## 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

- •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

```
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;
}
```

```
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;
```

#### 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!

#### Example < cmath>

- Common mathematical calculations
- •Global functions called as:

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

cout << sqrt(400.0) << end1; // 20.0
  double result;
  result = pow(2.0, 3.0); // 2.0^3.0</pre>
```

#### 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);</pre>
```

#### User-defined functions

•Return zero if any of the arguments are negative

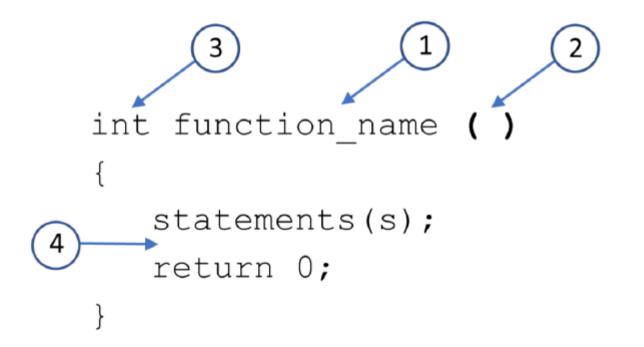
```
/* This is a function that expects two integers a and b
   It calculates the sum of a and b and returns it to the caller
   Only if a or b are non-negative. Otherwise, it returns 0
   Note that we specify that the function returns an int
*/

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

# Function Definition

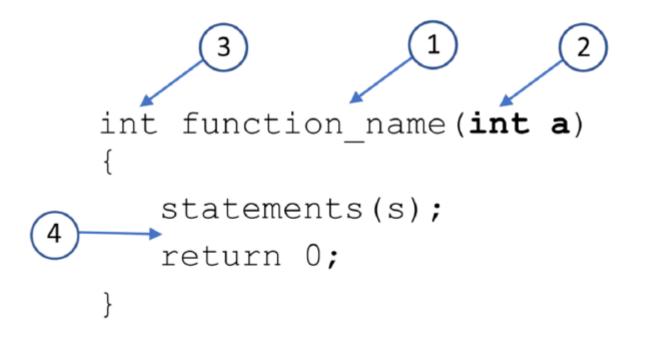
- name
  - •the name of the function
  - same rules as for variables
  - should be meaningful
  - usually a verb or verb phrase
- parameter list
  - the variables passed into the function
  - •their types must be specified
- return type
  - •the type of the data that is returned from the function
- body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

Example with no parameters



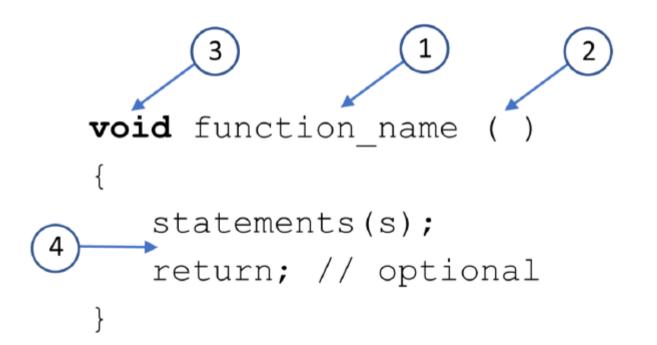
- 1. Name
- 2. Parameters
- 3. Return type
- 4. Body

Example with 1 parameter



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

Example with no return type (void)



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

Example with multiple parameters

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

A function with no return type and no parameters

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

```
void say_hello () {
   cout << "Hello" << endl;
}
int main() {
   say_hello();
   return 0;
}</pre>
```

```
void say_hello () {
   cout << "Hello" << endl;</pre>
int main() {
   for (int i{1} i<=10; ++i)
      say_hello();
   return 0;
```

```
void say_world () {
   cout << " World" << endl;</pre>
void say_hello () {
   cout << "Hello" << endl;</pre>
   say_world();
int main() {
   say_hello();
   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;
int main() {
  say hello();
  cout << " Bye from main" << endl;
  return 0;
```

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

- 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;
}</pre>
```

# 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

```
int function_name(); // prototype

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

```
void function_name(); // prototype

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

```
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;
}</pre>
```

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

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

say_world();
cout << " Bye from main" << endl;
cout << "Hell say_world();
cout << " Bye from cout << " Bye
```

```
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;
}</pre>
```

# 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

```
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;
```

```
void say_hello(std::string name) {
   cout << "Hello " << name << endl;
}
say_hello("Frank");
std::string my_dog {"Buster"};
say_hello(my_dog);</pre>
```

#### 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

```
void param test(int formal) { // formal is a copy of actual
  cout << formal << endl; / 50
  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

- 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

#### 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;
```

#### Example – single default argument

#### 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

- •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

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

```
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);
```

Return type is not considered

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

// Error

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

# 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)

```
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!!
```

```
void print array(int numbers [], size_t size);
int main() {
  int my numbers[] \{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;</pre>
```

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

#### 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
}</pre>
```

# 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

```
int main() {
 int number {1000};
 cout << number << endl; // 100
 return 0;
void scale_number(int &num) {    // definition
 if (num > 100)
   num = 100;
```

```
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};
                                   // 1 2 3 4 5
   print(data);
   return 0;
void print(std::vector<int> v) {
   for (auto num: v)
      cout << num << endl;</pre>
```

## vector example – pass by reference

```
void print(std::vector<int> &v);
int main() {
   std::vector<int> data {1,2,3,4,5};
                                  // 1 2 3 4 5
   print(data);
   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

```
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

## 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

#### 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