Functions

There are three steps in using names functions:

1. Prototype the function
2. Define the function
3. Call the function

A function prototype gives type information for the function but does not define it. A function definition also provides type information but define what the function does

Examples of prototypes

```
int a_func(int n);
int b_func(int n);
```

We can also define prototypes without specifying the argument names, i.e. `int a_func(int)`.

Typically included in a header file using

```
#include "my_proj.h"
```

Defining a function

All function definitions are also declarations, they also include statements that define what the function does

```
return_type name(args) {
    statements
}
```

- Unless a function has a `void` return type, it should always return some data.
- Arguments are passed by value, meaning functions get their own copy of the data. Arrays are the exception
- Variables declared in functions are local
  - They have automatic storage, meaning memory is allocated when the function is called, and removed when the function terminates.
  - The `static` keyword can be used to persist the variable through the life of the program

Complete function syntax

```
[modifiers] return_type name(args) [override] [const] [final]
```

Modifiers

- `const`: cannot modify any of the class data. In effect the function is read-only.

- `constexpr`: compiler treats return values from the functions as compile-time constants
- `explicit`: when applied, the constructor function is not used to supply an implicit conversion function.
- `final`: applied to virtual functions, prevents the function from being overridden by derived classes.
- `inline`: suggestion to complier to make the function inline, when a function is inlined the body of the function is expanded into the body of the calling function, removing function-call overhead.
- `override`: member functions only, specifies that the function overrides a function specified in the base class.
- `static`: when applied to a global function is visible with the current source file only. When applied to a member function, the function becomes static, so that it cannot access other class members except those declared as static.
- `virtual`: address not resolved until runtime

Function overloading

- Function can have the same name as long as they can be differentiated by their argument lists.

In some cases a function can be matched to more than one call, for example

```
void func(int n);
void func(double x);
```

The compiler will make the best match it can. These functions can have different return types. Return types are not used to match the function.

Variable-Length Argument Lists

```
return_type name(args ...)
```

Functions can be defined to take any number of arguments. Include

```
#include <cstdarg>
```

in sources files to use these kinds of functions. Argument processing functions include:

- `va_list`: declares a list name
- `va_start`: initialises the argument list, up to some last named argument
- `va_arg`: returns the next argument from the variable argument list
- `va_end`: terminates reading the argument list

Lambda functions

```
[closure] (args) { statements }
```

`[]` intended literally, `[closure]` is an optional list of variables.

Return types are usually implicit but can be specified with

```
[closure] (args) -> return_type { statements }
```

Closures allow for variables defined in a block to be captured by the lambda expression and used internally without explicitly being passed as an argument.

Possible closure values include:

- `[]`: capture nothing
- `[=]`: capture all local variables declared in the surrounding code, but only by value
- `[&]`: all variables in surrounding scope captured by reference
- `[=, args]`: all variables captured by value, except for any args.
- `[&, args]`: all variables captured by reference, except for any args.

Mutable keyword

```
[closure] (args) mutable -> return_type { statements }
```

The `mutable` keyword is used to indicate that variables can be changed, but only as temporary variables within the lambda. Changes has no effect outside the lambda.

`constexpr` Functions

`constexpr` keyword causes the compiler to treat calls to the function as runtime constants if possible. Calls are treated as constants if the arguments supplied are also constants. These values are computed at compile time (i.e. before the program is ever run).

Storing and Returning Lambdas

When a lambda is stored variables ate captured at the point the lambda is defined. If a variable is captured by reference then a fixed value is not captured. Functions can be written that return a lambda which is done using the `<functional>` header. Ex:

```
function<return_type(arg_type_list)> function_name(args)
```