Functions in C++

Dr. Anwar Shah, PhD, MBA(HR)

Assistant Professor in CS

FAST National University of Computer and Emerging Sciences CFD

Functions

- Function
  - definition
  - prototype
  - Parameters and pass-by-value
  - `return` statement
  - default parameter values
  - overloading
  - passing arrays to function
  - pass-by-reference
  - `inline` functions
  - `auto` return type
  - recursive functions

What is a Function?

What is a function?

- C++ programs
  - C++ Standard Libraries (functions and classes)
  - Third-party libraries (functions and classes)
  - Our own functions and classes

- Functions allow the modularization of a program
  - Separate code into logical self-contained units
  - These units can be reused

What is a function?

```
int main() {

 // read input
   statement1;
   statement2;
   statement3;
   statement4;

 // process input
   statement5;
   statement6;
   statement7;

 // provide output
   statement8;
   statement9;
   statement10;

   return 0;
}
```

Modularized Code

```
int main() {

 // read input
   read_input();

 // process input
   process_input();

 // provide output
   provide_output();

   return 0;
}
```

What is a function?

```
int main() {

   read_input();

   process_input();

   provide_output();

   return 0;
}
```

```
read_input() {
   statement1;
   statement2;
   statement3;
   statement4;
}

process_input() {
   statement5;
   statement6;
   statement7;
}

provide_output() {
   statement8;
   statement9;
   statement10;
}
```

What is a function?

Boss/Worker analogy

- Write your code to the function specification
- Understand what the function does
- Understand what information the function needs
- Understand what the function returns
- Understand any errors the function may produce
- Understand any performance constraints

- Don't worry about HOW the function works internally
  - Unless you are the one writing the function!

What is a function?

Example `<cmath>`

- Common mathematical calculations
- Global functions called as:

```
function_name(argument);
function_name(argument1, argument2, …);
```

```
cout << sqrt(400.0) << endl;  // 20.0
double result;
result = pow(2.0, 3.0);       // 2.0^3.0
```

What is a function?

User-defined functions

- We can define our own functions
- Here is a preview

```
/* This is a function that expects two integers a and b
   It calculates the sum of a and b and returns it to the caller
   Note that we specify that the function returns an int
*/

int add_numbers(int a, int b)
{
    return a + b;
}

// I can call the function and use the value that is returns

cout << add_numbers(20, 40);
```

What is a function?

User-defined functions

- Return zero if any of the arguments are negative

```
/* This is a function that expects two integers a and b
   It calculates the sum of a and b and returns it to the caller
   Only if a or b are non-negative. Otherwise, it returns 0
   Note that we specify that the function returns an int
*/

int add_numbers(int a, int b)
{
    if (a < 0 || b < 0)
        return 0;
    else
        return a + b;
}
```

Function Definition

Defining Functions

- name
  - the name of the function
  - same rules as for variables
  - should be meaningful
  - usually a verb or verb phrase
- parameter list
  - the variables passed into the function
  - their types must be specified
- return type
  - the type of the data that is returned from the function
- body
  - the statements that are executed when the function is called
  - in curly braces {}

Defining Functions

Example with no parameters

```
int function_name ( )
{
    statements(s);
    return 0;
}
```

1. Name
2. Parameters
3. Return type
4. Body

Defining Functions

Example with 1 parameter

```
int function_name(int a)
{
    statements(s);
    return 0;
}
```

1. Name
2. Parameters
3. Return type
4. Body

Defining Functions

Example with no return type (void)

```
void function_name ( )
{
    statements(s);
    return; // optional
}
```

1. Name
2. Parameters
3. No return type
4. Body

Defining Functions

Example with multiple parameters

```
void function_name(int a, std::string b)
{
    statements(s);
    return; // optional
}
```

Defining Functions

A function with no return type and no parameters

```
void say_hello () {
    cout << "Hello" << endl;
}
```

Calling a function

```
void say_hello () {
    cout << "Hello" << endl;
}

int main() {
    say_hello();
    return 0;
}
```

Calling a function

```
void say_hello () {
   cout << "Hello" << endl;
}

int main() {
   for (int i{1} i<=10; ++i)
      say_hello();
   return 0;
}
```

Calling a function

```
void say_world () {
   cout << " World" << endl;
}

void say_hello () {
   cout << "Hello" << endl;
   say_world();
}

int main() {
   say_hello();
   return 0;
}
```

Calling a function

```
void say_world () {
   cout << " World" << endl;
   cout << " Bye from say_world" << endl;
}

void say_hello () {
   cout << "Hello" << endl;
   say_world();
   cout << " Bye from say_hello" << endl;
}

int main() {
   say_hello();
   cout << " Bye from main" << endl;
   return 0;
}
```

```
Hello
World
Bye from say_world
Bye from say_hello
Bye from main
```

Calling functions

- Functions can call other functions
- Compiler must know the function details **BEFORE** it is called!

```
int main() {
    say_hello();  // called BEFORE it is defined ERROR
    return 0;
}

void say_hello ()
{
    cout << "Hello" << endl;
}
```

Function Prototype

Function Prototypes

- **The compiler must ‘know’ about a function before it is used**

  - Define functions before calling them
    - OK for small programs
    - Not a practical solution for larger programs

  - Use function prototypes
    - Tells the compiler what it needs to know without a full function definition
    - Also called forward declarations
    - Placed at the beginning of the program
    - Also used in our own header files (.h) – more about this later

Example

```
int function_name();  // prototype


int function_name()
{
    statements(s);
    return 0;
}
```

Example

```
int function_name(int);    // prototype
                         // or
int function_name(int a); // prototype


int function_name(int a) {
   statements(s);
   return 0;
}
```

Example

```
void function_name(); // prototype


void function_name()
{
    statements(s);
    return; // optional
}
```

Example

```
void function_name(int a, std::string b);
// or
void function_name(int, std::string);


void function_name(int a, std::string b)
{
   statements(s);
   return; // optional
}
```

A function with no return type and no parameters

```
void say_hello();

void say_hello() {
    cout << "Hello" << endl;
}
```

Calling a function

```
void say_hello();

int main() {
    say_hello();          // OK
    say_hello(100);          // Error
    cout << say_hello(); // Error
                       // No return value
    return 0;
}
```

Example

```
void say_hello(); // prototype
void say_world(); // prototype

int main() {
   say_hello();
   cout << " Bye from main" << endl;
   return 0;
}
```

```
void say_world () {
   cout << " World" << endl;
   cout << " Bye from say_world" << endl;
}

void say_hello () {
   cout << "Hello" << endl;
   say_world();
   cout << " Bye from say_hello" << endl;
}
```

Parameters and Pass by Values

Function Parameters

- When we call a function we can pass in data to that function
- In the function call they are called arguments
- In the function definition they are called parameters
- They must match in number, order, and in type

Example

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

int main() {
    int result {0};
    result = add_numbers(100,200);  // call
    return 0;
}

int add_numbers(int a, int b) { // definition
    return a + b;
}
```

Example

```
void say_hello(std::string name) {
   cout << "Hello " << name << endl;
}

say_hello("Frank");

std::string my_dog {"Buster"};
say_hello(my_dog);
```

Pass-by-value

- When you pass data into a function it is passed-by-value
- A copy of the data is passed to the function
- Whatever changes you make to the parameter in the function does NOT affect the argument that was passed in.

- Formal vs. Actual parameters
  - Formal parameters – the parameters defined in the function header
  - Actual parameters – the parameter used in the function call, the arguments

Example

```
void param_test(int formal) {  // formal is a copy of actual
   cout << formal << endl;    / 50
   formal = 100;             // only changes the local copy
   cout << formal << endl;   // 100
}

int main() {
   int actual {50};
   cout << actual << endl;   // 50
   param_test(actual);        // pass in 50 to param_test
   cout << actual << endl;   // 50   - did not change
   return 0
}
```

Return Statement

Function Return Statement

- If a function returns a value then it must use a `return` statement that returns a value
- If a function does not return a value (`void`) then the `return` statement is optional
- `return` statement can occur anywhere in the body of the function
- `return` statement immediately exits the function
- We can have multiple `return` statements in a function
    - Avoid many return statements in a function
- The return value is the result of the function call

Default Parameter Values

Default Argument Values

- When a function is called, all arguments must be supplied

- Sometimes some of the arguments have the same values most of the time

- We can tell the compiler to use default values if the arguments are not supplied

- Default values can be in the prototype or definition, not both
  - best practice – in the prototype
  - must appear at the tail end of the parameter list

- Can have multiple default values
  - must appear consecutively at the tail end of the parameter list

Default Argument Values

Example – no default arguments

```
double calc_cost(double base_cost, double tax_rate);

double calc_cost(double base_cost, double tax_rate) {
    return base_cost += (base_cost * tax_rate);
}

int main() {
    double cost {0};
    cost = calc_cost(100.0, 0.06);
    return 0;
}
```

Default Argument Values

Example – single default argument

```
double calc_cost(double base_cost, double tax_rate = 0.06);

double calc_cost(double base_cost, double tax_rate) {
   return base_cost += (base_cost * tax_rate);
}

int main() {
   double cost {0};
   cost = calc_cost(200.0);   // will use the default tax
   cost = calc_cost (100.0, 0.08);      // will use 0.08 not the defauly
   return 0;
}
```

Default Argument Values

Example – multiple default arguments

```
double calc_cost(double base_cost, double tax_rate = 0.06, double shipping = 3.50);

double calc_cost(double base_cost, double tax_rate, double shipping) {
   return base_cost += (base_cost * tax_rate) + shipping;
}

int main() {
   double cost {0};
   cost = calc_cost (100.0, 0.08, 4.25);  // will use no defaults
   cost = calc_cost(100.0, 0.08);    // will use default shipping
   cost = calc_cost(200.0);      // will use default tax and shipping
return 0;
}
```

Overloading

Overloading Functions

- We can have functions that have different parameter lists that have the same name

- Abstraction mechanism since we can just think ‘print’ for example

- A type of polymorphism
  - We can have the same name work with different data types to execute similar behavior

- The compiler must be able to tell the functions apart based on the parameter lists and argument supplied

Overloading Functions

Example

```
int add_numbers(int, int);     // add ints
double add_numbers(double, double);  // add doubles

int main() {
   cout << add_numbers(10,20) << endl;     // integer
   cout << add_numbers(10.0, 20.0) << endl; // double
 return 0;
}
```

Overloading Functions

Example

```
int add_numbers(int a, int b) {
   return a + b;
}

double add_numbers(double a, double b) {
   return a + b;
}
```

Overloading Functions

Example

```
void display(int n);
void display(double d);
void display(std::string s);
void display(std::string s, std::string t);
void display(std::vector<int> v);
void display(std::vector<std::string> v);
```

Overloading Functions

Return type is not considered

```
int    get_value();
double get_value();

// Error

cout << get_value() << endl; // which one?
```

Passing Array to Function

Passing Arrays To Functions

- We can pass an array to a function by providing square brackets in the formal parameter description

```
void print_array(int numbers []);
```

- The array elements are NOT copied
- Since the array name evaluates to the location of the array in memory – this address is what is copied
- So the function has no idea how many elements are in the array since all it knows is the location of the first element (the name of the array)

Example

```
void print_array(int numbers []);

int main() {
   int my_numbers[] {1,2,3,4,5};
   print_array(my_numbers);
   return 0;
}

void print_array(int numbers []) {
   // Doesn't know how many elements are in the array???
   // we need to pass in the size!!
}
```

Example

```
void print_array(int numbers [], size_t size);

int main() {
   int my_numbers[] {1,2,3,4,5};
   print_array(my_numbers, 5);    / 1 2 3 4 5
   return 0;
}

void print_array(int numbers [], size_t size) {
   for (size_t i{0}; i < size; ++i )
      cout << numbers[i] << endl;
}
```

Example

- Since we are passing the location of the array
    - The function can modify the actual array!

```
void zero_array(int numbers [], size_t size) {
    for (size_t i{0}; i < size; ++i )
        numbers[i] = 0;                      // zero out array element
}
int main() {
    int my_numbers[] {1,2,3,4,5};
    zero_array(my_numbers, 5);          // my_numbers is now zeroes!
    print_array(my_numbers, 5);         // 0 0 0 0 0
    return 0;
}
```

const parameters

- We can tell the compiler that function parameters are const (read-only)
- This could be useful in the print_array function since it should NOT modify the array

```
void print_array(const int numbers [], size_t size) {
    for (size_t i{0}; i < size; ++i )
        cout << numbers[i] << endl;
    numbers[i] = 0;    // any attempt to modify the array
                       // will result in a compiler error
}
```

Pass by Reference

Pass by Reference

- Sometimes we want to be able to change the actual parameter from within the function body
- In order to achieve this we need the location or address of the actual parameter
- We saw how this is the effect with array, but what about other variable types?
- We can use reference parameters to tell the compiler to pass in a reference to the actual parameter.
- The formal parameter will now be an alias for the actual parameter

Example

```
void scale_number(int &num);        // prototype

int main() {
   int number {1000};
   scale_number(number);          // call
   cout << number << endl;        // 100
   return 0;
}

void scale_number(int &num) {   // definition
   if (num > 100)
      num = 100;
}
```

Example

```
void swap(int &a, int &b);

int main() {
    int x{10}, y{20};
    cout << x << " " << y << endl;   // 10 20
    swap(x, y);
    cout << x << " " << y << endl;   // 20 10
    return 0;
}

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

vector example – pass by value

```
void print(std::vector<int> v);

int main() {
   std::vector<int> data {1,2,3,4,5};
   print(data);                        // 1 2 3 4 5
   return 0;
}

void print(std::vector<int> v) {
   for (auto num: v)
      cout << num << endl;
}
```

vector example – pass by reference

```
void print(std::vector<int> &v);

int main() {
   std::vector<int> data {1,2,3,4,5};
   print(data);                       // 1 2 3 4 5
   return 0;
}

void print(std::vector<int> &v) {
   for (auto num: v)
      cout << num << endl;
}
```

vector example – pass by const reference

```
void print(const std::vector<int> &v);

int main() {
   std::vector<int> data {1,2,3,4,5};
   print(data);                         // 1 2 3 4 5
   return 0;
}

void print(const std::vector<int> &v) {
   v.at(0) = 200;                               // ERROR
   for (auto num: v)
      cout << num << endl;
}
```

Scope Rules

- C++ uses scope rules to determine where an identifier can be used
- C++ uses static or lexical scoping
- Local or Block scope
- Global scope

Local or Block scope

- Identifiers declared in a block { }
- Function parameters have block scope
- Only visible within the block { } where declared
- Function local variables are only active while the function is executing
- Local variables are NOT preserved between function calls
- With nested blocks inner blocks can ‘see’ but outer blocks cannot ‘see’ in

Static local variables

- Declared with static qualifier

```
static int value {10};
```

- Value IS preserved between function calls
- Only initialized the first time the function is called

Global scope

- Identifier declared outside any function or class
- Visible to all parts of the program after the global identifier has been declared
- Global constants are OK
- Best practice – don’t use global variables

How do Function Calls Work?

- **Functions use the 'function call stack'**
  - Analogous to a stack of books
  - LIFO – Last In First Out
  - push and pop

- **Stack Frame or Activation Record**
  - Functions must return control to function that called it
  - Each time a function is called we create an new activation record and push it on stack
  - When a function terminates we pop the activation record and return
  - Local variables and function parameters are allocated on the stack

- **Stack size is finite – Stack Overflow**

Inline Functions

Inline Functions

- Function calls have a certain amount of overhead
- You saw what happens on the call stack
- Sometimes we have simple functions
- We can **suggest** to the compiler to compile them 'inline'
  - avoid function call overhead
  - generate inline assembly code
  - faster
  - could cause code bloat
- Compilers optimizations are very sophisticated
  - will likely inline even without your suggestion

Example

```
inline int add_numbers(int a, int b) {// definition
    return a + b;
}

int main() {
    int result {0};
    result = add_numbers(100,200);// call
    return 0;
}
```

Recursive Functions

Recursive Functions

- A recursive function is a function that calls itself
  - Either directly or indirectly through another function
- Recursive problem solving
  - Base case
  - Divide the rest of problem into subproblem and do recursive call
- There are many problems that lend themselves to recursive solutions
- Mathematic – factorial, Fibonacci, fractals,...
- Searching and sorting – binary search, search trees, ...

Example - Factorial

```
0! = 1
n! = n * (n-1)!
```

- **Base case:**
  - factorial(0) = 1

- **Recursive case:**
  - factorial(n) = n * factorial(n-1)

Example - Factorial

```
unsigned long long factorial(unsigned long long n) {
   if (n == 0)
      return 1;                            // base case
   return n * factorial(n-1);       // recursive case
}

int main() {
   cout << factorial(8) << endl; // 40320
   return 0;
}
```

Example - Fibonacci

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2)
```

- **Base case:**
  - Fib(0) = 0
  - Fib(1) = 1

- **Recursive case:**
  - Fib(n) = Fib(n-1) + Fib(n-2)

Example - Factorial

```
unsigned long long fibonacci(unsigned long long n) {
   if (n <= 1)
      return n;                    // base cases
   return fibonacci(n-1) + fibonacci(n-2); // recursion
}

int main() {
   cout << fibonacci(30) << endl; // 832040
   return 0;
}
```

Important notes

- If recursion doesn't eventually stop you will have infinite recursion
- Recursion can be resource intensive
- Remember the base case(s)
    - It terminates the recursion
- Only use recursive solutions when it makes sense
- Anything that can be done recursively can be done iteratively
    - Stack overflow error

Thank You