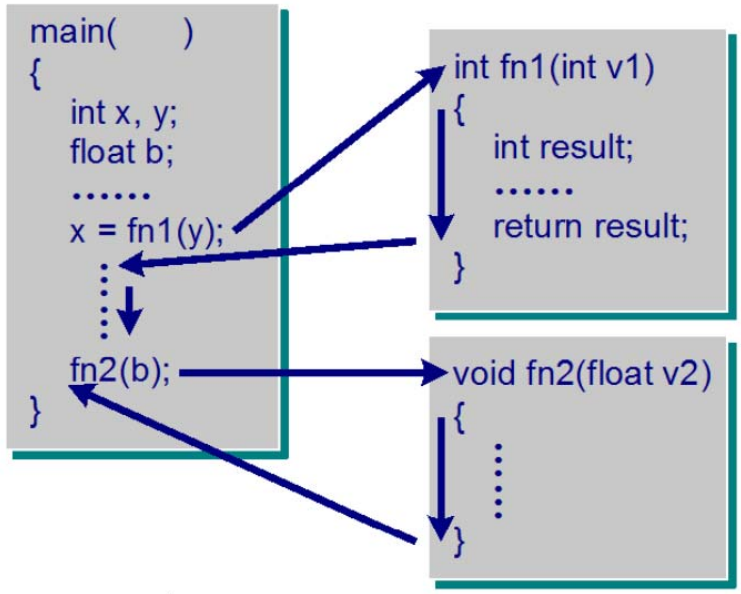
Function Flow

A function call causes the function to be executed.

A function call has the following format:

Function_name(Argument_list);

```
#include <stdio.h>
void hello();  // function prototype

main()
{
    hello();  // function call
    return 0;
}

void hello()  // function definition
{
    printf("hello\n");
}
```



Function Flow: Examples

```c
#include <stdio.h>
char findGrade(float);

int main() {
    char answer;
    answer = findGrade(68.5);
    printf("Grade is %c", answer);
    return 0;
}

char findGrade(float marks) {
    ...
}
```

```c
#include <stdio.h>
float areaOfCircle(float);

int main() {
    float answer;
    answer = areaOfCircle(2.5);
    printf("Area is %.1f", answer);
    return 0;
}

float areaOfCircle(float radius) {
    ...
}
```

II. Scope of Variables & Parameter Passing

Scope of Variables in a Function

- Variables declared in a function is **ONLY** visible within that function.
- In the example below, variables **radius**, **pi** and **area** are **NOT** visible outside this function.

```
float areaOfCircle(float radius)
{
  const float pi = 3.14;
  float area = pi*radius*radius;
  return area;
}
```

Scope of Variables

```
#include <stdio.h>
int global_var = 5;       // global variable
int fn1(int, int);
float expn(float);
int main() {
   char reply;            // local variables – these two variables are
   int num;               // only known inside main()
   ...
}

int fn1(int x, int y) {   // local x, y – formal parameters
   ...                    // only known inside this function

   float fnum;            // local – these two variables are known
   int temp;              // in this function only
   global_var += 10;
   ...
}

float expn(float n) {
   float temp;            // local – this variable is known in expn()
   ...
}
```

Parameter Passing: Call by Value

Communications between a function and the calling body is done through **arguments** and the **return value** of a function.

```
#include <stdio.h>
int add1(int);

int main()
{
    int num = 5;
    num = add1(num); // num – argument
    printf("The value of num is: %d", num);
    return 0;
}

int add1(int value) // value – parameter
{
    return ++value;
}
```

Output:
The value of num is: 6

Parameter Passing: Example

```
#include <stdio.h>
#include <math.h>
double distance(double, double);

int main(void)
{
    double dist;
    double x=2.0, y=4.5, a=3.0, b=5.5;
    dist = distance(2.0, 4.5); // 2.0, 4.5 – arguments
    printf("The dist is %f\n", dist);
    dist = distance(x*y, a*b); // x*y, a*b – arguments
    printf("The dist is %f\n", dist);
    return 0;
}

double distance(double x, double y) // x, y – parameters
{
    return sqrt(x*x + y*y);
}
```

Output:
The dist is 4.924429
The dist is 18.794946

Function Calling Another Function

```
#include <stdio.h>
int max3(int, int, int); // function prototypes
int max2(int, int);

int main(void)
{
    int x, y, z;
    printf("Input 3 integers: ");
    scanf("%d %d %d", &x, &y, &z);
    printf("Maximum of the 3 is %d\n", max3(x, y, z));
    return 0;
}

int max3(int i, int j, int k) {
    printf("Find the max in %d, %d and %d\n", i, j, k);
    return max2(max2(i,j), max2(j, k));
}

int max2(int h, int k) {
    printf("Find the max of %d and %d\n", h, k);
    return h > k ? h : k;
}
```

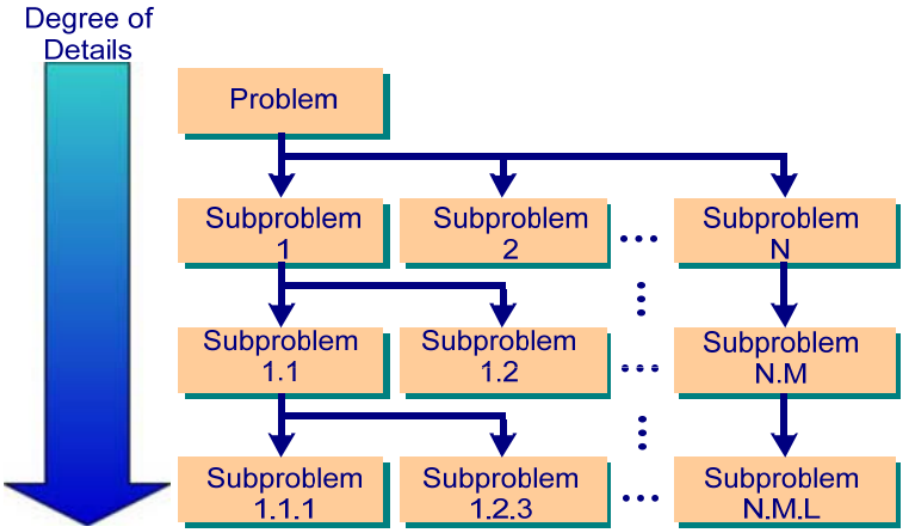
Output:

Input 3 integers: ***7 4 9***
Find the max in 7, 4 and 9
Find the max of 7 and 4
Find the max of 4 and 9
Find the max of 7 and 9
Maximum of the 3 is 9

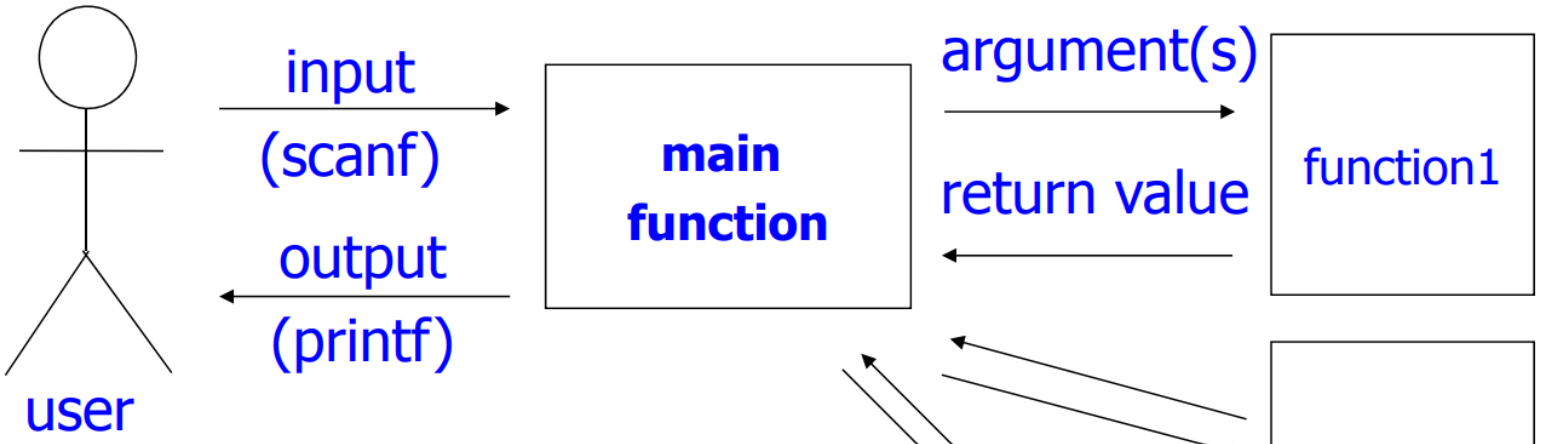
III. Function Decomposition

Functional Decomposition

- **Functional decomposition** basically means ***Stepwise Refinement***.
- In C, functional decomposition starts with the high-level description of the program and decomposes the program (the main() function) into successively smaller functions until we arrive at suitably sized **functions**.
- Then design the code for the individual functions using stepwise refinement. At each level, we are only concerned with **what** the lower level functions will do, but not **how**.



Functional Decomposition



- **Note:**
  - The main function is responsible to interact with the user to read in or display data
  - The other functions' input and output are passed using the arguments and return type via the main function

Functional Decomposition

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

main(void)
{
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
} /* end.  line 2000 */
```

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

main(void)
{
.....
} /* line 20 */

float f1(float h)
{
.....
} /* line 55*/
......
void f18(void)
{
......
} /* line 1560 */
```

Functional Decomposition

- **What is a suitably sized function?**
  - Smaller functions are **easier to understand** (according to research into human psychology)
  - Smaller functions promote software **reusability**.
  - If functions are very small, we need many of them.
  - Function size should be no longer than a page. Better yet, a function should be no longer than half a page.
- **Why do we need functional decomposition?**
  - Program better structured
  - Program easier to understand
  - Program easier to modify
  - Shorter program
  - Easy to debug

Placing Functions into Different Files

- **Why place parts of the program in different files:**
  - The functions in different files can be used by more than one program (**reusability**).
  - Only the files that are changed need be re-compiled.
- **How to place functions in different files?**
- **For example, the code of a program is placed into two files:**
  - One file contains the main(). The main() body calls function1() and function2().
  - These two functions are in another file. There are two constants defined by #define and used by the program, CONST1 and CONST2.
  - The constant definitions and the function declarations are in the header file called def.h.

file1: **mainF.c**

```
#include <stdio.h>
#include “def.h” // double quotes mean the file is in the current directory
int main(void)
{
    ...
    count = function1(h, k);
    function2(&h, &k);
    ...
}
```

file2: **support.c** – contains all the supporting functions

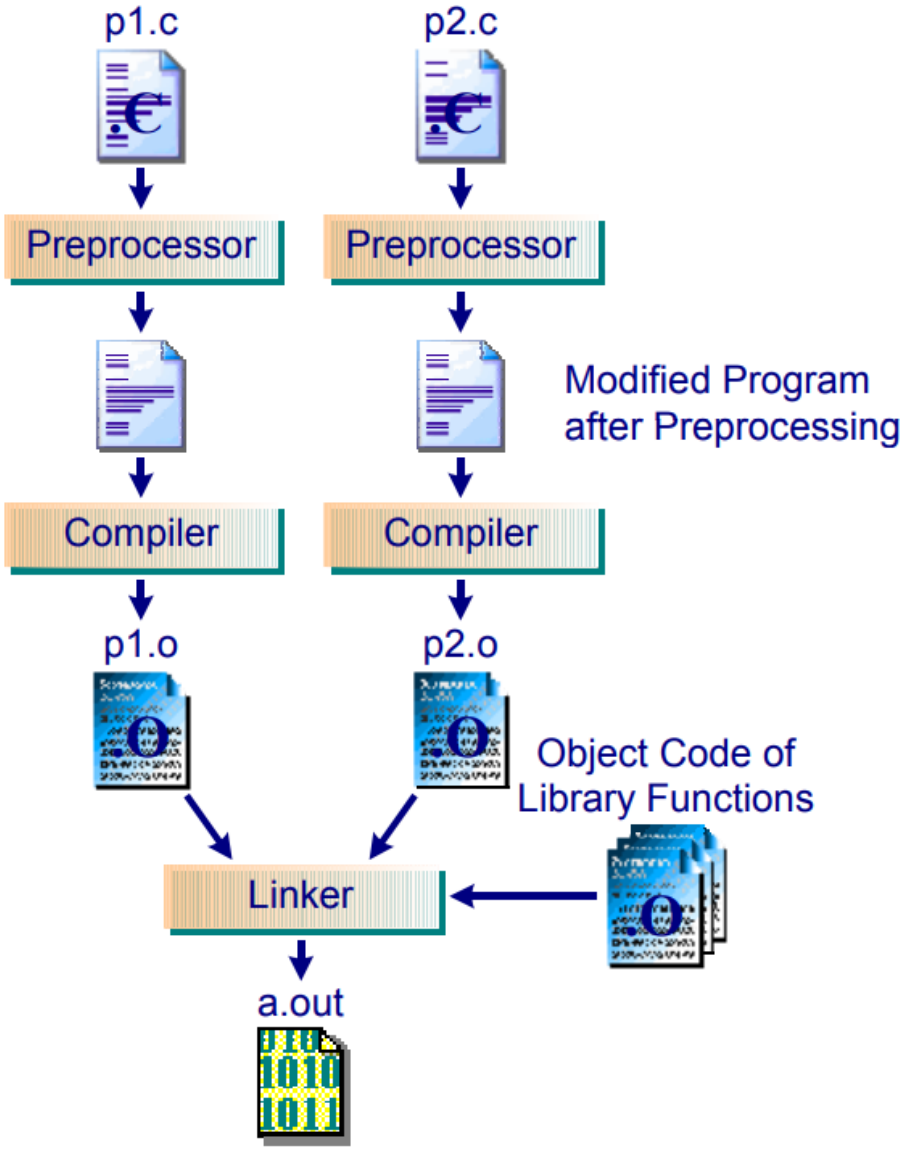
```
#include <stdio.h>
#include “def.h” // double quotes mean the file is in the current directory
int function1(int f, int g)
{
    ...
}

void function2(int *p, float *q)
{
    ...
}
```

file3: **def.h**

```
#define CONST1 80
#define CONST2 100

int function1(int, int);
void function2(int *, float *);
```

- **def.h** contains the **constant definitions** and the **function declarations** for the program.
- If we do not use a header file, these lines have to be in both file1 and file2.

Process of C Program Compilation



Compiling Program with Several Files

- Take the example above:

```
$ gcc –ansi mainF.c support.c –o mainF
```

- The compiler compiles **mainF.c** and produces **mainF.o**, then it compiles **support.c** and produces **support.o**. The linker will produce the executable file **mainF** after linking the two **.o** files and the library functions.
- If, after successful compilation, changes are made to **mainF.c** but not support.c, then

```
$ gcc –ansi mainF.c support.o –o mainF
```

or, changes are made to support.c but not mainF.c, then

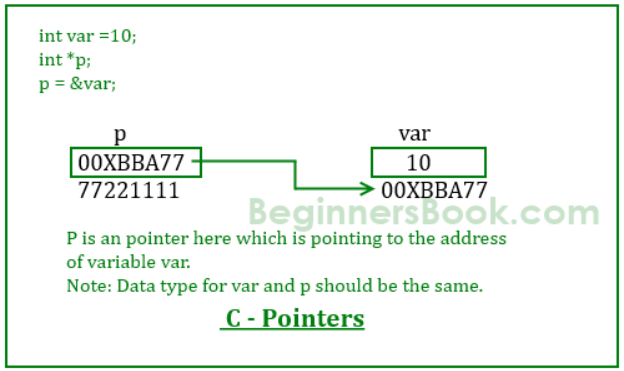
```
$ gcc –ansi mainF.o support.c –o mainF
```

- In the last two situations, no re-compilation is done to the file whose **.o** file is given in the command line.

IV. Address and Pointer

Address of a Variable

- **What is the address in C?**
  - An integer indicating the numerical number of memory storage unit
  - Usually in binary or hexadecimal format



Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00			
90000001	00	sum	int (4 bytes)	000000FF(255_{10})
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1_{10})
90000005	FF			
90000006	1F	averge	double (8 bytes)	1FFFFFFFFFFFFFF ($4.45015E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrSum	int* (4 bytes)	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

Address Operator (&)

```
#include <stdio.h>
int main(void)
{
   int num = 5;

   printf("num = %d, &num = %p\n", num, &num);
   scanf("%d", &num);
   printf("num = %d, &num = %p\n", num, &num);
   return 0;
}
```

This value is just for **illustration**, and may be different for another run.

Output:

num = 5, &num = 1024

10

num = 10, &num = 1024

Pointer Variables

- We may have variables which store the addresses of memory locations of some data objects. These variables are called **pointers**.
- A **pointer variable** is declared by **`dataType *pointerName`**, for example:

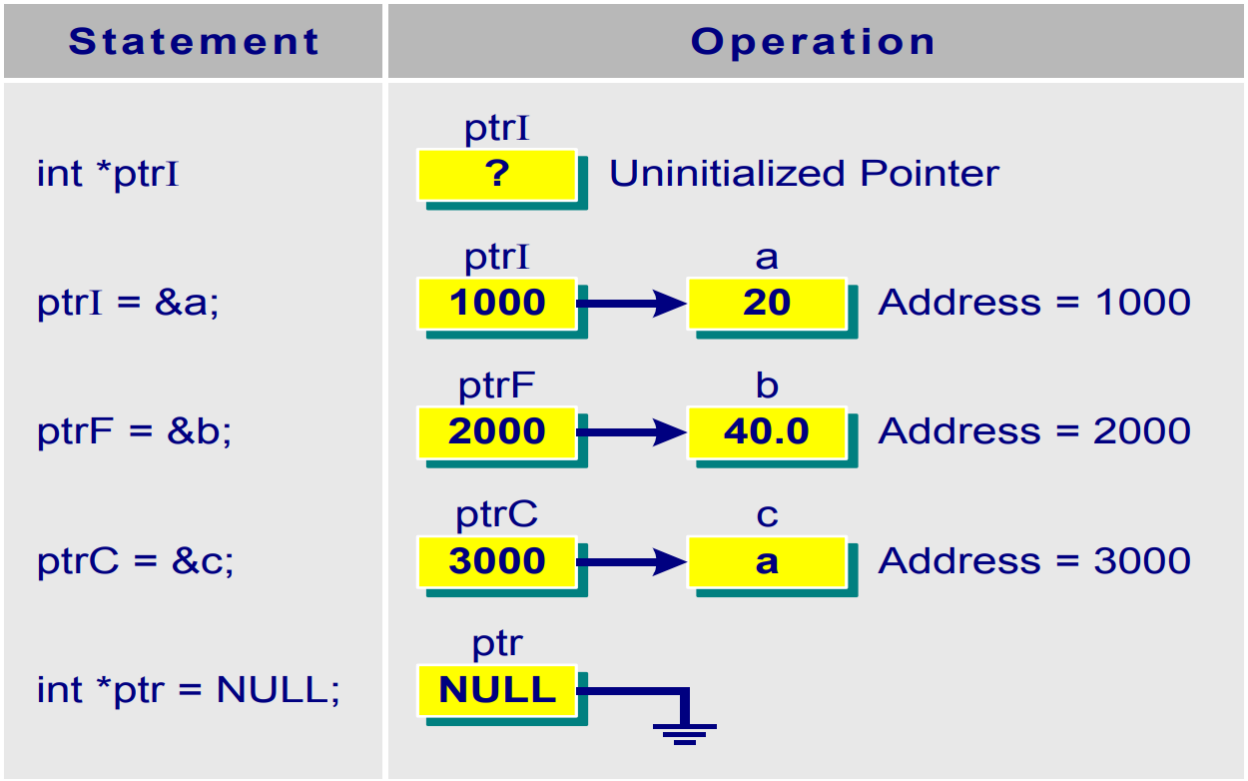
`int *ptrI;`	/* Variable `ptrI` is a pointer. It stores the address of a memory location for an **integer** */
`float *ptrF;`	/* Variable `ptrF` is a pointer. It stores the address of a memory location for a **float** */
`char *ptrC;`	/* Variable `ptrC` is a pointer. It stores the address of a memory location for a **char** */

- The **value** of a pointer variable is an **address**.

Pointer Variables

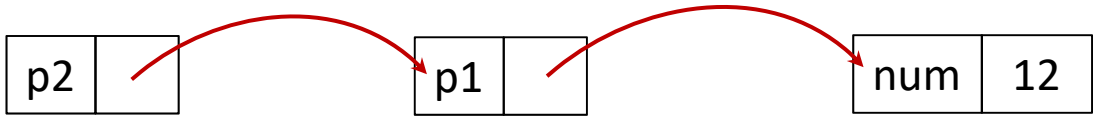
Example:

```
int a = 20; float b = 40.0; char c = ‘a’;
int *ptrI; float *ptrF; char *ptrC;
ptrI = &a; ptrF = &b; ptrC = &c;
```



Pointer To Pointer

- A pointer storing the value of another pointer




```
#include <stdio.h>

int main(void)
{
   float num=12;
   float* p1=&num;
   float** p2=&p1;

   return 0;
}
```

Indirection Operators (*)

- The **content** of the memory location pointed to by a pointer variable is referred to by using the **indirection operator** ***.

- If a pointer variable is defined as **ptr**, we use the expression ***ptr** to *dereference* the pointer to obtain the value stored at the address pointed to by the pointer **ptr**.

Indirection Operator – Example 1

```
#include <stdio.h>
int main(void)
{
   int num = 3;
   int *ptr;

   ptr = &num;
   printf("num = %d, &num = %p\n", num, &num);
   printf("ptr = %p, *ptr = %d\n", ptr, *ptr);
   *ptr = 10;
   printf("num = %d, &num = %p\n", num, &num);
   return 0;
}
```

Statement	Operation
ptr = #	ptr 1024 → num 3 Address = 1024
*ptr = 10;	ptr 1024 → num 10 Address = 1024

Output:

num = 3, &num = 1024

ptr = 1024, *ptr = 3

num = 10, &num = 1024

Indirection Operator – Example 2

```
/* example to show the use of pointers */
#include <stdio.h>

int main(void)
{
    int num1 = 3, num2 = 5;
    int *ptr1, *ptr2;

    ptr1 = &num1; // put the address of num1 into ptr1
    printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);

    (*ptr1)++; /* increment by 1 the content of the
                  memory location pointed to by ptr1 */
    printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);

    ptr2 = &num2; // put the address of num2 into ptr2
    printf("num2 = %d, *ptr2 = %d\n", num2, *ptr2);
```

Code continues in next slide …

Output:

num1 = 3, *ptr1 = 3
num1 = 4, *ptr1 = 4
num2 = 5, *ptr2 = 5

```
    *ptr2 = *ptr1; /* copy the content of the location
                      pointed to by ptr1 into the
                      location pointed to by ptr2 */
    printf("num2 = %d, *ptr2 = %d\n", num2, *ptr2);

    *ptr2 = 10;  /* 10 is copied into the location
                    pointed to by ptr2 */
    num1 = *ptr2; /* copy the content of the memory
                     location pointed to by ptr2
                     into num1 */
    printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);

    *ptr1 = *ptr1 * 5;
    printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);

    ptr2 = ptr1; // address in ptr1 copied into ptr2
    printf("num2 = %d, *ptr2 = %d\n", num2, *ptr2);

    return 0;
}
```

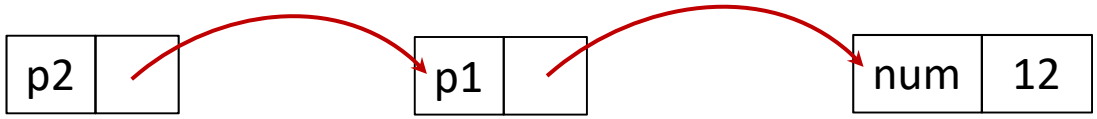
Output:

num2 = 4, *ptr2 = 4
num1 = 10, *ptr1 = 10
num1 = 50, *ptr1 = 50
num2 = 10, *ptr2 = 50

Indirection Operator – Example 2

Statement	num1 (addr = 1024)	num2 (addr = 2048)	ptr1	ptr2
int num1 = 3, num2 = 5;	3	5		
int *ptr1, *ptr2;	3	5	?	?
ptr1 = &num1;	3	5	1024	?
(*ptr1)++;	4	5	1024	?
ptr2 = &num2;	4	5	1024	2048
*ptr2 = *ptr1;	4	4	1024	2048
*ptr2 = 10;	4	10	1024	2048
num1 = *ptr2;	10	10	1024	2048
*ptr1 = *ptr1 * 5;	50	10	1024	2048
ptr2 = ptr1;	50	10	1024	1024

Multiple Indirection




```
#include <stdio.h>

int main(void)
{
   float num=12;
   float* p1=&num;
   float** p2=&p1;

   printf("the content with indirection once: %f", *p1);
   printf("the content with indirection twice: %f", *(*p2));

   return 0;
}
```

How function is called?

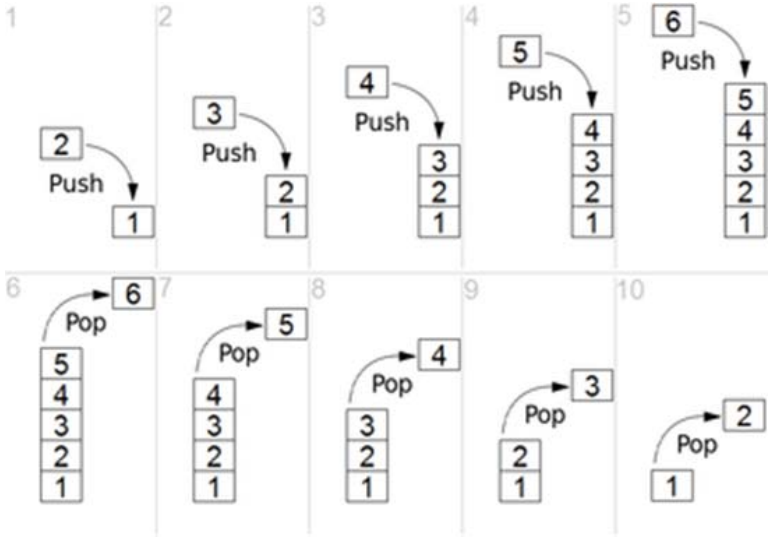
- **Entry point**
  - the first instruction a program is executed
  - In C/C++, the main() function

```
int main(void);
int main();

int main(int argc, char **argv);
```

  - Loader of operating system will load the program into memory, giving the entry point a specific address
  - This marks the transition from load time to run time.

How function is called?

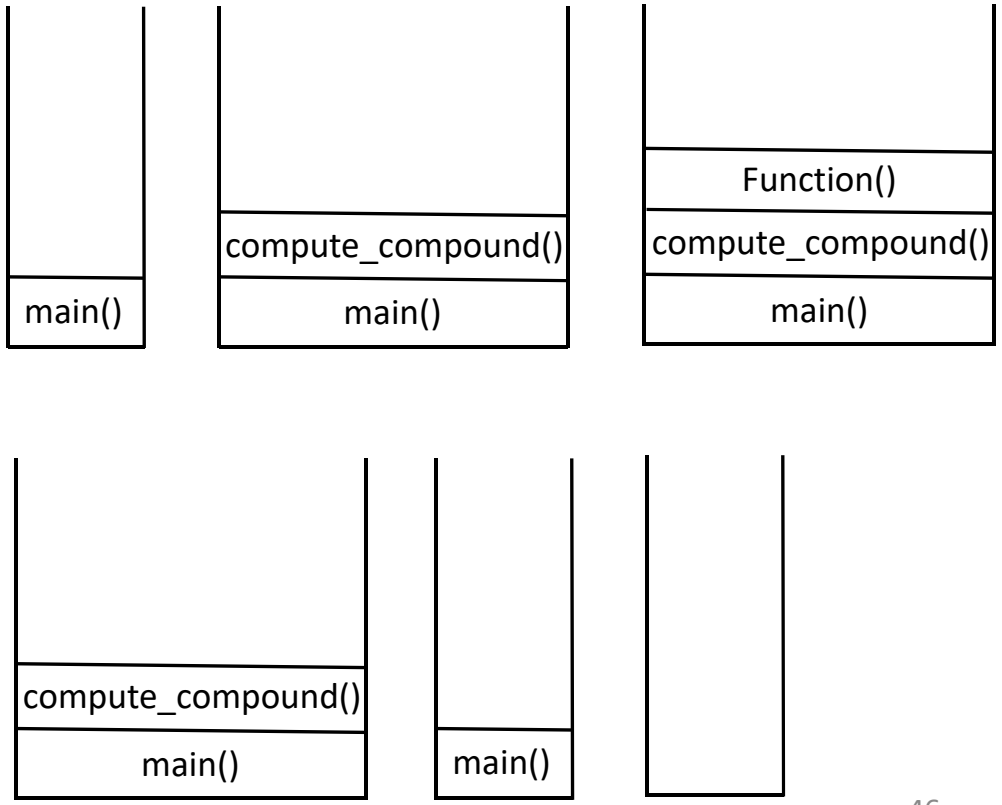
- **Function (call) stack**
  - What is a stack?
    - A data structure to store data in a first-in-last-out order
    - Two operations:
      - push (into the stack) / pop (out of the stack)



How function is called?

- **Function call process**
  - With the aid of function call stack, storing pointers

```
#include <stdio.h>
float compute_compound(float x);
float function(float x);

void main()
{
    float f=compute_compound(10.0);
    printf("the result is: %f\n", f);
    return 0;
}

float compute_compound()
{
    return exp(function(x));
}

float function(float x)
{
    return sin(x)*sin(x);
}
```

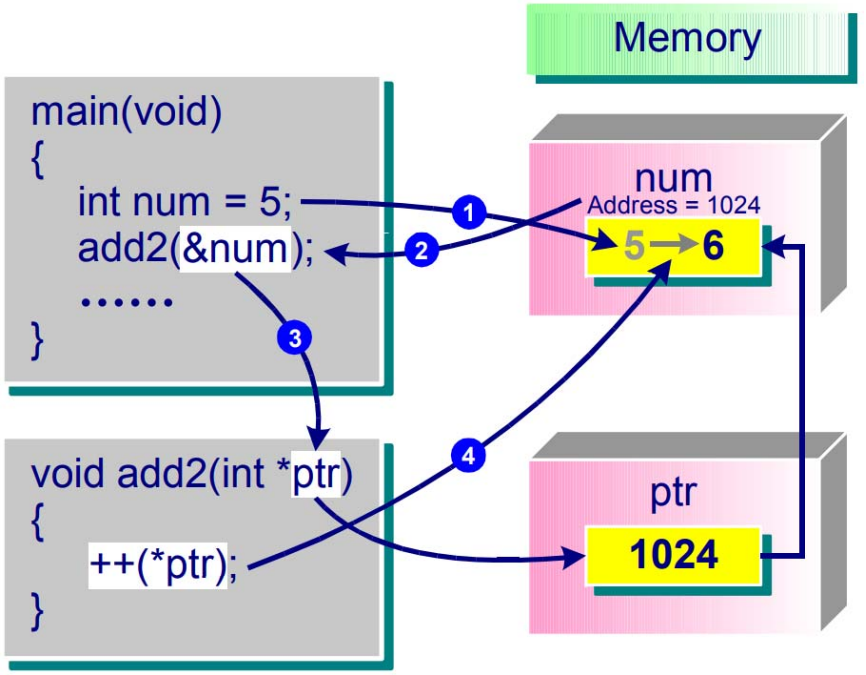


Call by Pointer

- **Parameter passing between functions has two modes:**
  - Call by value
  - Call by pointer
- **Call by value**: The argument of a function has a local copy when it is passed to the function.
- **Call by pointer**: The argument of a function shares the same **address** of the argument variable, no argument copy is performed.
  - Therefore, a change to the value pointed to by the parameter **changes** the argument value (instantly).

Call by Pointer – Example 1

```
#include <stdio.h>
void add2(int *ptr);
int main(void)
{
    int num = 5;
    // passing the address of num
    add2(&num);
    printf("Value of num is: %d",
           num);
    return 0;
}

void add2(int *ptr)
{
    ++(*ptr);
}
```

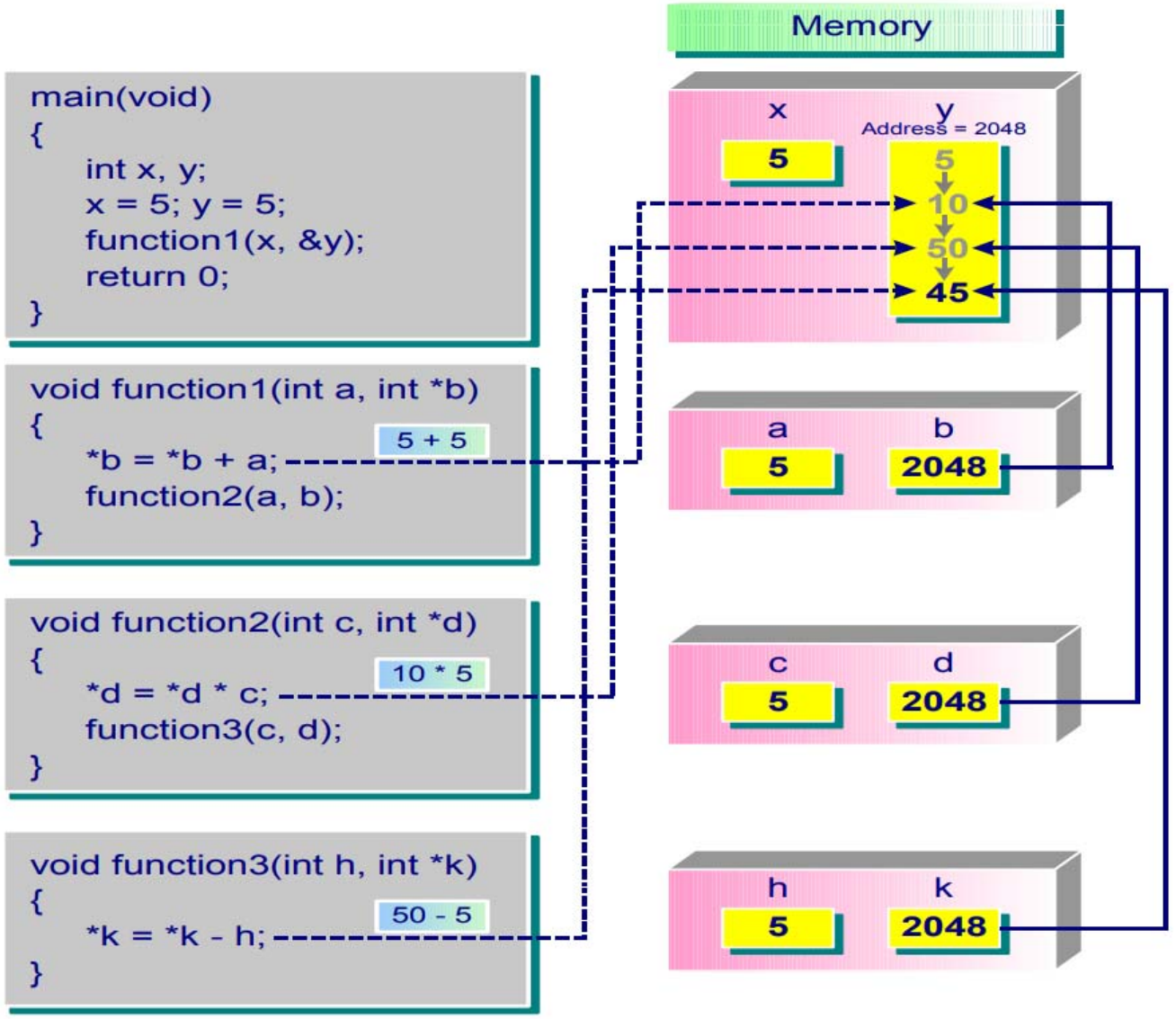


Output:
Value of num is: 6

Call by Pointer – Example 2

```
#include <stdio.h>

void function1(int a, int *b);
void function2(int c, int *d);
void function3(int h, int *k);

int main(void) {
   int x, y;
   x = 5; y = 5;                    /* (i) */
   function1(x, &y);                /* (x) */
   return 0;
}
void function1(int a, int *b) {     /* (ii) */
   *b = *b + a;                     /* (iii) */
   function2(a, b);                 /* (ix) */
}
void function2(int c, int *d) {     /* (iv) */
   *d = *d * c;                     /* (v) */
   function3(c, d);                 /* (viii) */
}
void function3(int h, int *k) {     /* (vi) */
   *k = *k – h;                     /* (vii) */
}
```

Call by Pointer – Example 2



Call by Pointer – Example 2

	x	y	a	*b	c	*d	h	*k	remarks
(i)	5	5	-	-	-	-	-	-	
(ii)	5	5	5	5	-	-	-	-	
(iii)	5	10	5	10	-	-	-	-	
(iv)	5	10	5	10	5	10	-	-	
(v)	5	50	5	50	5	50	-	-	
(vi)	5	50	5	50	5	50	5	50	
(vii)	5	45	5	45	5	45	5	45	
(viii)	5	45	5	45	5	45	-	-	
(ix)	5	45	5	45	-	-	-	-	
(x)	5	45	-	-	-	-	-	-	

When to Use "Call by Pointer"?

- When you need to pass more than one value **back** from a function.

- When using *call by value,* it results in a large piece of information being **copied** to the local memory, e.g. passing large arrays.
  - In such cases, for the sake of **efficiency**, we'd better use call by reference.

Call by Reference

- **Call by reference**: The argument of a function is another name of the same variable, no argument copy is performed.
  - Therefore, a change to the value of the parameter **changes** the argument value (instantly).
  - Declaration of a reference variable

```
float a=10.0;
float& b=a; //reference variable should be initialized.

b=15;
printf("the value of a is: %f\n",a);
```

Example: multiple function return values

- Get the area and circumference of a circle

```
#include <stdio.h>
#define PI 3.1415926

void GetCircleInfo(float R, float *area, float *circum);

int main(void) {                                    /*Using pointer implementation*/
  float R=0, area=0, circum=0;
  scanf("Input circle radius: %f", &R);

  GetCircleInfo(R, &area, &circum);
  printf("The circle area is :%f\n", area);
  printf("The circle circumference is :%f\n", area);

  return 0;
}

void GetCircleInfo(float R, float *area, float *circum){
   *area=PI*R*R;
   *circum=2*PI*R;
}
```

Example: multiple function return values

- Get the area and circumference of a circle

```
#include <stdio.h>
#define PI 3.1415926

void GetCircleInfo(float R, float &area, float &circum);

int main(void) {                                    /*Using reference implementation*/
  float R=0, area=0, circum=0;
  scanf("Input circle radius: %f", &R);

  GetCircleInfo(R, area, circum);
  printf("The circle area is :%f\n", area);
  printf("The circle circumference is :%f\n", area);

  return 0;
}

void GetCircleInfo(float R, float *area, float *circum){
   area=PI*R*R;
   circum=2*PI*R;
}
```

Function Pointers

- **A subroutine pointer or procedure pointer**
  - A pointer that points to a function
  - Points to executable code within memory

```c
#include <stdio.h>  /* for printf */
#include <string.h> /* for strchr */

double cm_to_inches(double cm) {
    return cm / 2.54;
}

// "strchr" is part of the C string handling (i.e., no need for declaration)
// See https://en.wikipedia.org/wiki/C_string_handling#Functions

int main(void) {
    double (*func1)(double) = cm_to_inches;
    char * (*func2)(const char *, int) = strchr;
    printf("%f %s", func1(15.0), func2("Wikipedia", 'p'));
    /* prints "5.905512 pedia" */
    return 0;
}
```

Function Pointer as Function Parameter

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

// Function taking a function pointer as an argument
double compute_sum(double (*funcp)(double), double lo, double hi) {
    double sum = 0.0;

    // Add values returned by the pointed-to function '*funcp'
    int i;
    for(i = 0;  i <= 100;  i++) {
        // Use the function pointer 'funcp' to invoke the function
        double x = i / 100.0 * (hi - lo) + lo;
        double y = funcp(x);
        sum += y;
    }
    return sum / 101.0;
}

double square(double x) {
     return x * x;
}

```

```c
int main(void) {
    double  sum;

    // Use standard library function 'sin()' as the pointed-to function
    sum = compute_sum(sin, 0.0, 1.0);
    printf("sum(sin): %g\n", sum);

    // Use standard library function 'cos()' as the pointed-to function
    sum = compute_sum(cos, 0.0, 1.0);
    printf("sum(cos): %g\n", sum);

    // Use user-defined function 'square()' as the pointed-to function
    sum = compute_sum(square, 0.0, 1.0);
    printf("sum(square): %g\n", sum);

    return 0;
}
```