Functions

Function Parameters

- Parameters are variables and expressions that are used to **pass information to and from functions**.
  - A formal parameter is a variable declared in a function heading.
  - An actual parameter is a variable (or expression) listed in function call.

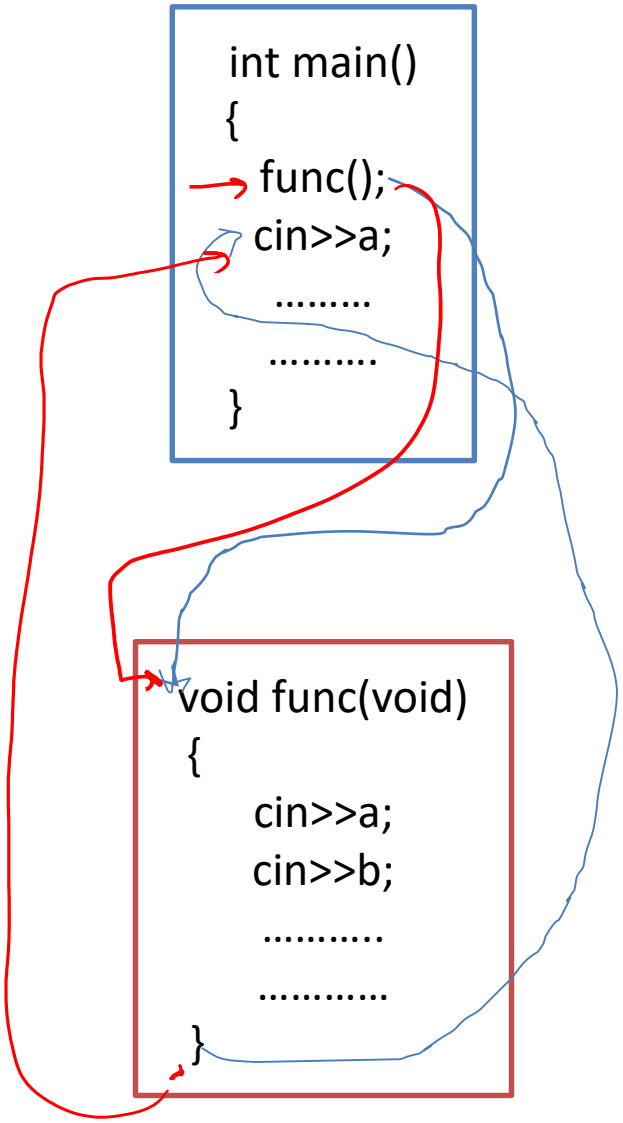
```
int our_formula(int,int); // prototype

int main()
 {
 int a,b,c;
  a = 5;
  b = 7;
  c = our_formula(a,b); // function call
  cout <<"result = " << c;
 }

int our_formula(int x,int y) // function body
{
  int c;
  c = 2 * x + 3 * y;
  return(c);
 }
```

Actual Parameters

Formal Parameters

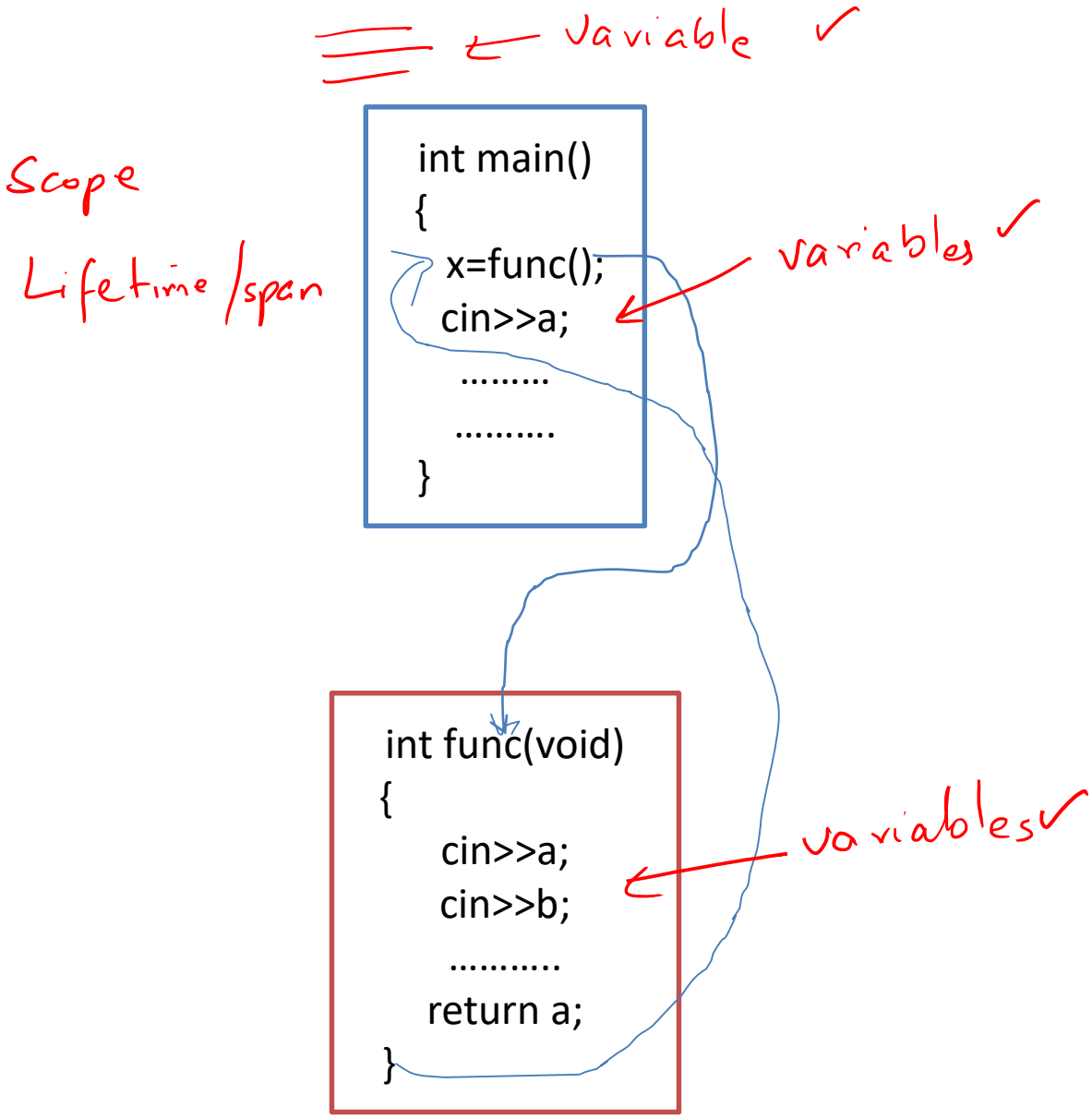
Some General Rules

- The function is given a name and **called** (or invoked) using this name each time the task is to be performed within the program.
- The function that is being called is often referred to as the **called function**.
- The program that calls a function is referred to as the calling program or **calling function**.
- The **execution transfers** from the **calling function** to the **called function.**


int main()
{
func();
cin>>a;
……
…….
}
void func(void)
{
cin>>a;
cin>>b;
……..
……..
}

Some General Rules

- Function name is followed by a number of arguments in brackets ().
- If the called function is returning a value, then it is used as a **right hand side value** of an expression.
- Body of the function must be enclosed within braces {}
- A function can take as many arguments as you want
- A function can only return ONE value!
- When the end brace(**}**) of that called function is reached, **execution returns** to a point immediately after the place at which the function was called.


variable
Scope
Lifetime /span
int main()
{
x=func();
cin>>a;
.....
.....
}
variables
int func(void)
{
cin>>a;
cin>>b;
.....
return a;
}
variables

Function Signatures

- Portion of function prototype that includes the name of the function and the types of its arguments is called the **function signature**.
- Function in the same scope **must** have unique signatures.

Variables and Storage Classes

- The storage class of a variable determines which parts of the program can access it (scope)and how long it stays in existence (lifetime)
- Types of variables
  - Automatic or Local
  - Global or External Variable
  - Static

Scope and Life Time of Variable

- Life Time
  - The time b/w creation and destruction of variable is called the life time or duration. Lifetime of a variable is limited to save memory.
- Scope
  - Scope refers to blocks of code within which variables are visible. Scope limits visibility to improve organization and modularity of the program.

Scope

- The life-time of variables exists within the variable's *scope.*
- After declaration, a variable can be referenced anywhere within its scope.
- Every set of braces has its own scope, and can contain *local variables*.
- The moment the set of braces in which a variable was declared ends, the variable *goes out of scope*, i.e. it can no longer be referenced as an identifier.

Scope

- The program erases variables that have gone out of scope from memory, i.e. their life-time ends.
- The scope of arguments to a function is the entire function body.
- No two variables may share the same name within a scope.

Scope

- *Example:* A variable declared in the first line of a function can be used anywhere in the function, but nowhere outside of it.
- The moment the function exits, the variable ceases to exist in memory, i.e. its scope and lifetime end.

Automatic or Local Variables

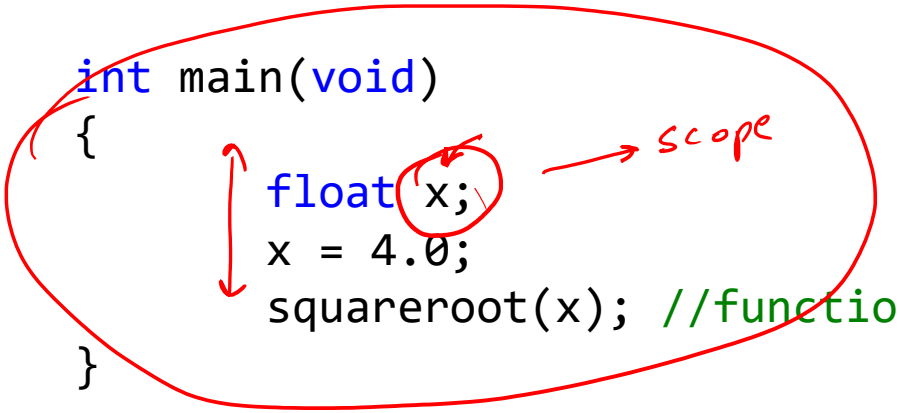
- Variables defined within the function body (or a set of braces) are called automatic or local variables.
- **Properties**
  - It is most commonly used.
  - It is not created until the function in which it is defined is called.

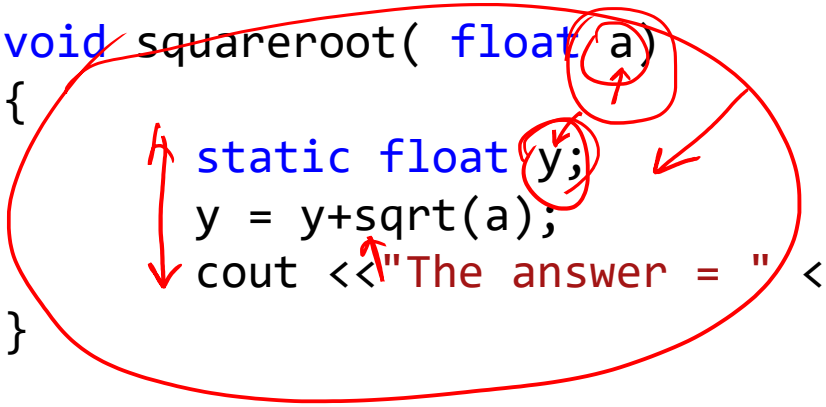
Automatic or Local Variables

- **Life Time**
  - When control is returned to the calling program, these variables are lost. Life time is equal to the life of function.
- **Scope**
  - Automatic variable are only visible with in function, they can only be accessed within the function in which they are created.

Example (Local Variables)

```
#include <math.h>

void squareroot(float);

int main(void)
{
        float x;
        x = 4.0;
        squareroot(x); //function call
}

void squareroot( float a)
{
        static float y;
        y = y+sqrt(a);
        cout <<"The answer = " << y <<endl;
}
```





External or Global Variables

- Variable defined outside the function. Normally it is declared before main function.
- **Properties**
  - Every function can access global variables.
  - Thus, any function can change the values of global variables.
  - This creates organizational problems

Global Variables

- Global variables should generally be avoided unless necessary.
- It is a good practice to use *::globalvariable_name* for using a global variable.
- *eg, if 'a' is the global variable, write '::a' where-ever 'a' is used in the code*

Global Variables

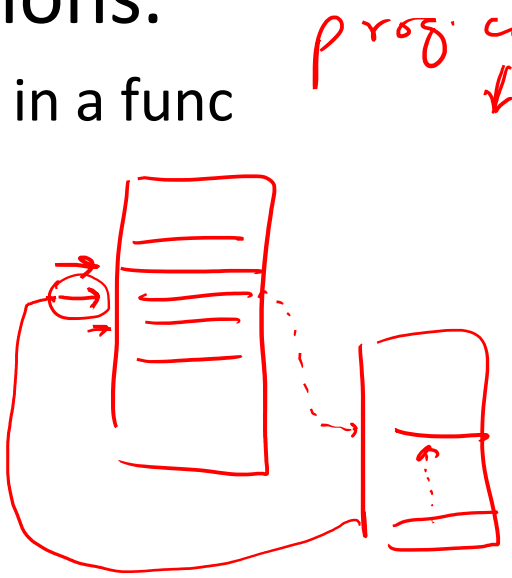
- **Life Time**
  - is equal to the life of program.
- **Scope**
  - It is visible to all the function in a program.

Example (Global Variables)

```
float sum(); // prototype
float first, second;

int main(void)
{
        ::first = 23.45;
        ::second = 87.555;
        cout <<"sum = " << sum();
        return 0;
}

float sum()
{
        return ::first + ::second;
}
```

Static Local Variables

- Static variable remembers and maintains its value even after its scope ends.
- Static Local variables are used when it is necessary for a function to remember a value between call to other functions.
  - For example, variable count used in a func



Static Local Variables

- **Scope**
  - Static variable is visible only inside the function in which it is defined.
  - It has the same scope as local variables.
- **Life time**
  - Life of a static variable is the same as that of a global variable, i.e. it remains in existence for the life of the program.

Example – Static Variables

```
void function(void);

int main()
{
        cout << "Count the number of times the function is
                  called : ";
        for (int i=0; i<5; i++)
              function();
}

void function( void )
{
    static int count = 1; // initialized first time only
    cout << "\nlocal static count is " << count << " on
                  entering the function" << endl;
    count++;
    cout << "local static count is " << count << " on
                  exiting useStaticLocal" << endl;
}
```

2 , 3 , 4 , 5 , 6

Variables and Storage Classes

Variable Type	Scope	Lifetime
Local or Auto	Within braces{}	Within braces{}
Global or External	Entire program	Entire program
Static	Within braces {}	Entire program

Important Points about Functions

- It should be noticed that the prototype is written before the main () function. This cause the prototype to be visible to all the functions in a file.
- The presence of ; should be noted carefully .
- The arguments and return type must match to the corresponding variable types of the prototype and definition.
- Note that you can only return **one** value from a function, however you can pass several values to a function.
- If we want to return more than one value from a function then we have to use pointers.
- We can pass constants as well as variable to the functions.

Important Points about Functions

- We can have same as well as different names for the variable used in calling and called programs.
- Any function can call any other function.
- All functions are visible to all other functions.

Important Points about Functions

- The variables which are declared inside a function are only visible to that function and are called local variables.
- If variable is declared outside the main, it will be visible to all functions.
- Global variable, use memory less efficiently than local variables. The rule is that variable, should be local unless there is a very good reason to make it global.
- A Static variable, declared within the function is visible to that function only. However it keeps its value b/w call to the function.