Introduction to Programming

Lecture 2 — Reproducing tasks with functions

Nicolas Gry

Université de Saint-Étienne | Year 2026/2027

digicrea digital creativity

Université Jean Monnet Saint-Étienne

-1 Recap’ time

Recap: Variables

Recap: Variables

- Have a type
- Have a declaration and initialization
- Store a value
- Have a scope
- Can be constant with `const`

Recap: Booleans

Recap: Booleans

- Have type `bool`
- Can only have one two values: `true` or `false`
- Can be combined with operations like `&&` (logical and) and `||` (logical or)
- The result of comparison expressions is a boolean

Recap: Conditional statements

Recap: Conditional statements

- Using `if`, `else if`, `else` or `switch-case/default`
- Allow branching according to the value of a boolean expression or variable

Recap: Loops

Recap: Loops

- Conditional loops with `while`
- Finite loops with `for`
- Allow repeating actions

00 Assignment review

Solution: Exercise 1

Let's look at the solution: `https://godbolt.org/z/cEeT3x9Kr`

Solution: Exercise 2

Let's look at the solution: `https://godbolt.org/z/T9rq1PvYM`

01 Recycle pieces of code with Functions

Fresh example

A brand new example

About functions

Functions allow reusing portions of code. They have:

About functions

```
void print();
```

Functions allow reusing portions of code. They have:

- A declaration

About functions

Functions allow reusing portions of code. They have:

- A declaration
- A definition

```
void print();
```

```
void print() {
    std::cout "Hello, World!" <<
        std::endl;
}
```

About functions

Functions allow reusing portions of code. They have:

- A declaration
- A definition

‘Calling a function’ means using it.

```
void print();


void print() {
    std::cout "Hello, World!" <<
        std::endl;
}


print();
```

02 Parameterizing functions

Arguments

Arguments

Functions can take **arguments**. It's called *passing argument(s)* to a function.

Arguments

Functions can take **arguments**. It's called *passing argument(s)* to a function.

```
void print(std::string message) {
    std::cout << message << std::endl;
}
```

Arguments

Functions can take **arguments**. It's called *passing argument(s)* to a function.

```
void print(std::string message) {
    std::cout << message << std::endl;
}
void two_lines(std::string line1, std::string line2) {
    std::cout << line1 << std::endl;
    std::cout << line2 << std::endl;
}
```

Arguments

Functions can take **arguments**. It's called *passing argument(s)* to a function.

```
void print(std::string message) {
    std::cout << message << std::endl;
}
void two_lines(std::string line1, std::string line2) {
    std::cout << line1 << std::endl;
    std::cout << line2 << std::endl;
}
```

```
print("Hello, World!");
two_lines("Hello,", "World!");
```

Arguments

Functions can take **arguments**. It's called *passing argument(s)* to a function.

```
void print(std::string message) {
    std::cout << message << std::endl;
}
void two_lines(std::string line1, std::string line2) {
    std::cout << line1 << std::endl;
    std::cout << line2 << std::endl;
}
```

```
print("Hello, World!");
two_lines("Hello,", "World!");
```

```
std::string msg1 = "Hello"
std::string msg2 = "world";
two_lines(msg1, msg2);
```

Practise time!

Value vs Reference

When passing an argument to a function, you either let that function:

Value vs Reference

When passing an argument to a function, you either let that function:

- copy the argument's value: any modification to the argument done in the function won't be effective outside of it;

```
void plus_one(int value) {
    value = value + 1;
    std::cout << value << std::endl;
}
int myvar = 4;
plus_one(myvar);
std::cout << myvar << std::endl; // prints 4
```

Value vs Reference

When passing an argument to a function, you either let that function:

- take the argument's reference: modification of the variable in the function's scope will be effective outside of it.

```
void plus_one(int& value) {
    value = value + 1;
    std::cout << value << std::endl;
}
int myvar = 4;
plus_one(myvar);
std::cout << myvar << std::endl; // prints 5
```

Passing by value, in more depth

When passing by value, a local variable is created in the function, and initialized with the given argument's value.

Passing by value, in more depth

When passing by value, a local variable is created in the function, and initialized with the given argument's value.

```
void plus_one(int local_var) {
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

Passing by value, in more depth

When passing by value, a local variable is created in the function, and initialized with the given argument's value.

```
void plus_one(int local_var) {
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

```
int myvar = 4;
plus_one(myvar);
```

Passing by value, in more depth

When passing by value, a local variable is created in the function, and initialized with the given argument's value.

```
void plus_one(int local_var) {
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

```
void plus_one(int local_var) { // local_var
    initialized with value of myvar
```

```
int myvar = 4;
plus_one(myvar);
```

Passing by value, in more depth

When passing by value, a local variable is created in the function, and initialized with the given argument's value.

```
void plus_one(int local_var) {
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

```
int myvar = 4;
plus_one(myvar);
```

```
void plus_one(int local_var) { // local_var
    initialized with value of myvar
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

Passing by value, in more depth

When passing by value, a local variable is created in the function, and initialized with the given argument's value.

```
void plus_one(int local_var) {
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

```
int myvar = 4;
plus_one(myvar);
```

```
void plus_one(int local_var) { // local_var
    initialized with value of myvar
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
} // local_var is deleted
// but myvar subsists
```

Passing by reference, in more depth

When passing by reference, the function's variable is like an alias of the given argument.

Passing by reference, in more depth

When passing by reference, the function's variable is like an alias of the given argument.

```
void plus_one(int& local_var) {
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

Passing by reference, in more depth

When passing by reference, the function's variable is like an alias of the given argument.

```
void plus_one(int& local_var) {
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

```
int myvar = 4;
plus_one(myvar);
```

Passing by reference, in more depth

When passing by reference, the function's variable is like an alias of the given argument.

```
void plus_one(int& local_var) {
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

```
void plus_one(int& local_var) { // local_var
     and myvar denote the same thing
int myvar = 4;
plus_one(myvar);
```

Passing by reference, in more depth

When passing by reference, the function's variable is like an alias of the given argument.

```
void plus_one(int& local_var) {
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

```
int myvar = 4;
plus_one(myvar);
```

```
void plus_one(int& local_var) { // local_var
     and myvar denote the same thing
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

Passing by reference, in more depth

When passing by reference, the function's variable is like an alias of the given argument.

```
void plus_one(int& local_var) {
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
}
```

```
int myvar = 4;
plus_one(myvar);
```

```
void plus_one(int& local_var) { // local_var
     and myvar denote the same thing
  local_var = local_var + 1;
  std::cout << local_var << std::endl;
} // the symbol local_var is deleted
// myvar subsists, and its value changed
```

Practise time!

Default arguments 1/3

Some arguments can get a *default* value. If the caller doesn't provide a value for the default argument, it will take its default value. Otherwise, the argument takes the user-provided value.

Default arguments 1/3

Some arguments can get a *default* value. If the caller doesn't provide a value for the default argument, it will take its default value. Otherwise, the argument takes the user-provided value.

```
void clamp(int pixel, int low = 0, int high = 25);
```

Default arguments 1/3

Some arguments can get a *default* value. If the caller doesn't provide a value for the default argument, it will take its default value. Otherwise, the argument takes the user-provided value.

```
void clamp(int pixel, int low = 0, int high = 25);

clamp(r); // low = 0, high = 255
```

Default arguments 1/3

Some arguments can get a *default* value. If the caller doesn't provide a value for the default argument, it will take its default value. Otherwise, the argument takes the user-provided value.

```
void clamp(int pixel, int low = 0, int high = 25);

clamp(r); // low = 0, high = 255
clamp(r, 10); // low = 10, high = 255
```

Default arguments 1/3

Some arguments can get a *default* value. If the caller doesn't provide a value for the default argument, it will take its default value. Otherwise, the argument takes the user-provided value.

```
void clamp(int pixel, int low = 0, int high = 25);

clamp(r); // low = 0, high = 255
clamp(r, 10); // low = 10, high = 255
clamp(r, 0, 128); // low = 0, high = 128
```

Default arguments 2/3

Default arguments are positionned last in the list of arguments. Consider :

Default arguments 2/3

Default arguments are positionned last in the list of arguments. Consider :

```
void foo(int arg1, int default1 = 9, int arg2);
```

Default arguments 2/3

Default arguments are positionned last in the list of arguments. Consider :

```
void foo(int arg1, int default1 = 9, int arg2);
foo(1,2); // who takes the value 2? default1 or arg2 ?
// ambiguous, so we forbid it
```

Default arguments 3/3

Default arguments are only specified in the function's declaration.
If you split the declaration from the definition, you don't have to repeat the arguments default values:

Default arguments 3/3

Default arguments are only specified in the function's declaration.
If you split the declaration from the definition, you don't have to repeat the arguments default values:

```
void score_song(std::string song_name, bool verbose =
    False);

void score_song(std::string song_name, bool verbose) {
    // verbose is still a default argument
   // function body here
}
```

Little practise time!

03 Getting results from functions

With references

With references

So far, you can get information back from a function through arguments passed by reference:

```
void plus(int a, int b, int& res) {
    res = a + b;
}

int res;
plus(1,2,res);
std::cout << res; // prints 3
```

Function’s return value

Functions can *return* a value.

Let’s look at that example

Function's signature

When declaring a function, you are declaring its *signature*.

return_type *function_name* (arg1_type argument1, . . .);

Function's signature

When declaring a function, you are declaring its *signature*.

return_type *function_name* (arg1_type argument1, . . .);

int *multiply* (int a, int b);

Function's signature

When declaring a function, you are declaring its *signature*.

return_type *function_name* (arg1_type argument1, . . .);

int *multiply* (int a, int b);

void *print* (const std::string message);

Function's signature

When declaring a function, you are declaring its *signature*.

return_type *function_name* (arg1_type argument1, . . .);

int *multiply* (int a, int b);

void *print* (const std::string message);

int *random* ();

Functions's exit point(s)

To return a value from a function, you use the `return` directive.

You can return from a function at different points in your program.

Functions's exit point(s)

To return a value from a function, you use the `return` directive.

You can return from a function at different points in your program.

```
std::string genre_label(const Genre genre){
    switch(genre){
        case Genre::Jazz:
            return "Jazz";
        case Genre::Rock:
            return "Rock";
        case Genre::Classical:
            return "Classical";
        default:
            return "Unknown";
    }
}
```

Functions early exit

One can return in the middle of a function, with a conditional statement, and/or during a loop.

Functions early exit

One can return in the middle of a function, with a conditional statement, and/or during a loop.

```
int complicated_get_4(int low = 0, int high = 10){
    for(auto number = 0; number < 10; ++number){
        if(number == 4)
            return number;
    }
    return 4;
}
```

Functions early exit

One can return in the middle of a function, with a conditional statement, and/or during a loop.

```
int complicated_get_4(int low = 0, int high = 10){
    for(auto number = 0; number < 10; ++number){
        if(number == 4)
            return number;
    }
    return 4;
}
complicated_get_4();
```

Functions early exit

One can return in the middle of a function, with a conditional statement, and/or during a loop.

```
int complicated_get_4(int low = 0, int high = 10){
    for(auto number = 0; number < 10; ++number){
        if(number == 4)
            return number;
    }
    return 4;
}
complicated_get_4();
complicated_get_4(7,12);
```

Functions early exit

One can return in the middle of a function, with a conditional statement, and/or during a loop.

```
int complicated_get_4(int low = 0, int high = 10){
    for(auto number = 0; number < 10; ++number){
        if(number == 4)
            return number;
    }
    return 4;
}
complicated_get_4();
complicated_get_4(7,12);
```

Don't forget `return` in your non-`void` functions, or your program might crash!

Practise time!

04 One function, Multiple definitions

Adding numbers

```
int add(int a, int b) { return a+b; }
add(1, 2);
```

Adding numbers

```
int add(int a, int b) { return a+b; }
add(1, 2);

// add(1, 2, 3); ?
```

Solution here

Overloading functions 1/2

A function can have multiple definitions: they’re called **overloads**.

Overloading functions 1/2

A function can have multiple definitions: they’re called **overloads**.

You can *overload* a function by:

- Using a different amount of arguments

Overloading functions 1/2

A function can have multiple definitions: they’re called **overloads**.

You can *overload* a function by:

- Using a different amount of arguments
- Changing the arguments types

Overloading functions 1/2

A function can have multiple definitions: they’re called **overloads**.

You can *overload* a function by:

- Using a different amount of arguments
- Changing the arguments types
- Using a different amount of arguments with differing types

Overloading functions 2/2

Subtleties:

You can overload a function by:

- By **changing the constness** of arguments that are passed by **reference**

Overloading functions 2/2

Subtleties:

You can overload a function by:

- By **changing the constness** of arguments that are passed by **reference**

You **cannot** overload a function by:

- Using arguments passed by **value** vs using arguments passed by **reference**

Overloading functions 2/2

Subtleties:

You can overload a function by:

- By **changing the constness** of arguments that are passed by **reference**

You **cannot** overload a function by:

- Using arguments passed by **value** vs using arguments passed by **reference**

The idea is to wonder if you'd be able to call a specific overload without ambiguity — that is, you can find a specific situation where only one overload matches.

Overload through constness

You can overload with const references because there will be case where only one overload will be legal.

```
int add(int& a, int& b) {
    return a + b;
}
int add(const int& a, const int& b) {
    return a + b;
}

const int a = 1, b = 2;
add(a,b); // ok: a and b are const, calls second overload
```

Overload through constness

But what about cases where both can be used ?

```
int add(int& a, int& b) {
    return a + b;
}
int add(const int& a, const int& b) {
    return a + b;
}

int a = 1, b = 2;
add(a,b); // ?
```

Ambiguity of references

The standard will prefer the candidate that is closest to the caller's capability.

Ambiguity of references

The standard will prefer the candidate that is closest to the caller's capability.

```
int add(int& a, int& b) {
    return a + b;
}
int add(const int& a, const int& b) {
    return a + b;
}

int a = 1, b = 2; // a and b are mutable
add(a,b);
```

Ambiguity of references

The standard will prefer the candidate that is closest to the caller's capability.

```
int add(int& a, int& b) {
    return a + b;
}
int add(const int& a, const int& b) {
    return a + b;
}


int a = 1, b = 2;
add(a,b); // we prefer the first overload because it takes mutable
    references
```

Practise time!

05 Everything about scope

Local variables

A *local* variable is a variable that is accessible only in the perimeter of a block or function.

Local variables

A *local* variable is a variable that is accessible only in the perimeter of a block or function.

```
void a_function() {
    int local_var;  // local to a_function
}

int main(){
    {
        int local_var; // local in current block
    }
    int local_var; // local in main
}
```

Arguments passed by-value

Arguments passed by value to a function are local variables to that function.

Arguments passed by-value

Arguments passed by value to a function are local variables to that function.

```
void a_function(int a) {
```

Arguments passed by-value

Arguments passed by value to a function are local variables to that function.

```
void a_function(int a) {
    // a is declared here and initialized to whatever value was
        given to a_funtion
}

int main(){
    int a = 9; // a is local to main
    a_function(a); // main's a value is copied into a_function's a.
        They are two seperate spaces in memory.
}
```

Global variables

A *global* variable is accessible everywhere in the code, from the moment it's been declared. It is declared outside of any function or block.

Global variables

A *global* variable is accessible everywhere in the code, from the moment it's been declared. It is declared outside of any function or block.

```
void foo() {
    // glob is not accessible here
}
const int glob = 10;
void bar () {
    std::cout << glob << std::endl; // ok, glob exists
}
int main() {
    std::cout << glob << std::endl; // ok, glob exists
}
```

Global variables

A *global* variable is accessible everywhere in the code, from the moment it's been declared. It is declared outside of any function or block.

```
void foo() {
    // glob is not accessible here
}
const int glob = 10;
void bar () {
    std::cout << glob << std::endl; // ok, glob exists
}
int main() {
    std::cout << glob << std::endl; // ok, glob exists
}
```

Global variables are usually used for **constants**.

Variable shadowing

A local variable will *shadow* a global variable that has the same name.

Variable shadowing

A local variable will *shadow* a global variable that has the same name.

```
const int glob = 10;
void foo () {
    std::string glob = "Hello"; // local glob is now Hello
    std::cout << glob << std::endl; // prints Hello
}
int main() {
    foo(); // prints hello
    // glob is still 10
    int glob = 9;
    // local glob is 9
    // global glob is still 10
}
```

Accessing shadowed global variable

A global variable that is shadowed can be accessed by appending the prefix `::`.

Accessing shadowed global variable

A global variable that is shadowed can be accessed by appending the prefix `::`.

```
const int glob = 10;
void foo () {
    std::string glob = "Hello"; // local glob
    std::cout << ::glob << std::endl; // prints 10
}
int main() {
    foo(); // prints 10
    int glob = 9;
    ::glob = 12;
    std::cout << ::glob << std::endl; // 12
    std::cout << glob << std::endl; // 9
}
```

Practise time!

06 The main function

So, `main` is a function, right?

We've covered that `main` is the entry point of your program.

So, `main` is a function, right?

We've covered that `main` is the entry point of your program.

When your program starts, it automatically calls `main`.

Arguments to main

Surprise:

Arguments to main

Surprise: `main` can take arguments.

Arguments to main

Surprise: `main` can take arguments.

```
int main(int argc, char * argv[]) {
    return 0;
}
```

Arguments to main

Surprise: `main` can take arguments.

```
int main(int argc, char * argv[]) {
    return 0;
}
```

Let's see what this means

`main`'s signatures

`main` can only have one of the two following signatures:

```
int main();
int main(int argc, char * argv[]);
```

`main`'s signatures

`main` can only have one of the two following signatures:

```
int main();
int main(int argc, char * argv[]);
```

There can be only one `main` per program.

Enforcing program usage

Often, when a program expects a specific amount of argument, it will print the way the program is expected to be used when the call didn't provide the right amount of arguments.

Enforcing program usage

Often, when a program expects a specific amount of argument, it will print the way the program is expected to be used when the call didn't provide the right amount of arguments.The corresponding code is :

```
int main(int argc, char * argv[]) {
    if(argc != 3 ){ // program expects 2 arguments from
       user
        std::cout << "usage: ./program <arg1> <arg2>" <<
            std::endl;
    }
}
```

Optional parameters

Mandatory parameters are generally signaled in between herringbones <>.
Optional parameters are generally signaled in between brackets `[]`.

Optional parameters

Mandatory parameters are generally signaled in between herringbones <>.
Optional parameters are generally signaled in between brackets [].

```
int main(int argc, char * argv[]) {
    if(argc < 3 || argc > 4 ){ // program expects at least 2 and at
        most 3  arguments from user
         std::cout << "usage: ./program <arg1> <arg2> [optional1]" <<
              std::endl;
    } else if(argc == 4) {
        // do something special when three arguments are provided
    }
}
```

Practise Time!

See you next time!