Functions

They look like this:

```
return_type function_name(list_of_parameters) {
 ...statments and expressions...
}
```

Note that this is the *definition* of a function. We can also *declare* it.

```
return_type function_name(list_of_parameters);
```

return_type can be *void*. In this case nothing is returned.

Definition:

In [ ]:
```
/**
 * Calculate the total volume of rain water
 * @param raindrops number of raindrops
 * @param dropVol average volume of a drop
 * @return total volume of water
 **/
double waterVolume(long long raindrops, double dropVol) {
    auto totalVolume = raindrops * dropVol;
    return totalVolume;
}
```

Declaration:

In [ ]:
```
double waterVolume(long long raindrops, double dropVol);
```

How do we use the previous function?

Calling code:

In [ ]:
```
auto volume = waterVolume(45563, 5.77);

long long drops = 100;
double volPerDrop=500;
waterVolume(drops, volPerDrop);
```

The *declaration* is also called *signature* of a function. You can declare in several ways:

In [ ]:
```
double waterVolume(long long raindrops, double dropVol);
double waterVolume(long long raindrops, double dropVol=1.3);
double waterVolume(long long, double);
```

But why do we need a declaration?

Cpp File and H File

Separate declarations from definitions in each file.

RedBugs.h

In [ ]:
```
#ifndef REDBUGS_H_
#define REDBUGS_H_

namespace redbugs {
     int someFunction();
}

double anotherFunction(int m);

#endif
```

RedBugs.cpp

In [ ]:
```cpp
#include "RedBugs.h"

int redbugs::someFunction() {
    return 8;
}

double anotherFunction(int m){
    return m*4.35;
}
```

Usually the main function will go in another file

`main.cpp`

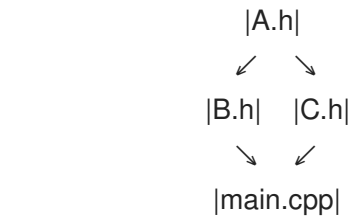
In [ ]:
```cpp
#include <iostream>
#include "RedBugs.h"

int main() {
    std::cout << redbugs::someFunction() << std::endl;
    std::cout << anotherFunction(4) << std::endl;
    return 1;
}
```

Why do we need the compiler directives?

```
#ifndef ...
#define ...
//...
#endif
```

This is a guard mechanism used to prevent multiple inclusions of the same header file in a source file (cpp file).



Compiling:

```
g++ main.cpp -c
g++ -std=c++20 RedBugs.cpp -c
g++ main.o RedBugs.o -o redbug.x
```

or directly:

```
g++ main.cpp RedBugs.cpp -o redbug.x
```

and execute:

```
./redbug.x
```

Calling

1. Calling by reference
    - Using pointers
    - Using reference
2. Calling by value

Calling by reference

Changes inside the function of a parameter by reference will be seen after calling the function.

Declaration:

```
void recipe(int &numApples);
```

Calling:

```
int apples = 10;
recipe(apples);
```

Calling by value

A copy of numApples will be done, and no change will be seen after calling the function.

Declaration:

```
void recipe2(int numApples);
```

Calling:

```
int apples = 10;
recipe2(apples);
```

In [ ]:
```
void count(int money){
    // does something...
    money--;
}
//Calling code:
int mymoney = 100;
count(mymoney);
```

Question: will mymoney be equal to 99?

✔ ✗

In [ ]:
```
// Notebook interpreter only
mymoney
```

In [ ]:
```
void charge(int & money){
    // other statements
    money-=20;
}

//Calling code:
int money = 100;
charge(money);
```

Question: will money be equal to 80?

✔ ✗

In [ ]:
```
// Notebook interpreter only
money
```

Main Function

The main function is the starting point of your program. There is no more and no less than one main function in your code.

Its simplest form is:

```
int main( ){
    // expressions and
    // statements
    return 0; //or another int
}
```

Accepting parameters from "outside" :

```
int main(int argc , char ** argv){
    // expressions and
    // statements
    return 0; //or another int
}
```

Same as above, but parameters are declared differently:

```
int main(int argc , char * argv []){
    // expressions and
    // statements
    return 0; //or another int
}
```

Lambda Functions

C++11

Lambda functions allow you to define small anonymous functions. They can be useful in situations where you need a function that is only used in one place or for simple transformations of data.

[*capture_list*](*parameter_list*) -> *return_type* { *function body* }

1. capture list: an optional list of variables to capture from the enclosing scope. It specifies which variables the lambda function can access and whether they are captured by value or by reference.
2. parameter list: a comma-separated list of parameters that the lambda function takes.
3. return type: the return type of the lambda function.
4. function body: the body of the lambda function, which contains the code to be executed when the function is called.

Example:

In [ ]:
```
auto getFoo() {
  int a = 5;
  return [a](int x, int y) {
     return a * x * y;
  };
}
```

Other variations below, which give the same result. (Note, don't run them on the notebook)

In [ ]:
```
// variations:
auto getFoo() {
  int a = 5;
  return [a](int x, int y) -> int {
     return a * x * y;
  };
}

auto getFoo() {
  int a = 5;
  return [=](int x, int y) -> decltype(a*x*y) {
     return a * x * y;
  };
}

auto getFoo() {
  int a = 5;
  return [a](int x, int y) -> decltype(auto) {
     return a * x * y;
  };
}
```

In [ ]:
```
#include <iostream>
using namespace std;

decltype(getFoo()) mylambda = getFoo();
//auto mylambda = getFoo();

cout << mylambda(2, 3) << endl;
```

In [ ]:
```
```