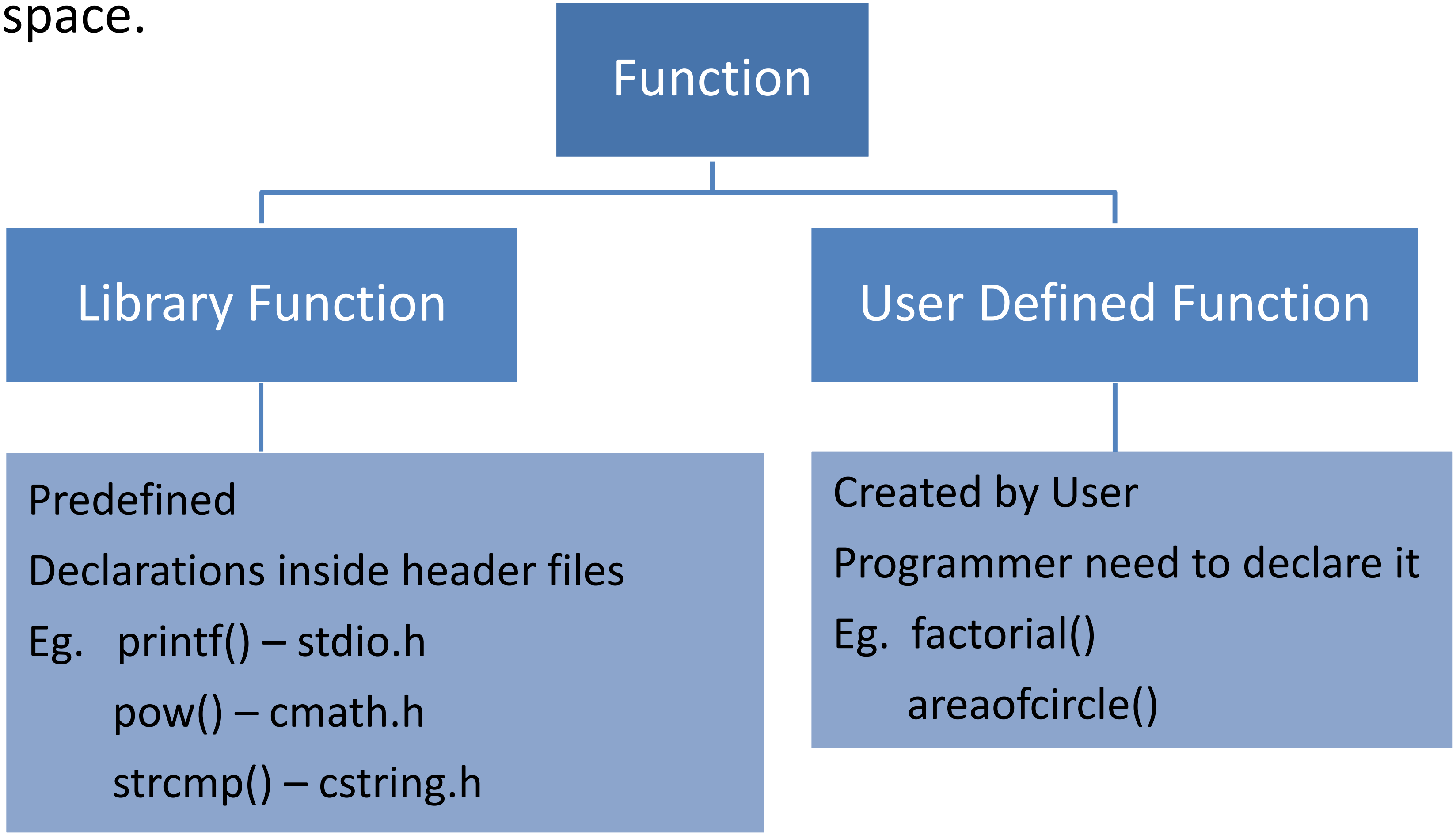
C++ Functions

C++ Function

- A **function** is a group of statements that together perform a task.
- Functions are made for code reusability and for saving time and space.



Function

Library Function

Predefined

Declarations inside header files

Eg. printf() – stdio.h

pow() – cmath.h

strcmp() – cstring.h

User Defined Function

Created by User

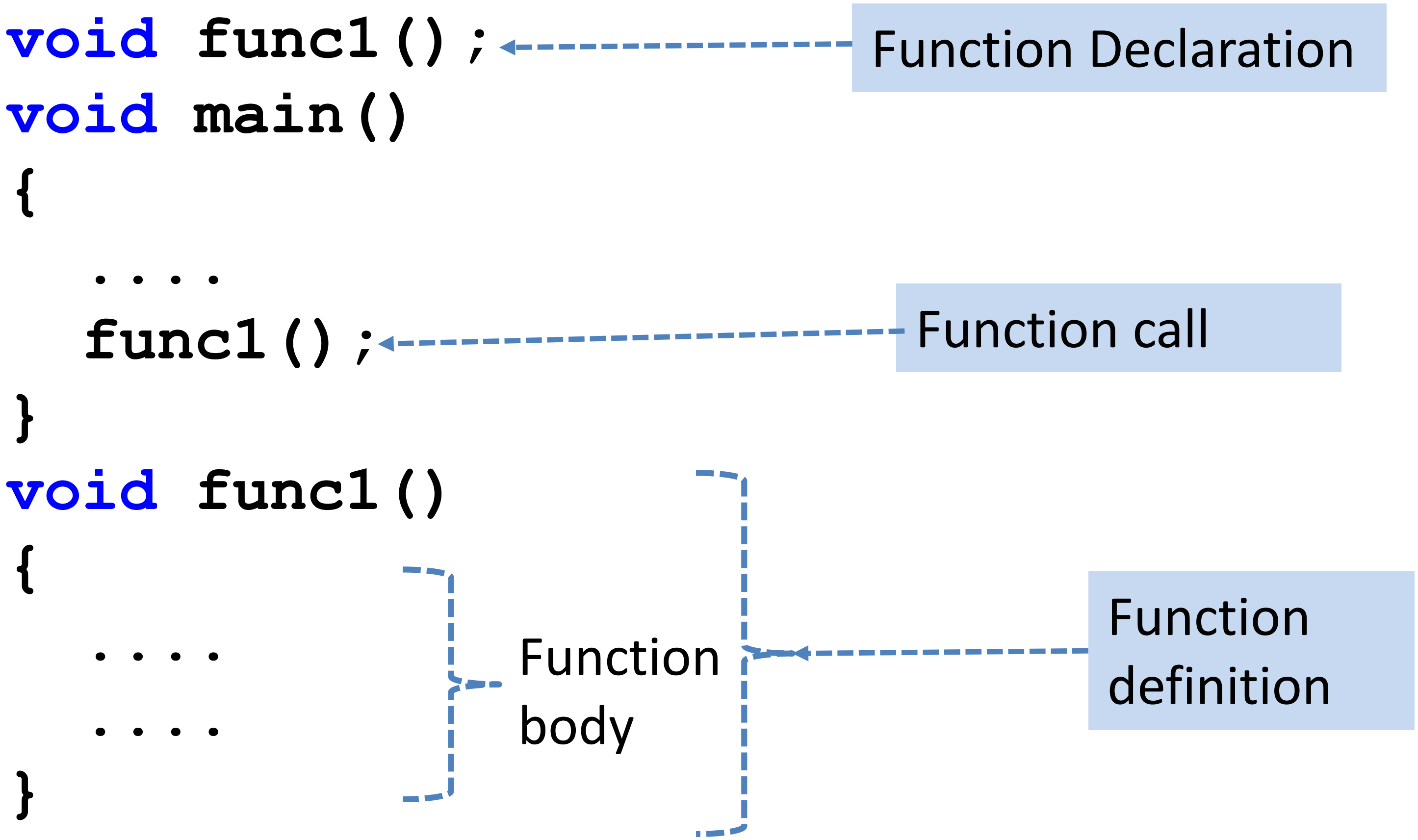
Programmer need to declare it

Eg. factorial()

areaofcircle()

C++ Function – (Cont…)

- There are three elements of user defined function



```
void func1();
void main()
{
    ....
    func1();
}
void func1()
{
    ....
    ....
}
```

Function Declaration

Function call

Function body

Function definition

Simple Function – (Cont…)

- **Function Declaration**

  Syntax:

  ```
  return-type function-name (arg-1, arg 2, …);
  ```

  Example: `int addition(int , int );`

- **Function Definition**

  Syntax:

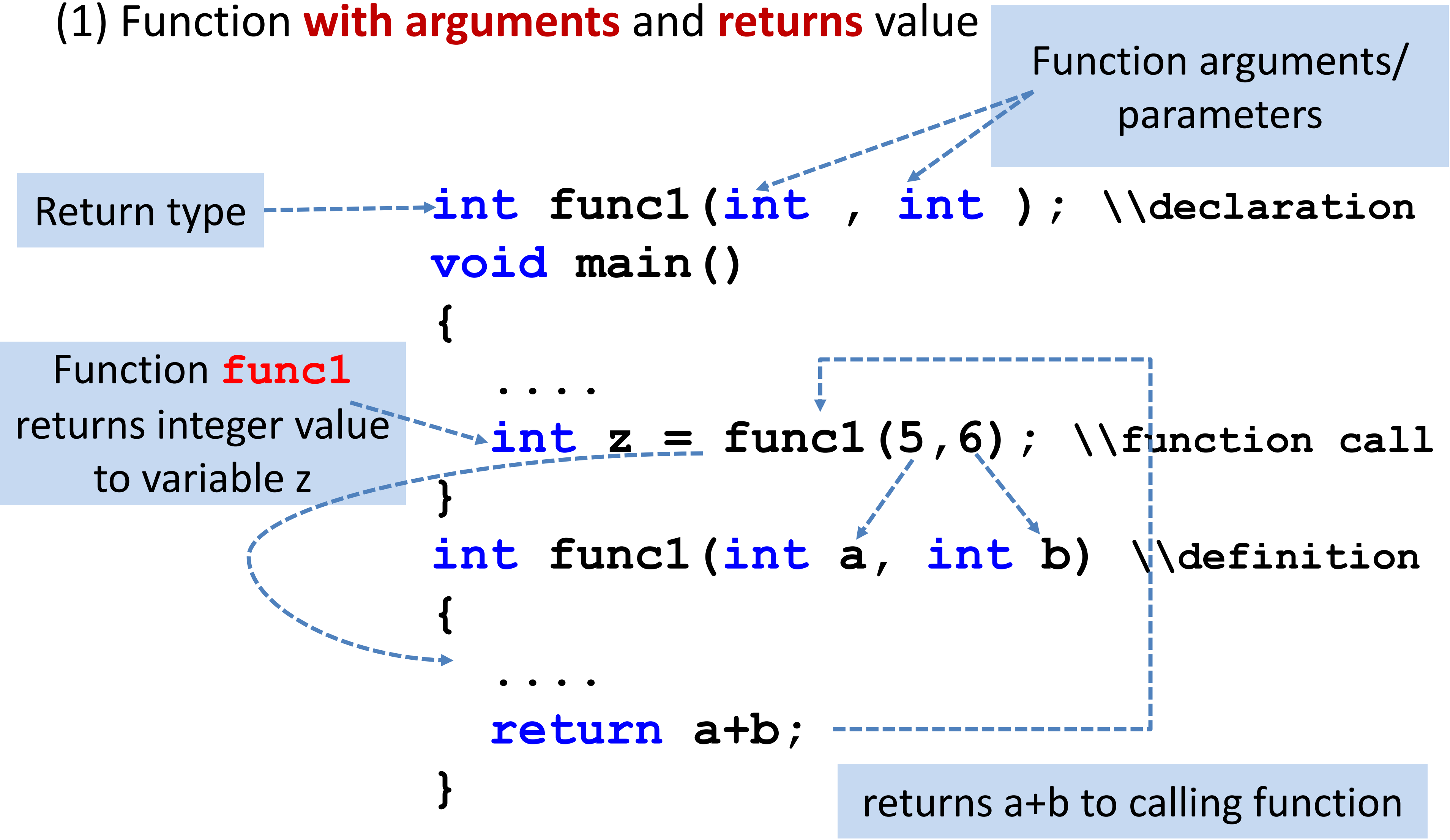
  ```
  return-type function-name (arg-1, arg 2, …)
  {
       ... Function body
  }
  ```

  Example:

  ```
  int addition(int x, int y)
  {
      return x+y;
  }
  ```

Categories of Function

(1) Function **with arguments** and **returns** value



Function arguments/ parameters

Return type

```
int func1(int , int ); \\declaration
void main()
{
   ....
   int z = func1(5,6); \\function call
}
int func1(int a, int b) \\definition
{
   ....
   return a+b;
}
```

Function **func1** returns integer value to variable z

returns a+b to calling function

Categories of Function (Cont…)

(2) Function **with arguments** but **no return** value

```
void func1(int , int ); \\function declaration
void main()
{
  ....
  func1(5,6); \\function call
}
void func1(int a, int b) \\function definition
{
  ....
  ....
}
```

Categories of Function (Cont..)

(3) Function with **no argument** but **returns** value

```
int func1();
void main()
{
    ....
    int z = func1();
}
int func1()
{
    ....
    return 99;
}
```

Categories of Function (Cont…)

(4) Function **with no argument** and **no return** value

```
void func1();
void main()
{
  ....
    func1();
}
void func1()
{
  ....
  ....
}
```

Program: Categories of function

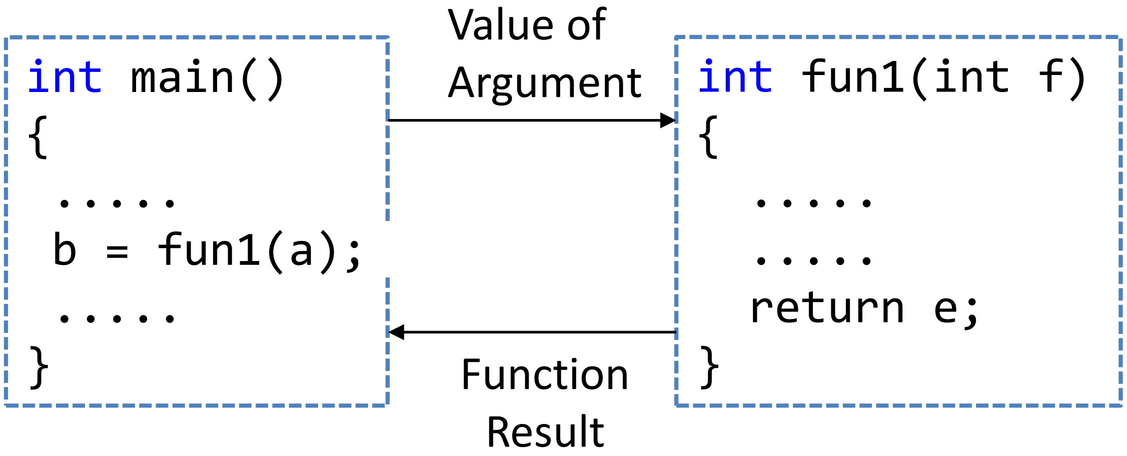
- Write C++ programs to demonstrate various categories of function, Create function **addition** for all categories.

Function with argument and returns value

```
#include <iostream>
using namespace std;

int add(int, int);

int main(){
  int a=5,b=6,ans;
  ans = add(a,b);
  cout<<"Addition is="<<ans;
  return 0;
}
int add(int x,int y)
{
  return x+y;
}
```



```
int main()
{
 .....
 b = fun1(a);
 .....
}
```

Value of Argument

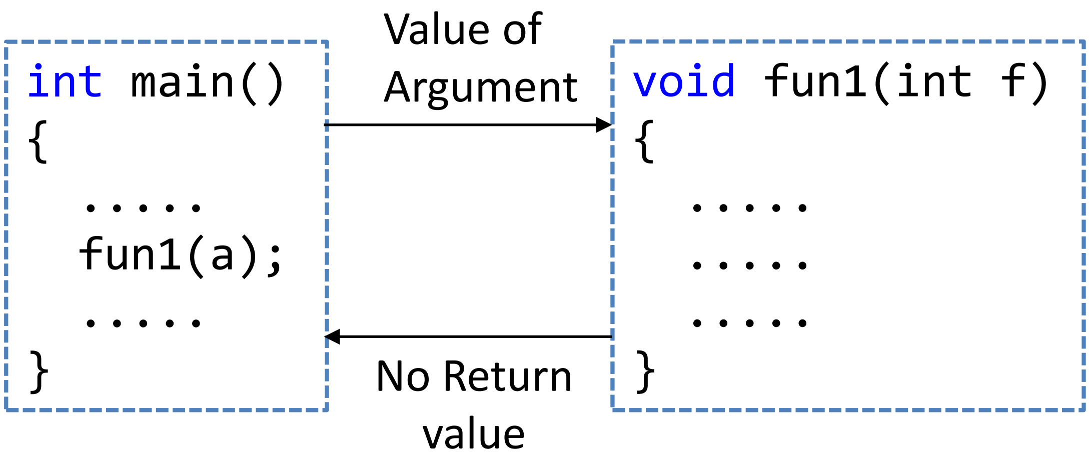
Function Result

```
int fun1(int f)
{
   .....
   .....
   return e;
}
```

Function with arguments but no return value

```cpp
#include <iostream>
using namespace std;

void add(int, int);
int main()
{
  int a=5,b=6;
  add(a,b);
  return 0;
}
void add(int x,int y)
{
 cout<<"Addition is="<<x+y;
}
```



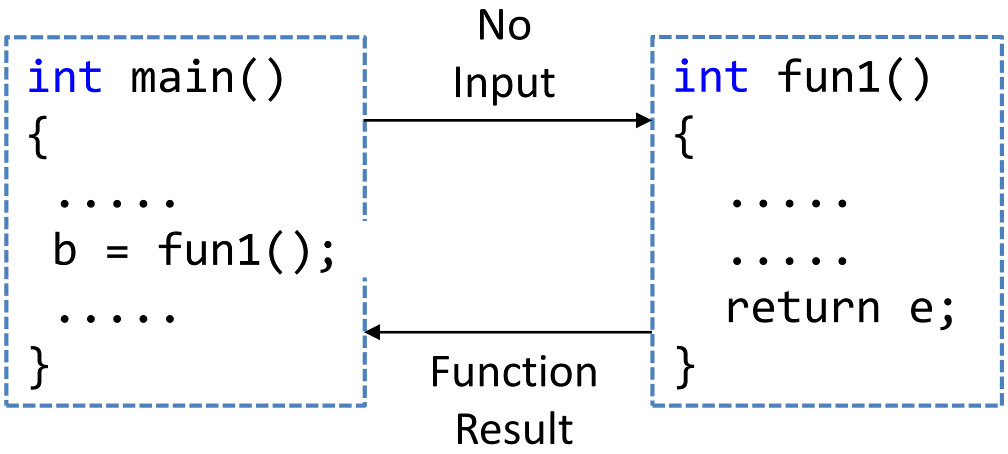
```cpp
int main()
{
   .....
   fun1(a);
   .....
}
```

```cpp
void fun1(int f)
{
   .....
   .....
   .....
}
```

Function with no argument but returns value

```
int add();

int main()
{
   int ans;
   ans = add();
   cout<<"Addition is="<<ans;
   return 0;
}
void add()
{
   int a=5,b=6;
   return a+b;
}
```



```
int main()
{
 .....
 b = fun1();
 .....
}
```

No Input

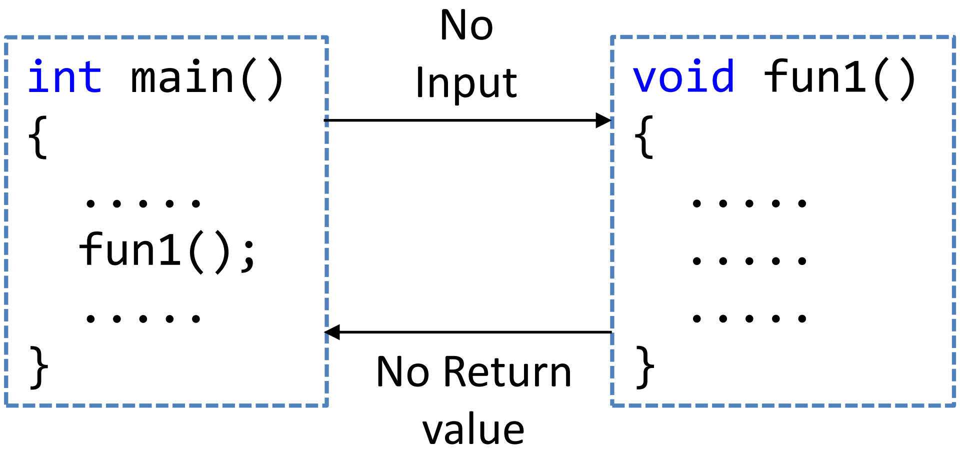
```
int fun1()
{
   .....
   .....
   return e;
}
```

Function Result

Function with no argument and no return value

```
void add();

int main()
{
   add();
   return 0;
}
void add()
{
   int a=5,b=6;
   cout<<"Addition is="<<a+b;
}
```



```
int main()
{
   .....
   fun1();
   .....
}
```

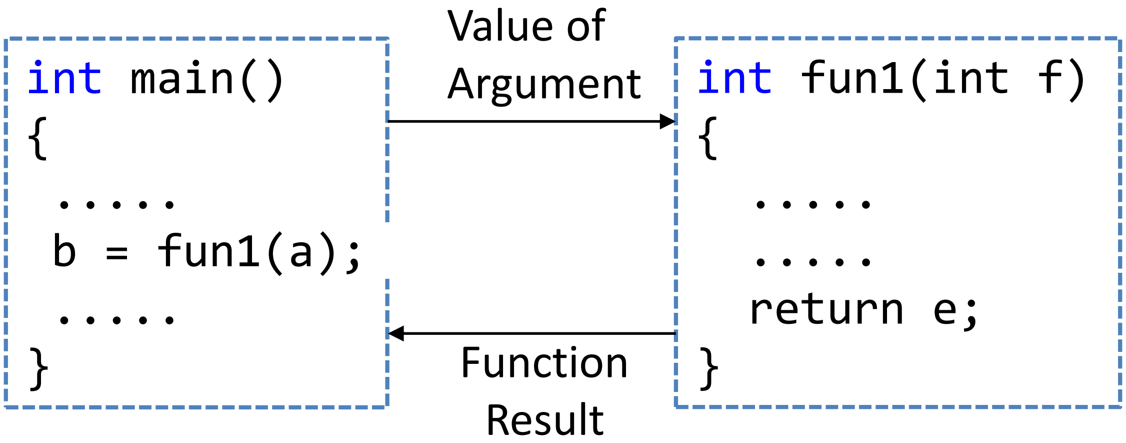
No
Input

No Return
value

```
void fun1()
{
   .....
   .....
   .....
}
```

Categories of Functions Summary

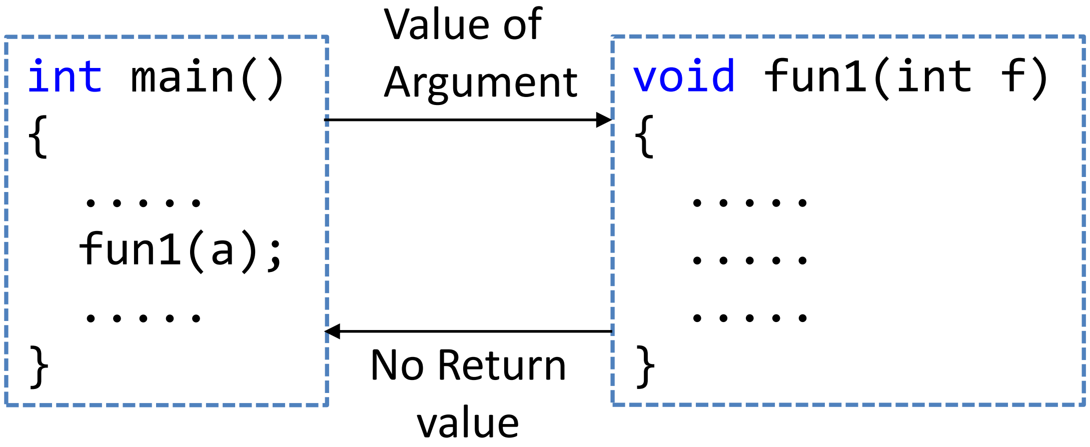
(1) Function with argument and returns value



```
int main()
{
 .....
 b = fun1(a);
 .....
}
```

```
int fun1(int f)
{
   .....
   .....
   return e;
}
```

(2) Function with argument and but no return value

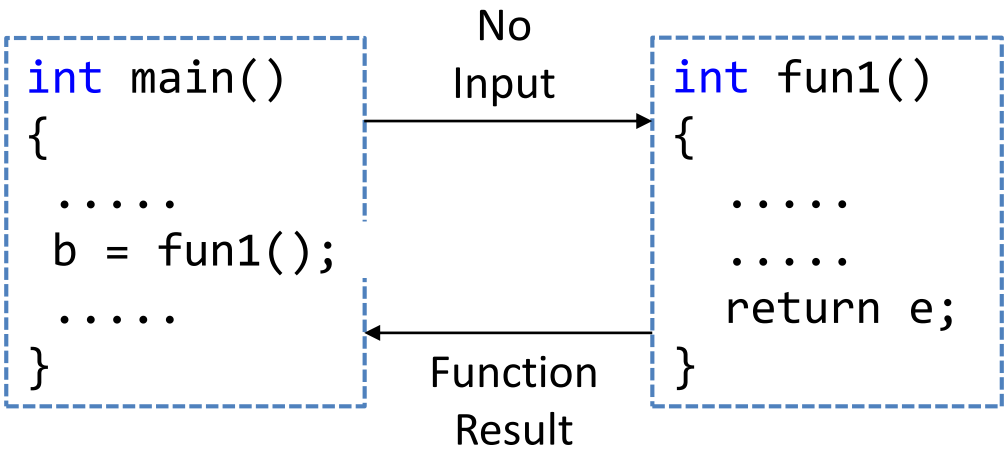


```
int main()
{
   .....
   fun1(a);
   .....
}
```

```
void fun1(int f)
{
   .....
   .....
   .....
}
```

Categories of Functions Summary

(3) Function with no argument and returns value



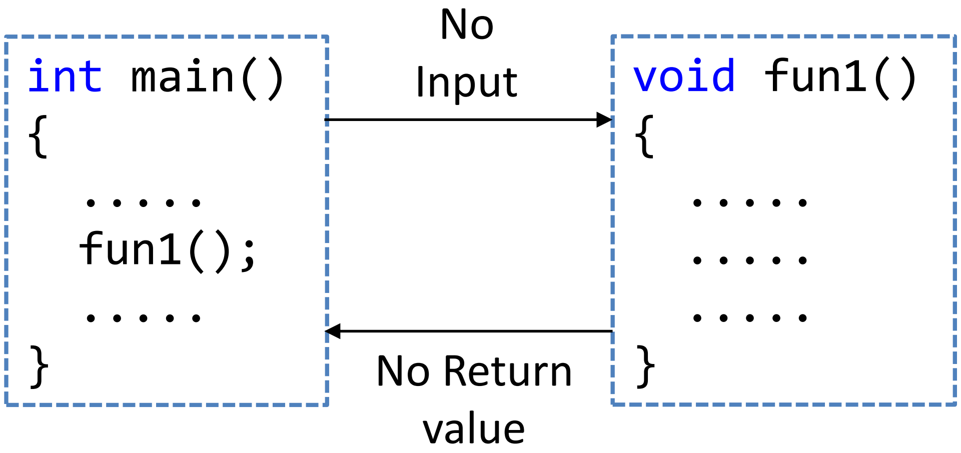
```
int main()
{
  .....
 b = fun1();
  .....
}
```

No Input

```
int fun1()
{
   .....
   .....
   return e;
}
```

Function Result

(4) Function with no argument and but no return value



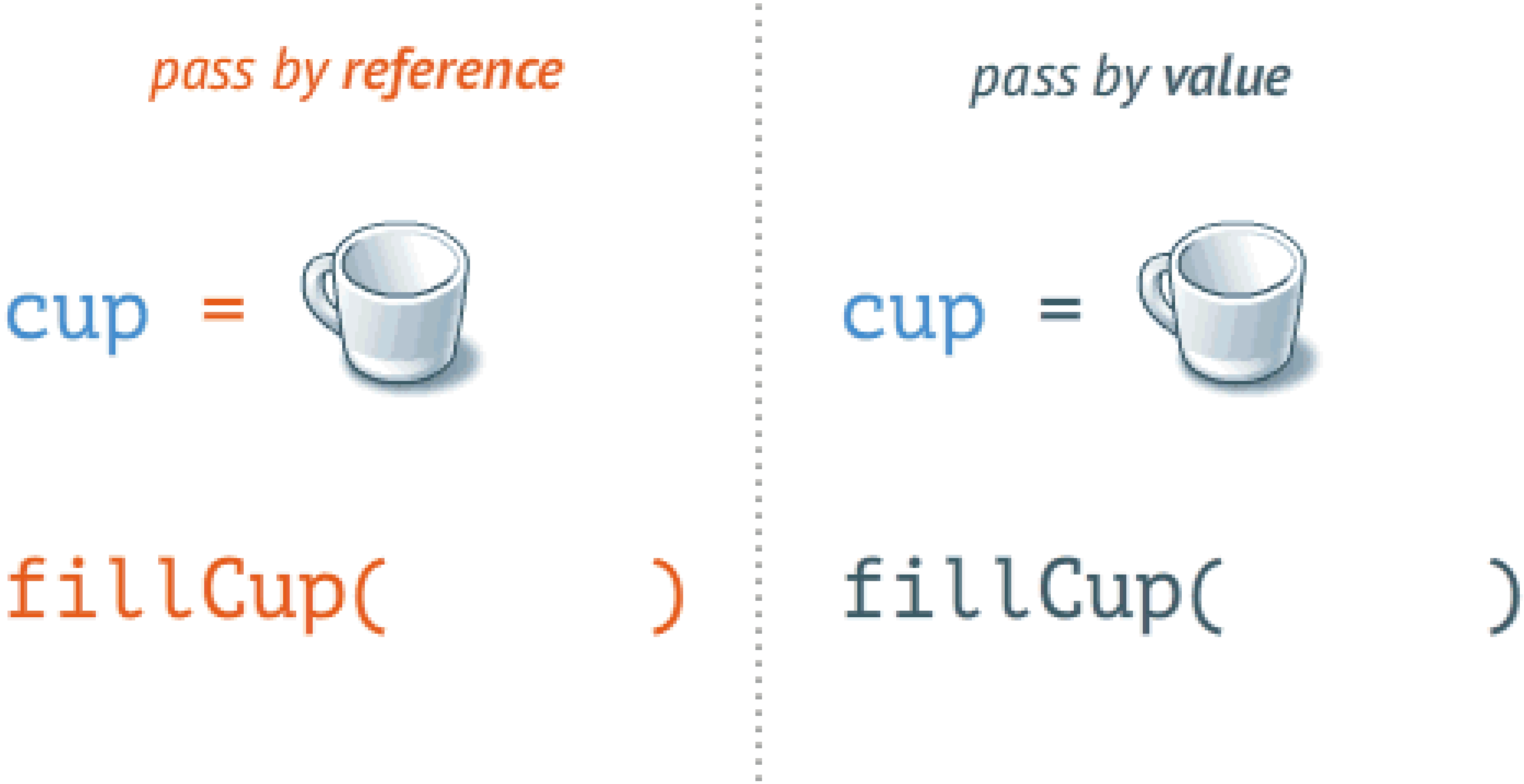
```
int main()
{
   .....
   fun1();
   .....
}
```

No Input

```
void fun1()
{
   .....
   .....
   .....
}
```

No Return value

Call by Reference



Call by reference

- The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter.
- Inside the function body, the reference is used to access the actual argument used in the call.

Actual Parameters

```
int main(){
      add(a,b);
}
```

Formal Parameters

```
void add(int x,int y){
      cout << x+y;
}
```

Note:

- *Actual parameters* are parameters as they appear in function calls.
- *Formal parameters* are parameters as they appear in function declarations / definition.

Program: Swap using pointer, reference

- Write a C++ program that to swap two values using function
  1. With pass by pointer
  2. With pass by reference

Program: Solution

```
void swapptr(int *x, int *y)
{
   int z = *x;
   *x=*y;
   *y=z;
}
void swapref(int &x, int &y)
{
   int z = x;
   x = y;
   y = z;
}
```

- Pointers as arguments
- References as arguments

```
int main()
{
   ...
   swapptr(&a,&b);
   swapref(a,b);
   ...
}
```

Program: Solution

```
void swapptr(int *, int *);
void swapref(int &, int &);
int main()
{
  int a = 45;
  int b = 35;
  cout<<"Before Swap\n";
  cout<<"a="<<a<<" b="<<b<<"\n";

  swapptr(&a,&b);
  cout<<"After Swap with pass by pointer\n";
  cout<<"a="<<a<<" b="<<b<<"\n";

  swapref(a,b);
  cout<<"After Swap with pass by reference\n";
  cout<<"a="<<a<<" b="<<b<<"\n";
}
```

Program: Solution (Cont…)

```
void swapptr(int *x, int *y)
{
   int z = *x;
   *x=*y;
   *y=z;
}
void swapref(int &x, int &y)
{
   int z = x;
   x = y;
   y = z;
}
```

```
OUTPUT
Before Swap
a=45 b=35
After Swap with pass by pointer
a=35 b=45
After Swap with pass by reference
a=45 b=35
```

Program: Return by Reference

- Write a C++ program to **return reference** of maximum of two numbers from function max.

Program: Solution

```
int& max(int &, int &);
int main()
{
   int a=5,b=6,ans;
   ans = max(a,b);
   cout<<"Maximum="<<ans;
}
int& max(int &x,int &y)
{
   if (x>y)
      return x;
   else
      return y;
}
```

- Function declaration returning reference

Program: Returning Reference

```
int x;
int& setdata();
int main()
{
    setdata() = 56;
    cout<<"Value="<<x;
    return 0;
}
int& setdata()
{
    return x;
}
```

- setx() is declared with a reference type,

  `int&`

  as the return type:
- `int& setx();`

  This function contains

  `return x;`
- You can put a call to this function on the left side of the equal sign:

  `setx() = 92;`
- The result is that the variable returned by the function is assigned the value on the right side of the equal sign.

```
#include <iostream>
using namespace std;
double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};
double& setValues( int i ) {
  return vals[i];   // return a reference to the ith element
}
// main function to call above defined function.
int main () {
  cout << "Value before change" << endl;
  for ( int i = 0; i < 5; i++ ) {
    cout << "vals[" << i << "] = ";
    cout << vals[i] << endl;
  }
  setValues(1) = 20.23; // change 2nd element
  setValues(3) = 70.8;  // change 4th element
  cout << "Value after change" << endl;
  for ( int i = 0; i < 5; i++ ) {
    cout << "vals[" << i << "] = ";
    cout << vals[i] << endl;
  }
  return 0;
}
```

```
Value before change
vals[0] = 10.1
vals[1] = 12.6
vals[2] = 33.1
vals[3] = 24.1
vals[4] = 50
Value after change
vals[0] = 10.1
vals[1] = 20.23
vals[2] = 33.1
vals[3] = 70.8
vals[4] = 50
```

C Preprocessors Macros

C Preprocessors Macros

- C **Preprocessor** is a text substitution in program.
- It instructs the compiler to do pre-processing before the actual compilation.
- All **preprocessor** commands begin with a hash symbol (#).

C Preprocessor Macro Example

```c
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)
int main()
{
   int radius;
   float area;
   printf("Enter the radius: ");
   scanf("%d", &radius);
   area = circleArea(radius);
   printf("Area = %f", area);
   return 0;
}
```

Preprocessor

- Every time the program encounters circleArea(argument), it is replaced by (3.1415*(argument)*(argument)).

Inline Functions

Inline Functions

- Every time a function is called it takes a lot of extra time to execute series of instructions such as
  1. Jumping to the function
  2. Saving registers
  3. Pushing arguments into stack
  4. Returning to the calling function
- If a function body is small then overhead time is more than actual code execution time so it becomes more time consuming.
- **Preprocessor macros** is a solution to the problem of small functions in C.
- In C++, **inline function** is used to reduce the function call overhead.

Inline Functions (Cont…)

Syntax:

```
inline return-type function-name(parameters)
{
   // function code
}
```

- Add **inline** word before the function definition to convert simple function to inline function.

Example:

```
inline int Max(int x, int y)
{
    if (x>y)
       return x;
    else
       return y;
}
```

Program: Inline function

- Write a C++ program to create inline function that returns cube of given number (i.e **`n=3,  cube=(n*n*n)=27`**).

Program: Solution

```
#include <iostream>
using namespace std;

inline int cube(int s)
{
    return s*s*s;
}
int main()
{
    cout << "The cube of 3 is: " << cube(3);
    return 0;
}
```

- Calls inline function cube with argument 3

Critical situations Inline Functions

- Some of the situations inline expansion may not work

  1) If a **loop**, a **switch** or a **goto** exists in function body.

  2) If function is not returning any value.

  3) If function contains **static variables**.

  4) If function is **recursive**.

Function Overloading

Function Overloading

- Suppose we want to make functions that add 2 values, add 3 values , add 4 values

In C

```
int sum(int a, int b);
int sum(int a, int b, int c);
int sum(int a, int b, int c, int d);
```

Function with same name in a program **is not allowed** in C language

In C++

```
int sum(int a, int b);
int sum(int a, int b, int c);
int sum(int a, int b, int c, int d);
```

Function with same name in a program **is allowed** in C++ language

Function overloading – Cont…

- C++ provides **function overloading** which allows to use multiple functions **sharing the same name** .
- Function overloading is also known as **Function Polymorphism** in OOP.
- It is the practice of declaring the same function with **different signatures**.

- However, the two functions with the same name must differ in at least one of the following,
  a) The **number** of arguments
  b) The **data type** of arguments
  c) The **order** of appearance of arguments

Arguments make the function unique

- **Function overloading** does not depends on return type.

Program: Function overloading

- Write a C++ program to demonstrate function overloading. Create function **display()** with different arguments but same name

Program: Solution (Cont…)

```cpp
void display(int var)
{
   cout << "Integer number: " << var << endl;
}
void display(float var)
{
   cout << "Float number: " << var << endl;
}
void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

Program: Solution

```
int main()
{
    int a = 5; float b = 5.5;
    display(a);
    display(b);
    display(a, b);
    return 0;
}
```

Program: Function overloading

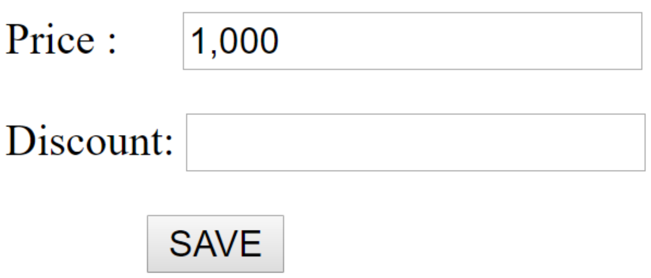
- Write a C++ program to demonstrate function overloading. Create function **`area()`** that calculates area of circle, triangle and box.

Program #7 Solution

```
float area(int r)
{
  return 3.14*r*r;
}
float area(int h, int b)
{
  return 0.5*h*b;
}
float area(int l, int w, int h)
{
  return l*w*h;
}
 int main(){
   cout<<"area of circle="<<area(5);
   cout<<“\n area of triangle="<<area(4,9);
   cout<<“\n area of box="<<area(5,8,2);
   return 0;
 }
```

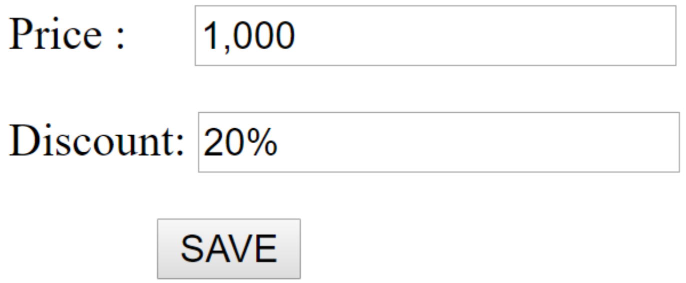
Default Function Arguments

Default Function Argument


Price : 1,000

Discount:

5%


Price : 1,000

Discount: 20%

20%

```
int cubevolume(int l=5, int w=6, int h=7)
{
   return l*w*h;
}
```

```
int main()
{
cubevolume();
cubevolume(9);
cubevolume(15,12);
cubevolume(3,4,7);
}
```

Here, argument can be four types of function calls possible

If, there is not specified then, compiler looks at declaration to see how many arguments a function uses and alert program to use default values

Default Argument Example

```
int volume(int l=5,int w=6, int h=7)
{
   return l*w*h;
}
int main() {
⇨ cout<<"volume="<<volume()<<endl;
⇨ cout<<"volume="<<volume(9)<<endl;
⇨ cout<<"volume="<<volume(15,2)<<endl;
⇨ cout<<"volume="<<volume(3,4,7)<<endl;
   return 0;
}
```

- Function call passing all arguments.
- Explicitly value **3,4,7** passed to **l,w,h** respectively.
- Default value **7** considered for **h** respectively.

Default Arguments

- while invoking a function If the argument/s are not passed then, the default values are used.
- We must add default arguments from right to left.
- We cannot provide a default value to a particular argument in the middle of an argument list.
- Default arguments are useful in situations where some arguments always have the same value.

```
int cubevolume( int l    , int w = 2, int h     )
{
   return l*w*h;
}
```

Default Arguments (Cont…)

- Legal and illegal default arguments

```
 void f(int a, int b, int c=0);   Valid
void f(int a, int b=0, int c=0);   Valid
void f(int a=0, int b, int c=0);  Invalid
void f(int a=0, int b, int c); Invalid
void f(int a=0, int b=0, int c=0);   Valid
```

Common Mistakes

(1)

```
void add(int a, int b = 3, int c, int d = 4);
```

- You cannot miss a default argument in between two arguments.
- In this case, **c** should also be assigned a default value.

(2)

```
void add(int a, int b = 3, int c, int d);
```

- If you want a single default argument, make sure the argument is the last one.

Program: Default Arguments

- Write a C++ program to create function **`sum()`**, that performs addition of 3 integers also demonstrate Default Arguments concept.

Program: Default Arguments

```cpp
#include <iostream>
using namespace std;
int sum(int x, int y=10, int z=20)
{
    return (x+y+z);
}
int main()
{
    cout << "Sum is : " << sum(5) << endl;
    cout << "Sum is : " << sum(5,15) << endl;
    cout << "Sum is : " << sum(5,15,25) << endl;
    return 0;
}
```

Thank You