

Function

Lecture 7

Outline

- Functions
- Pre-defined and User-defined Functions
- Types of Functions
- Advantages of Writing Functions
- Code Reuse

Functions

- A function is a subprogram that performs a specific task.
- Functions you know:
 `cout << "Hi";`
 `cin >> number;`

Pre-defined and User-defined Functions

Pre-defined Function

- Is a function that is already defined in one of the many C libraries.
- It is included in the program through the preprocessor directive statement **#include**< > and used in the program to perform a specific task.
- Ex: #include<iostream> → Defines printf & scanf functions

User-defined Function

- Is a function that is created by the user.
- It is defined before the main program through a **function prototype** and called upon in the main program to perform a specific task.

Pre-defined Functions

Arithmetic Functions

- Are functions that perform arithmetic operations.
- They are usually located in the **cmath** libraries.
(see page 109).
- Examples:
 - **abs(x)** is a function that returns the absolute value of its integer argument.
 - **sqrt(x)** is a function that returns the square root of x.

Pre-defined Functions

Function call in an Assignment Statement

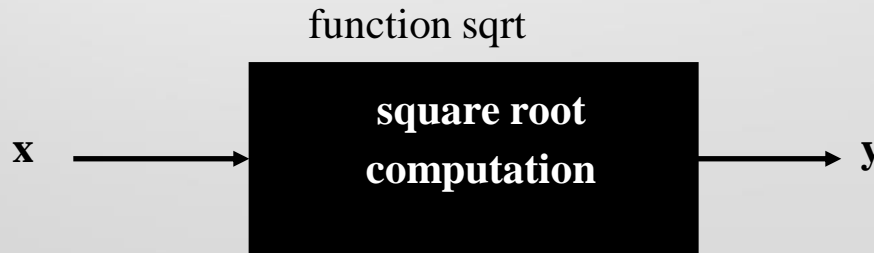
Syntax

$Y = \text{function name}(\text{argument});$

Example

$Y = \text{sqrt}(x);$

function call

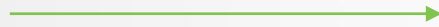


Pre-defined Functions

Examples

$x = -5$

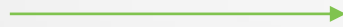
$y = \text{abs}(x)$



$y = 5$

$x = 90$

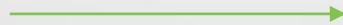
$y = \text{abs}(x) + 2$



$y = 92$

$x = 10$

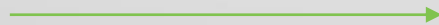
$y = \text{sqrt}(x + 6)$



$y = 4$

$x = 2.25$

$y = \text{sqrt}(x)$



$y = 1.25$

Types of Functions

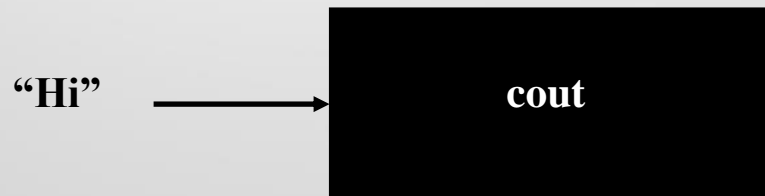
1. Program **sends** data to the function and **receives** data from the function.

Ex. `y=sqrt(x);`



2. Program **sends** data to the function and **doesn't receive** data from the function.

Ex. `cout << "Hi";`



Types of Functions (Continue)

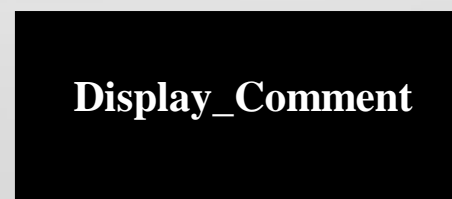
3. Program **doesn't sends** data to the function and **receives** data from the function.

Ex. `Number = Get_Number();`



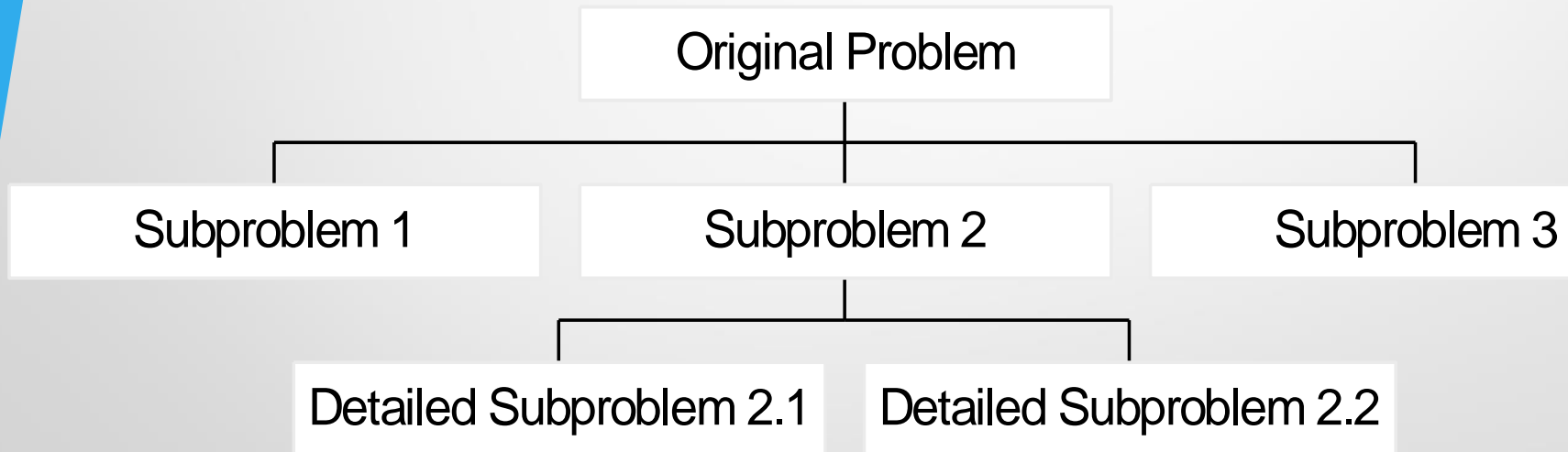
4. Program **doesn't sends** data to the function and **doesn't receive** data from the function.

Ex. `Display_Comment();`



Top-Down Design (Continue)

Structured Chart



Advantages of Writing Functions

1. **Decreases** the main program's length and reduces the chance to make **errors**.
2. Makes it **easier** to define programming tasks between programmers. Each programmer can be responsible for a particular set of functions.
3. Allows **procedural abstraction**; a programming technique in which a main function consists of a sequence of function calls and each function is implemented separately. This helps to focus on one function at a time instead of writing the complete program all at once.
4. Allows **code reuse** of function programs.

Code Reuse

- Reusing program fragments that have already been written and tested whenever possible.
- It is a way to help in writing error free code.

User-defined Functions

Function Prototype

- A way to declare a function.
- This tells the compiler:
 - The name of the function
 - The data type of received data(if there is).
 - The type & name of sent data (if there is). Also called the parameters of the function.


Syntax

ftype fname ();

Example

void drawcircle();

defines a function called drawcircle with not sent nor received data (when the program doesn't send to the function and doesn't receive from the function, the function is called a void function)



User-defined Functions

Function Call

- A function call transfers control from the main program to the function or subprogram.

Syntax

fname ();

Example

Drawcircle();

calls function drawcircle and causes it to begin execution



User-defined Functions

Function Definition

- Defines the function operation by writing the code for the function in a similar way to writing the code for the main program.

Syntax

```
Ftype fname(void)
{ local declaration
  executable statements }
```

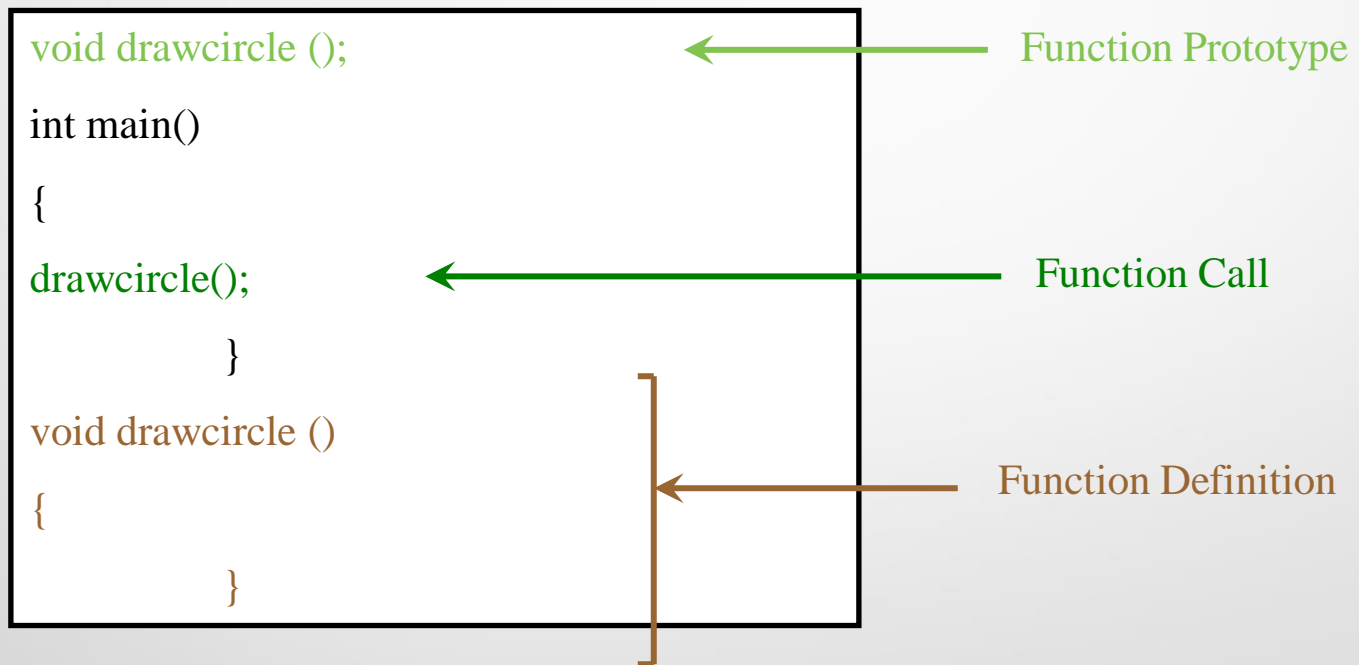
Example

```
void drawcircle(void)
{ cout << " *** " << endl;
  cout << "*"      "*" << endl;
  cout << " *** " << endl; }
```

does not contain the return statement because this function will not send any data to the program “main function” (i.e. a function should contain the return statement only if it is going to send data to the program)

User-defined Functions

Main Program



User-defined Functions

Main Program

```
void drawcircle ();
```

Function Prototype

```
void drawcircle ()
```

```
{
```

```
}
```

Function Definition

```
int main()
```

```
{
```

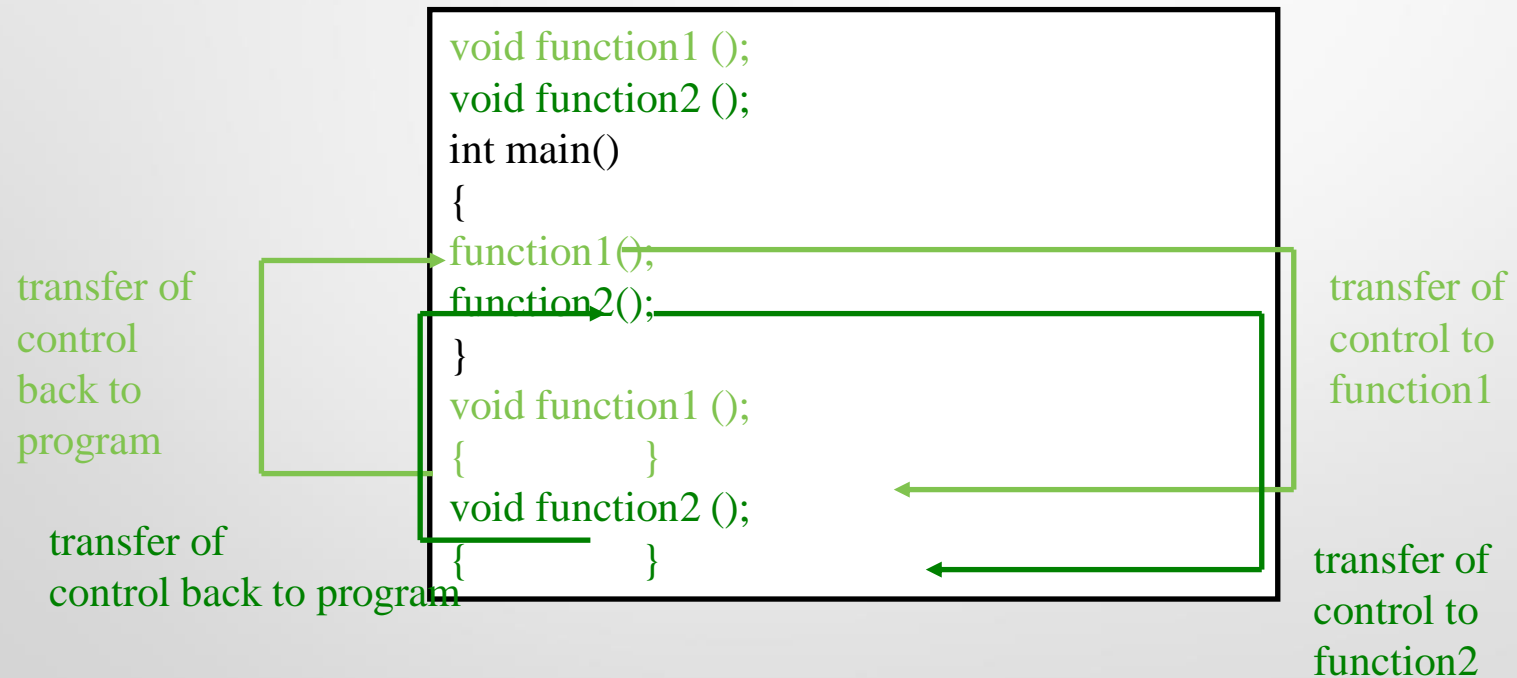
```
drawcircle();
```

Function Call

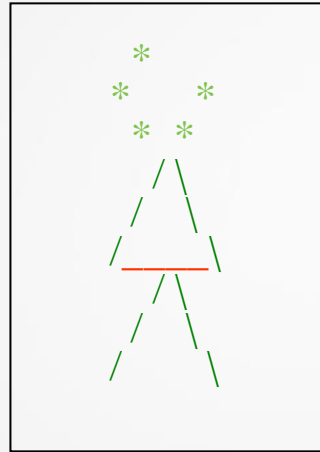
```
}
```

User-defined Functions

Order of Execution



Example - Drawing a Stick Figure



Draw a Figure

Draw a Circle

Draw a Triangle

Draw Intersecting Lines

Draw Intersecting Lines

Draw a Base

Example - Drawing a Stick Figure

```
/* draws a stick figure */
```

```
#include <iostream>
```

```
using namespace std;
```

```
void draw_circle();
```

```
void draw_intersect();
```

```
void draw_base();
```

```
void draw_triangle();
```

```
/* Function prototype */
```

```
/* Function prototype */
```

```
/* Function prototype */
```

```
/* Function prototype */
```

```
int
```

```
main(void)
```

```
{ draw_circle();
```

```
  draw_triangle();
```

```
  draw_intersect();
```

```
  return(0); }
```

```
/* function call */
```

```
/* function call */
```

```
/* function call */
```

```
void draw_circle()
```

```
{ cout << " * " << endl;
```

```
  cout << "* * " << endl;
```

```
  cout << " * * " << endl; }
```

```
/* Function definition */
```

(continued)

Example - Drawing a Stick Figure - Continue

draws a stick figure - continue*/

```
void draw_intersect()    /* Function definition */
{
    cout << "  /\\" << endl;
    cout << " /  \\" << endl;;
    cout << "/   \\" << endl;
}

void draw_base()         /* Function definition */
{
    cout << "-----" << endl;
}

void draw_triangle()     /* Function definition */
{
    draw_intersect();
    draw_base();
}
```

Using Functions

1. Declare a function before the main program.
2. Call the function inside the main program.
3. Define a function after the main program.

Review of Syntax

Function Prototype & definition

`f`type `f`name (data type name, ...);

Formal Parameter List

an identifier that represents a corresponding actual argument in a function definition.

Ex. `int square(int x)`

Function Call

`f`name (data type name, ...);

Actual Argument List

An expression used inside the parentheses of a function call

Ex. `square(x)`

Rules for Argument List

1. The **number** of actual arguments used in a function call must be the same as the number of the formal parameters listed in the function prototype and definition.
2. The **Order** of the arguments in the lists determines correspondence and must be the same in arguments and parameters.
The first actual argument corresponds to the first formal parameter, the second actual argument corresponds to the second formal parameter and so on.
3. The **data types** of the actual argument and the formal parameter must match.

Example

Prototype& definition:

double squared (int x, double y)



What is returned from
the function (i.e. result)



What is sent to the function

Call: __

y = squared (x,y)



What is returned from
the function (i.e. result)



What is sent to the function

Types of Functions

1. Program **sends** data to the function and **receives** data from the function.

Prototype & definition: **return_type** **function_name** (parameter list)

Call: **variable_name** = **function_name** (argument list)

2. Program **sends** data to the function and **doesn't receive** data from the function.

Prototype & definition: **void** **function_name** (parameter list)

Call: **function_name** (argument list)

Types of Functions (Continue)

3. Program **doesn't sends** data to the function and **receives** data from the function.

Prototype & definition: **return_type** **function_name** (**void**)

Call: **variable_name** = **function_name** ()

4. Program **doesn't sends** data to the function and **doesn't receive** data from the function.

Prototype & definition: **void** **function_name** (**void**)

Call: **function_name** ()

Examples of Function Prototype & definition

float squared (int number)



Returns back a float type
to the main program



An integer type called number is sent
to the function squared

Void print_report (int report_number)



Returns nothing
to the main program



An integer type called report_number
is sent to the function print_report

Examples of Function Prototype & definition

`char get_menu_choice ()`



Returns back a character type to the main program



Nothing is sent to the function `get_menu_choice`

`Void print_comment ()`



Returns nothing to the main program



Nothing is sent to the function `print_comment`

Program Example 1

.....
float half_of(float k); /* function prototype */

int
main()
{

.....
y=half_of(x); /* calling function */

.....
}

float half_of(float k) /* function definition */
{
 k = k / 2;
 return(k);
}



Program Example 2

```
..... computes the cube of num */
.....
int cubed(int x);          /* function prototype */

int
main()
{
    .....
    y=cubed(num);          /* calling function */
    .....
}

int cubed(int x)           /* function definition */
{
    return(x*x*x);
}
```

The diagram illustrates the relationship between a function call and its definition. A green arrow originates from the `y=cubed(num);` line in the `main()` function and points to the `int cubed(int x)` line in the function definition block. Another green arrow originates from the closing brace of the `main()` function and points to the same `int cubed(int x)` line, indicating the scope of the function call.

Program Example 3

```
.....  
int larger(int x,int y); /* function prototype */  
  
int  
main()  
{  
    .....  
    y=larger(num1,num2); /* calling function */  
    .....  
}  
int larger(int x,int y) ←  
{  
    .....  
    return(larger_num);  
}
```

The diagram illustrates the relationship between a function call and its definition. A green arrow originates from the function call `y=larger(num1,num2);` within the `main()` function and points to the function definition `int larger(int x,int y)`. This visualizes the compiler's process of resolving the function call by finding its definition.

Program Example 4

calculates the area and circumference of a circle */

```
.....  
float find_area(float r);    /* function prototype */  
float find_circum(float r);  /* function prototype */
```

```
int  
main()  
{
```

```
.....  
    area = find_area(r);    /* calling function */  
    circum = find_circum(r); /* calling function */
```

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

```
float find_area(float r) /* function definition */  
{ return(PI*POW(r,2)); }
```

```
float find_circum(float r)  
{ return(2.0*PI*r) }
```

Example 1

```
#include<iostream>
#include<cmath.h>
using namespace std;
double scale(double x, int n);

int main()
{
    double num1;
    int num2;
    cout << "Enter a real number> ";
    cin >> num1;
    cout << "Enter an integer> ";
    cin >> num2;
    cout << "Result of call to function scale is \n", scale(num1,num2));
    return 0;
}

double scale(double x, int n) /* This function multiplies its first */
{
    double scale_factor;      /* argument by the power of 10 */
    scale_factor = pow(10,n); /* specified by its second argumnet */
    return(x * scale_factor);
}
```

Example 2

```
#include<iostream>
# include<cmath>
using namespace std;
void print_comment();
double scale(double x, int n);

int main()
{   double num1,
    int  num2;
    print_comment();
    cout << "Enter a real number> ";
    cin >> num1;
    cout << "Enter an integer> ";
    cin >> num2;
    cout << "Result of call to function scale is \n",scale(num1,num2));
    return 0;
}

void print_comment(void)
{   cout << "The program will ..... \n";}

double scale(double x, int n) {   ..... }
```

Example 3

```
include<iostream>
# include<cmath>
using namespace std;
double scale(double x, int n);
void display(double result);

int main()
{ double num1, result;
  int num2;
  cout << "Enter a real number> ";
  cin >> num1;
  cout << "Enter an integer> ";
  cin >> num2;
  result = scale(num1,num2);
  display(result);
  return 0;}

double scale(double x, int n)
{ ..... }

void display(double result)
{ printf("Result of call to function scale is %.2f\n", result);}
```

Example 4

```
#include<iostream>
# include<cmath>
using namespace std;
double get_num1();
int get_num2();
double scale(double x, int n);

int main(void)
{ double num1,
  int num2;
  num1 = get_num1();
  num2 = get_num2();
  cout << "Result of call to function scale is\n",scale(num1,num2));
  return 0;}

double get_num1(void)
{ double number1;
  cout << "Enter a real number> ";
  cin >> number1;
  return(number1);}

int get_num2(void)
{ .....
  return(number2); }

double scale(double x, int n) { ..... }
```

Summary

- Functions are small programs that divide the program in manageable tasks.
- Predefined or ready to use functions are provided in the C++ library.
- User defined functions may or may not return a value.
- Variables passed to a function in the heading are called parameters.
- Actual values replace the parameters when a function is called.
- Value returned from a function can be used anywhere in the main or another function.



Thank You