Lecture 3: Functions

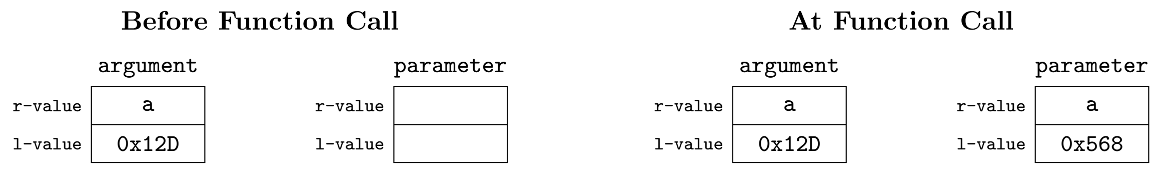
Introduction

A *function* modularizes a program (divides it into small parts), improving readability and versatility. Functions perform tasks that can be reused multiple times within a program and other files. They are structured similarly to mathematical functions in that they take inputs and produce an output; however, in C++, functions can also be defined without inputs or outputs.

Parameters

A function can take inputs called *parameters* and produce an output called a *return value*. A parameter is a local variable of a function, assigned an initial value when the function is invoked. Parameters accept arguments in several ways which are determined by the format of the parameter. The most common formats are *pass-by-value* and *pass-by-reference*.

A parameter pass-by-value receives a copy of its argument content (r-value) like a variable assignment. Thus, it can accept its argument as a literal, an expression, a variable, or a return-value function invocation. Additionally, modifications to the parameter do not reflect in the argument since they are unique entities (their l-values are different).

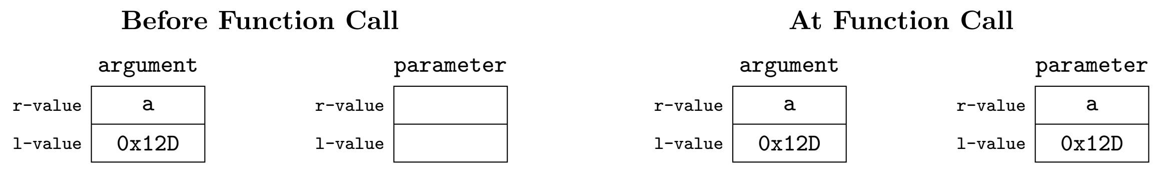


Its syntax is

$[\texttt{const}]^{?}$ *data-type identifier*

When it is made constant, it cannot be modified in the function as expected.

A parameter pass-by-reference, called a *reference parameter*, can only accept nonconstant variables as arguments if it is not constant. It shares the same address (l-value) as its argument, meaning they are the same entity with different names; hence, changes to the parameter are reflected in the argument. A reference parameter is useful for modifying multiple arguments and returning multiple values from a function. When it is strictly used as an output is called an *out parameter*.



Its syntax is

$[\texttt{const}]^{?}$ *data-type*`&` *identifier*

When made constant, along with prohibiting modifications to it in the function, it can accept all types of arguments.

Function Prototype

A *function prototype* is a function declaration. It tells the compiler that the function will be defined later and allows invocations of the function before its definition is written. Its syntax is

return-type identifier`(`$[$*parameter-list*$]^{?}$`);`

where *return-type* is data type or the keyword '`void`', which is the "no type" data type.

Ultimately, it is a *function header* followed by a semicolon such that parameter identifiers are optional. It can be written any scope (local and global); however, only functions with function prototypes in the same local scope can invoke each other. Furthermore, multiple prototypes can be declared in a list if they share the same return type.

Example:
The following code provides examples of function prototypes

```
void a(int); //prototype of a void function
int b(double& t), c(char,int); //a list of prototypes of int functions

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

Function Definitions

A function definition provides the function's behavior; that is, it is a collection of statements that perform a task. Its syntax is

return-type identifier([*parameter-list*]$^?$){*body*}

Unlike a function prototype, it must be written in the global scope and its parameters must have identifiers. If its return type is not void, it is called a *return-value function* and its body must contain at least one reachable *return statement* with the syntax

`return` *argument* `;`

where the type of the *argument* must match (or implicitly typecast to match) the return type.

The return statement produces the output of a function; hence, it terminates the function when it is executed regardless of where it is in the function's body. Therefore, although multiple return statements can be present in a function, only one will be executed.

Example:
In the following program only the first display, '`Line A is executed`', is execited even though there are additional displays because the first return statement executes before the other displays are reached.

```
int main()
{
  cout << "Line A is executed\n";
  return 0;
  cout << "Line B is executed\n";
  return 0;
  cout << "Line C is executed\n";
  return 0;
}
```

If the function's return type is void, it is called a *procedure*, which is usually used to display content or manipulate reference parameters. Like return-value functions, procedures can include return statements; however, the statement must omit the argument; otherwise, it produces an error.

`return;`

Example:
A few examples of procedures are illustrated below

```
void input(std::string msg,int& in)
{
  cout << msg << '\n';
  cin >> in;
}
```

```
void swapChar(char& a,char& b)
{
  char t = a;
  a = b;
  b = t;
}
```

```
void echo(const std::string& msg)
{
  cout << msg << "\n" << msg;
  return;
}
```

It is possible to define several functions with the same name as long as they have different parameter lists. This process is called *function overloading*. Parameter lists are considered distinct if the parameters in the same position have different incompatible data types or if the number of parameters in the lists differs. The return types of functions do not contribute to the overloading process; hence, they may match or differ.

Example:
An example of overloaded functions is illustrated below

```
double d(double x, double y)
{
  return x * x + y * y;
}

void d(double a)
{
  std::cout << 2 * a;
}

std::string d(std::string msg)
{
  return msg + msg + msg;
}
```

A slightly similar approach to function overloading available is using *default values*. A function can be defined so that some or all of its parameters have default values, allowing its invocations to accept fewer arguments. Default values must be specified from right to left, meaning once a parameter is given a default value, all subsequent parameters must also have default values. Furthermore, default values are assigned in the function declaration (function prototype) if both declaration and definition are provided; otherwise, they are assigned in the function definition. Its syntax is

parameter = *argument*

Example:
An example of functions with default values

```
std::string repeat(const std::string&,unsigned int = 3);

double sum(int x,int y = 1,int z = 1) {return (x+y+z);}

std::string repeat(const std::string& msg,unsigned int n)
{
  std::string r;
  for(int i = 0;i < n;i += 1) {r += msg + '\n';}
  return r;
}
```

Function Caller

A *function caller* (or *caller*) is a function evaluator. Its syntax is

function-name($[$*argument-list*$]^?$)

where *function-name* is the identifier of an existing function definition or prototype. Each argument corresponds to the parameter in the same position in their respective lists; hence, the lists must have the same length and arguments must have the appropriate data type and acceptable format of their respective parameter. Last, callers can only be written in local scopes and callers of return-value functions can be used as arguments.

Example:
An example of function callers of previously defined functions is illustrated below

```
int main()
{
  int x;
  char a = '8', b = 'V';

  input("Enter an integer:",x); //x will be assigned a value from the user
  swapChar(a,b); //after call a = 'V' and b = '8'
  echo(d("Hi")); //displays "HiHiHi\nHiHiHi"
  d(d(3.0,4.0)); //displays 50
  std::cout << sum(3,2,5) << '-'<< sum(4,6) << '-'<< sum(7); //displays 10-11-9
  echo(repeat(repeat("so",2)));
  //previous displays "so\nsoso\nsoso\nso\nso\nsoso\nsoso\nso"
}
```