Functions

C++ for Developers

Functions

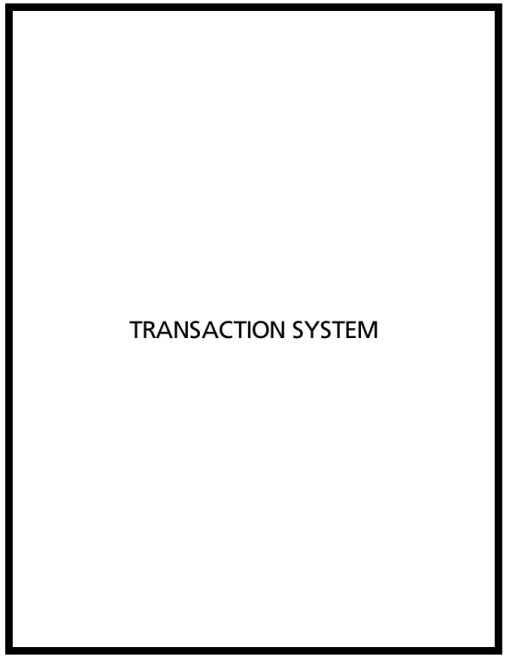
- Functions
- Syntax
- Copy by value
- Copy by reference
- Scope
- Overloading

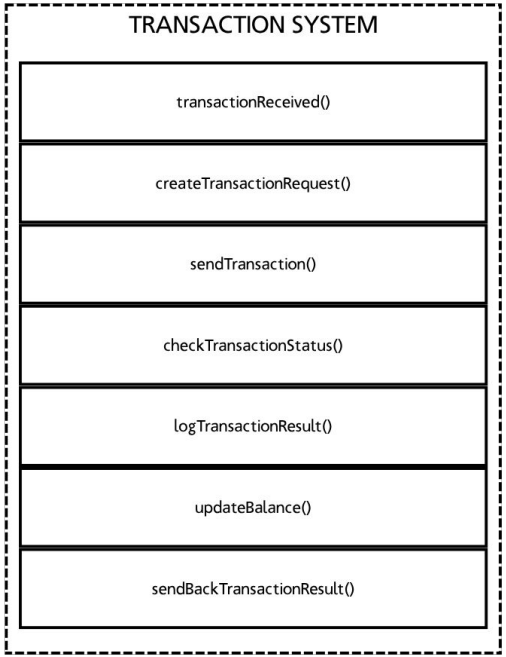
What is a function?

A set of instructions constrained to a module of code.

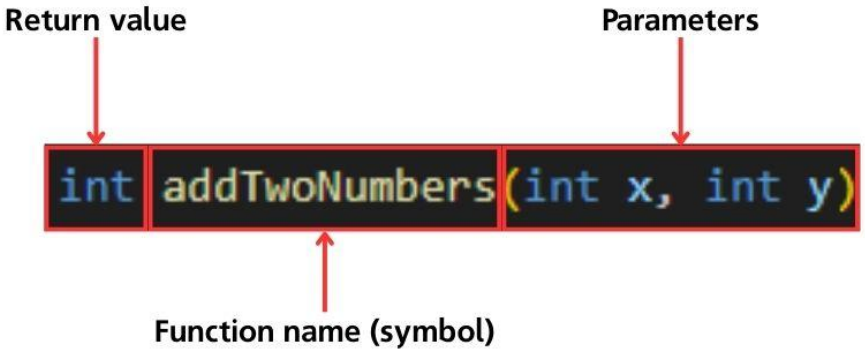
The defined function can be reused in your code.

Why do we use functions?





Syntax



Syntax

```cpp
// DECLARING A FUNCTION
int addTwoNumbers(int x, int y);

// DEFINING A FUNCTION
int addTwoNumbers(int x, int y) {
    return x + y;
}

int main(void) {
    int number1 = 3;
    int number2 = 22;

    std::cout << number1 << " + " << number2 << " = " << addTwoNumbers(number1, number2);

    return 0;
}
```

Scope

```
void staticAddTwoNumbers(int x, int y) {
    static bool calculated = false;
    static int lastCalculation;

    if (calculated) {
        std::cout << "Last calculation: " << lastCalculation << std::endl;
    }
    lastCalculation = x + y;
    std::cout << "New calculation: " << lastCalculation << std::endl;
    calculated = true;
}

int main(void) {

    staticAddTwoNumbers(20, 30);
    staticAddTwoNumbers(12, 32);

    return 0;
}
```

The variables inside the function only exists within the function. They are not reachable outside.

You can use the ‘static’ keyword to have a variable that lives on within the function. This can be a counter or last entered name etc…

lafftale for developers

Copy by value

```
void addTogether(int x, int y) {
    x += y;
}

int main(void) {
    int number1 = 3;
    int number2 = 22;

    addTogether(number1, number2);

    return 0;
}
```

What is the value of number1 after calling the function ‘addTogether()’?

lafftale for developers

Copy by reference

```cpp
// COPY BY REFERENCE
void addTogetherByReference(int& x, int &y) {
    x += y;
}

int main(void) {
    int number1 = 3;
    int number2 = 22;

    // COPY BY REFERENCE
    addTogetherByReference(number1, number2);
    std::cout << "Number 1: " << number1 << std::endl;
```

What is the value of number1 after calling the function ‘addTogether()’?

Reference

A reference is like a shortcut or alias to a variable. Instead of working with a copy, you’re working with the original, which means changes made through the reference will affect the actual variable. You create a reference by adding an ‘&’ after the data type.

We’ll explore the details later, but for now just think of a reference as a direct link to the real thing.

Overloading

```
int main(void) {
    int number1 = 3;
    int number2 = 22;
    double number3 = 3.2;
    double number4 = 2.5;

    // WRITING OUT THE RESULT OF ADDING TWO INTEGERS
    std::cout << number1 << " + " << number2 << " = " << addTwoNumbers(number1, number2) << std::endl;

    // WRITING OUT THE RESULT OF ADDING TWO DOUBLES
    std::cout << number3 << " + " << number4 << " = " << addTwoNumbers(number3, number4) << std::endl;
```

What will the result of adding number3 and number4 together with our function addTwoNumbers()?

Overloading

```
// DEFINING A FUNCTION
int addTwoNumbers(int x, int y) {
    return x + y;
}

// OVERLOADING A FUNCTION
double addTwoNumbers(double x, double y) {
    return x + y;
}
```

By creating another function with the same name, we can ‘overload’ it.

This practically mean that we create more use cases for the function.

Type of functions

C++ for Developers

Recursive Functions

A function directly or indirectly calls itself.

```
void countToNumber(int x) {
    static int goal = x;
    static int counter = 0;

    if (counter <= goal) {
        std::cout << counter << std::endl;
        counter++;
        countToNumber(goal);
    }
}
```

lafftale for developers

Function pointers

```
void doCalculation(int x, int y, void (*callback)(int)) {
    callback(addTwoNumbers(x,y));
}
```

```
void (*callback)(int)
```

Function pointer

Return type

Function name

Parameters

std::function is defined i
<functional>

```
void doDivision(double x, double y, std::function<void(double)> callback) {
    callback(x / y);
}
```

```
std::function<void(double)> callback
```

Return type

Function name

Parameters

lafftale for developers

Callback Functions

A function you pass as an argument to a function and is called from within the function when the time is right.

```
// FUNCTION THAT TAKES FUNCTION AS PARAMETER
void doCalculation(int x, int y, void (*callback)(int)) {
    callback(addTwoNumbers(x,y));
}

void writeResult(int value) {
    std::cout << "The result is: " << value << std::endl;
}

int main(void) {
    int number1 = 3;
    int number2 = 22;

    // USING A CALLBACK FUNCTION
    doCalculation(number1, number1, writeResult);

    return 0;
}
```

Lambda functions

A lambda is a tiny, nameless function you write inline.
It's defined right where you use it, and it can "capture" variables from the surrounding scope so it can use them.

```cpp
// USING A LAMBDA FUNCTION TO SORT A VECTOR
std::sort(v.begin(), v.end(), [](int a, int b) {
    return a < b;
});

for (auto number : v) {
    std::cout << number << std::endl;
}

// USING A LAMBDA FUNCTION TO ADD NUMBERS
int number5 = [=]() {
    return number1 + number2;
}();

// DECLARING A LAMBDA FUNCTION
auto localAdder = [](int x, int y) {
    return x + y;
};
int number6 = localAdder(number3, number4);
std::cout << "Lambda localAdder result: " << number6 << std::endl;

// PASSING LAMBDA FUNCTION AS A PARAMETER
doCalculation(number1, number2, [](int value) {
    std::cout << "Lambda callback function value : " << value << std::endl;
});
```

lafftale for developers

Lambda syntax

Within [ ] you define the scope you capture with the lambda function:

[ ] = capture nothing

[=] = any used variable from outside is captured by copy

[&] = any used variable from outside is captured by reference

[a, &b] = capture variable a by copy and b by reference

[a = aNew] = captures the variable by copy and gives it a new name. It works the same for reference.

Best practices

C++ for Developers

Functionality

Keep each function focused on a single purpose. This makes the function easier to reuse, test, and update without unintended side effects.

Naming conventions

Choose a descriptive name that clearly conveys the function’s purpose. If the name feels overly complex, it’s often a sign the function should be broken down into smaller, more focused functions.

Keeping track of results

Try to keep track what happens inside important functions by passing a reference and then use the return type as an indicator.

```
int addStringsTogether(std::string& firstWord, std::string secondWord, int maxLength) {
    if ((firstWord.length() + secondWord.length()) > maxLength) {
        return 1;
    } else {
        firstWord.append(secondWord);
        return 0;
    }
}
```

```
std::string firstName = "Kral";
std::string lastName = "Morg";

if (addStringsTogether(firstName, lastName, 8) == 0) {
    std::cout << "String concatenated properly" << std::endl;
    std::cout << "String: " << firstName << std::endl;
} else {
    std::cout << "String concatination failed" << std::endl;
    std::cout << "String: " << firstName << std::endl;
}
```

Don’t create rabbit holes

If your program ends up as a never-ending chain of functions calling functions calling functions... well, you’ve basically cooking spaghetti, and nobody likes debugging pasta.

Keep your call stacks shallow and easy to follow.

Let’s do some exercises

Functions

1. Hello World
2. The string joiner
3. Tax calculation


lafftale
for developers