Functions, templates, functionals, and lambdas in C++

While this is specific to c++, some general ideas hold for many languages (except templates)

1. Recap: function arguments; pass by value (copy), reference, pointer

Pass by value (aka pass by copy):

- Gets a new copy of variable
- Changes to this variable inside function do not affect original variable

```
double sum_by_value(double x, double y) {
  // Gets own copy of x and y
  x += y;
  return x;
}
```

Pass by reference:

- Gets a *reference* to existing variable
- Changes to this variable inside function **do** affect original variable

```
double sum_by_reference(double &x, double &y) {
  // Get *reference* to x and y - operate on existing variable!
  x += y;
  return x;
}
```

Pass by *const* reference:

- Gets a *reference* to existing variable (so, no copy)
- Changes to this variable inside function are not allowed

```
double sum_by_constref(const double &x, const double &y) {
  // x += y; // would not compile, cannot modify 'const' variable
  return x + y;
}
```

Why?

- When passing around simple data (e.g., an `int`, `double`) usually doesn't matter
- If instead we are passing large data structures (e.g., an entire matrix), we don't want to unnecessarily copy all this data (this is slow)
  - Sometimes we *do* want to copy the data, but this allows us to control when that happens
- Beware passing by reference -- errors caused by accidentally modifying input can be hard to debug.

- Usually, we pass by reference to avoid copying the data. We normally don't want to modify the input data inside the function. We can enfore this by passing by `const reference`
    - Best practise rule: *prefer pass by const reference for large objects, value for small*
- You can also pass by pointer, const pointer, etc. etc.

Example:

Interactive version: https://godbolt.org/z/xcWsjbnWs

```
#include <iostream>
// ... include 3 above functions ...
int main() {
  double x = 2.0;
  double y = 3.5;

  std::cout << "x=" << x << ", y=" << y << "\n";

  double result1 = sum_by_value(x, y);
  std::cout << "x=" << x << ", y=" << y << ", result=" << result1 << "\n";

  double result2 = sum_by_reference(x, y);
  std::cout << "x=" << x << ", y=" << y << ", result=" << result2 << "\n";

  double result3 = sum_by_constref(x, y);
  std::cout << "x=" << x << ", y=" << y << ", result=" << result3 << "\n";
}
```

Output:

```
x=2, y=3.5
x=2, y=3.5, result=5.5
x=5.5, y=3.5, result=5.5
x=5.5, y=3.5, result=9
```

Optional/Default arguments

- We can specify optional function arguments
- They must be at the end of the input list
- We must give them a default value in the function signature (declaration)

```
// declare function:
double func(double x, double y, double z = 2.5);
// z is optional - if not given, will assume value is 2.5

// Then, on calling the function:
func(3.1, 2.7);      // call with default argument
func(3.1, 2.7, 2.5); // exactly same as above
func(3.1, 2.7, 9.5); // call with different argument
```

Recursive functions

- We can write functions which call themselves
- Often very useful; but be careful not get stuck in an infinite loop - must have an exit condition
- Consider this simple example that calculates x^n:

```
double my_pow(double x, int n) {
  if (n == 0) {
    return 1.0;
  } else if (n < 0) {
    return 1.0 / my_pow(x, -n);
  } else {
    return x * my_pow(x, n - 1);
  }
}
```

---

2. Function overloading, and templates

Say we want a function that will work with more than one type, for example:

```
double sum(double a, double b) { return a + b; }
int sum(int a, int b) { return a + b; }
```

- C++ allows *function overloading* -- where the same function name can be used for different arguments (not allowed in C)
- Which version of the function will be called will depend on the *type* of input variable (this has potential for confusion, and hard-to-debug errors)
- This example may appear silly, since int can be converted to double; however, it becomes important for more complex types that cannot be so easily converted between (e.g., an array of `int` cannot be simply converted to an array of `double`) - we may also want to avoid conversions for performance/correctness reasons
- This gets tedious quickly - there are an infinite number of types (since types can be user defined)
- To be more generic, we may use *templates*

```
template <typename T>
T sum(T a, T b) {
  return a + b;
}
```

- Here, 'T' is the name of a generic type - 'T' may be anything (called 'T' by convention, but can be anything; can be more than one, e.g., `<typename T, typename U>` etc.)
- Technically, this will generate code for you, at compile time; it will write each function overload for you, based on which types you actually use in your code
- Must be defined either in a header file, or in the same .cpp file you use them
    - (They must be visible in each 'translation unit')
- There are fancy ways to restrict which types T is allowed to take -- but we will ignore this complexity for now.
    - Look up 'type traits' if you're interested!
- This is just an extremely basic introduction to templates; the template system in c++ is its own entire language, and is extremely powerful (though sometimes difficult to use)

Example:
Interactive version: https://godbolt.org/z/6974asEax

```cpp
#include <iostream>
#include <string>

// Declare a template: T is generic type, may be any type
template <typename T>
T sum(T a, T b) {
  return a + b;
}

int main() {
  int i1 = 1;
  int i2 = 2;
  int result1 = sum(i1, i2);  // calls sum(int, int)

  double d1 = 1.01;
  double d2 = 2.02;
  double result2 = sum(d1, d2);  // calls sum(double, double)

  // Even works with strings! (this may not be what you want)
  // Works with any type for which (a+b) is defined
  std::string s1 = "Hello ";
  std::string s2 = "world!";
  std::string result3 = sum(s1, s2);  // calls sum(string, string)

  std::cout << result1 << " " << result2 << " " << result3 << "\n";

  // auto result = sum(i1, d1);  // This will not compile!
  // i1 is int, d1 is double. So c++ cannot deduce what T should be
  // If you must, you can force it, be explicitely stating the type using `<>`:
  auto result4 = sum<double>(i1, d1);
  // This will compile, but will -Wconversion warning,
  // since i1 is being converted to a double
}
```

---

3. Functionals: Passing functions to functions

- Sometimes it is extremely useful to have a function that takes another function as one of its arguments
- For example, we may want a function called 'integrate' that takes a function, `f(x)`, and integrates it between `x = [a,b]`, e.g.:

```cpp
// nb: This doesn't work quite yet..., because there is no 'Function' type
double integrate(Function f, double a, double b); //??
```

Can we do this? Yes! But not quite so simply. There are three key ways to do this: function pointers, templates, and using c++ library std::function
(The templates case is really the same as function pointers, since `T` will be deduced as a fn pointer.)

..using function pointers

- Old; we typically try to avoid this, because complicated
- We can pass the memory address of a function

    - In c++, a function name converts implicitly to the memory address of a function (function pointer), so we do not need to use the `&` operator (though, we can)
- For a function:

```
OutType funcName(InType1 x, InType2 y, ...);
```

- Function pointer has the form:

```
OutType (*funcName)(InType1, InType2, ...)
```

- So, we could write our 'integrate' function as

```
double integrate(double (*f)(double), double a, double b);
```

- and call it like:

```
double result = integrate(f, a, b);
// double result = integrate(&f, a, b); // equivalent to above
```

..using templates

- Since a template can be *any* type (including function pointer), this gives a simple way to pass functions to functions
- This is powerful, however, it is complex; if you get the code wrong, sometimes the error messages will be extremely hard to decipher
- This is commonly seen in code - probably because it uses the least typing (not a great reason, but hey)
- We could write our 'integrate' function as

```
template<typename Function>
double integrate(Function f, double a, double b);
```

And then use it in the exact same way as above

..using std::function

- C++ provides a general class to hold a function (called *function objects*, or *callables*)
- Requires c++11 or newer; you may need to add -std=c++11 compile option
- need: `#include <functional>`
- Avoids complexity of using templates and function pointers
- When problems happen, usually get nice error messages
- Has type of form

```
std::function<OutType(InType1, InType2, ...)>
```

- Often used with `using` keyword (like an alias) to save typing

```
using FuncType = std::function<OutType(InType1, InType2, ...)>;
// Then, just use 'FuncType' as the type when needed
```

- e.g., our f(x) = x^2 function would be simply:

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

Example:

Functions that takes a function and integrates it (trapezoid rule) - using the three methods form above.

Interactive version: https://godbolt.org/z/KGnMKbxPc

```
#include <functional>
#include <iostream>

// Simple function, f(x)=x^2, which we will integrate
double f(double x) { return x * x; }

// Function that integrates another function; uses function pointer
double integrate_fp(double (*f)(double), double a, double b) {
  int n_pts = 100;
  double dx = (b - a) / (n_pts - 1);
  double integral = (f(a) + f(b)) * (dx / 2.0);
  for (int i = 1; i < n_pts - 1; ++i) {
    double x = a + i * dx;
    integral += f(x) * dx;
  }
  return integral;
}

// ...; uses templates
template <typename Function>
double integrate_tmpl(Function f, double a, double b) {
// ... same as above ...
}

// ...; uses std::function
double integrate_std(std::function<double(double)> f, double a, double b) {
// ... same as above ...
}

int main() {
  double exact = 2.0 / 3;
  double result1 = integrate_fp(f, -1.0, 1.0);
  double result2 = integrate_tmpl(f, -1.0, 1.0);
  double result3 = integrate_std(f, -1.0, 1.0);
  std::cout << exact << ", " << result1 << ", " << result2 << ", " << result3 << "\n";
}
```

---

4. Lambdas

- Lambdas are "anonymous" inline functions
  - ('anonymous' is a little confusing, since they can have names..)
- Requires c++11 or newer; you may need to add -std=c++11 compile option
- A nice way to define (usually short) functions inline (inside `main()`)
- They are often passed as input to other functions (like STL standard algorithms)
- Have general form: `[captures](parameters){function body;}`

    - We'll see what this means below
- For example, a lambda version of our 'f' function from above, which takes a double x and returns a double x*x, is

```
[](double x){ return x * x; }
```

- Captures allow us to include local data in a function - usually there are no captures, so the '[]' are empty

```
double y = 7.5;
[y](double x){ return y * x * x; }
```

- We may also capture by reference:

```
double y = 7.5;
[&y](double x){ return y * x * x; }
```

- You can (though may not want to) capture *everything* in the local scope (be careful with this), either by value or by reference:

```
[=](double x){ ... } // capture all by value
[&](double x){ ... } // capture all by reference
```

- Combining with our 'integrate' function from above, we could have:

```
double result = integrate_std([](double x) { return x * x; }, -1.0, 1.0);
```

- We can give lambdas names, which makes code more readable. But, you *must* use `auto` for the type:

```
auto my_lambda = [](double x) { return x * x; };
double result = integrate_std(my_lambda, -1.0, 1.0);
```

- **Note** Lambdas can 'decay' to function pointers (i.e., you can pass a lambda to a function expecting a function pointer), but **only** if the lambda has no captures.
- If you want to pass a lambda that has captures to a function, you must use std::functional, which can take any lambda

Lambda example

Interactive version: https://godbolt.org/z/j448zW8GG

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    auto l1 = []() { return "Hello world\n"; };
    auto l2 = []() { std::cout << "Hello world\n"; };

    std::cout << l1();
    l2();

    auto l3 = [](double x) { return x * x; };
    std::cout << l3(1.0) << ' ' << l3(2.0) << ' ' << l3(3.0) << '\n';

    int a = 1;
    auto l_value = [a]() { return a; };
    std::cout << a << ' ' << l_value() << ' ' << a << '\n';

    auto l_reference = [&a]() {
        a *= 2;
        return a;
    };
    std::cout << a << ' ' << l_reference() << ' ' << a << '\n';

    // Lambdas are very useful for interfacing with std algorithms:
    std::vector<int> vec{3, 9, -2, 1, 5, -12, 66, 0, 12, -8};

    // use std::sort to sort smallest to largest (default)
    std::sort(vec.begin(), vec.end());
    for (auto &el : vec) {
        std::cout << el << ", ";
    }
    std::cout << '\n';

    // We can define a lamda that instead sorts by absolute value:
    auto compare_by_abs = [](int a, int b) {
        return std::abs(a) < std::abs(b);
    };
    std::sort(vec.begin(), vec.end(), compare_by_abs);
    for (auto &el : vec) {
        std::cout << el << ", ";
    }
    std::cout << '\n';

    // Another commone use-case, is for_each method:
    std::for_each(vec.begin(), vec.end(),
                  [](int i) { std::cout << i << ", "; });
    std::cout << '\n';

    // with c++-17, we can use 'auto' as lambda parameter type
    // (equivilant to using templates)
    // This will need -std=c++17 command-line option to work
    std::for_each(vec.begin(), vec.end(),
                  [](auto i) { std::cout << i << ", "; });
    std::cout << '\n';
}
```