Idiomatic Modern C++ for Linux

Week 5: Generic Programming, Function Templates, Overloading

Today’s agenda

- Function overloading
- Function deletion
- Default args
- Function templates
- Our first custom class
- Examples of operator overloading for our custom class

Motivation

- So far, hasn’t it been annoying writing different functions with different args that do more or less the same thing?
- Today we’ll look at some tools in our toolbelt to address that

```
// isn't it silly how we're having to define
// a different print function for each type?
void print_integer(int i)
{
    cout << i << endl;
}

void print_double(double d)
{
    cout << d << endl;
}
```

Function overloading

- We can give multiple functions the same name *but the compiler will differentiate them by their arguments*

```
void print(int i)
{
    cout << i << endl;
}

void print(double d)
{
    cout << d << endl;
}
```

```
// compiler knows to call void print(int i)
print(12);
// compiler knows to call void print(double d)
print(86.3);
```

Function overloading

- The compiler needs to be able to differentiate the functions otherwise your code won’t compile
- Take this example with 2 versions of double add, but one having an optional argument. There’s still some ambiguity so the compiler complains

```cpp
double add (double a, double b) {
    return a + b;
}

double add (double a, double b, double c = 14.5) {

    return a + b + c;
}
```

```cpp
add (1.4, 5.7);
```

```
[ 16%] Building CXX object CMakeFiles/function_overloading.dir/src/function_overloading.cpp.o
/code-examples/W5-overloading-and-templates/src/function_overloading.cpp: In function 'int main(int, char**)':
/code-examples/W5-overloading-and-templates/src/function_overloading.cpp:22:9: error: call of overloaded 'add(double, double)' is ambiguous
   22 |     add (1.4, 5.7);
      |     ~~~~^~~~~~~~~~
/code-examples/W5-overloading-and-templates/src/function_overloading.cpp:6:5: note: candidate: 'int add(int, int)'
    6 | int add (int a, int b) {
      |     ^~~
/code-examples/W5-overloading-and-templates/src/function_overloading.cpp:10:8: note: candidate: 'double add(double, double)'
   10 | double add (double a, double b) {
      |        ^~~
/code-examples/W5-overloading-and-templates/src/function_overloading.cpp:14:8: note: candidate: 'double add(double, double, double)'
   14 | double add (double a, double b, double c = 14.5) {
      |        ^~~
gmake[2]: *** [CMakeFiles/function_overloading.dir/build.make:76: CMakeFiles/function_overloading.dir/src/function_overloading.cpp.o] Error 1
gmake[1]: *** [CMakeFiles/Makefile2:87: CMakeFiles/function_overloading.dir/all] Error 2
gmake: *** [Makefile:91: all] Error 2
root@2d773ae3bc0f:/code-examples/W5-overloading-and-templates# sc
```

Resolving which overloaded function to call

- The compiler will apply type promotion and type conversion when determining which overloaded function to call
- This can sometimes introduce unintended side-effects

```
void print_int(int x)
{
    cout << x << endl;
}
```

```
print_int(12);
print_int('a'); // promotion of char to int
print_int(false); // promotion of bool to int
```

Function deletion

- Implicit type conversion can sometimes result in undesired outcomes.
- e.g. if we had a function that expects integers, does it *really* make sense for the compiler to let us call the function with a char or a bool?
- We can prevent a function from being called with a certain argument or type by marking it as “delete”

```
void print_int(int x)
{
    cout << x << endl;
}

void print_int(char) = delete;
void print_int(bool) = delete;
```

```
print_int(12);
print_int('a');
print_int(false);
```

```
-- Build files have been written to: /code-examples/W5-overloading-and-templates/build
[ 16%] Building CXX object CMakeFiles/function_overloading.dir/src/function_overloading.cpp.o
/code-examples/W5-overloading-and-templates/src/function_overloading.cpp: In function 'int main(int, char**)':
/code-examples/W5-overloading-and-templates/src/function_overloading.cpp:37:14: error: use of deleted function 'void print_int(char)'
   37 |     print_int('a');
      |     ~~~~~~~~~^~~~
/code-examples/W5-overloading-and-templates/src/function_overloading.cpp:28:6: note: declared here
   28 | void print_int(char) = delete;
      |      ^~~~~~~~~
/code-examples/W5-overloading-and-templates/src/function_overloading.cpp:38:14: error: use of deleted function 'void print_int(bool)'
   38 |     print_int(false);
      |     ~~~~~~~~~^~~~~~~
/code-examples/W5-overloading-and-templates/src/function_overloading.cpp:29:6: note: declared here
   29 | void print_int(bool) = delete;
      |      ^~~~~~~~~
gmake[2]: *** [CMakeFiles/function_overloading.dir/build.make:76: CMakeFiles/function_overloading.dir/src/function_overloading.cpp.o] Error 1
gmake[1]: *** [CMakeFiles/Makefile2:87: CMakeFiles/function_overloading.dir/all] Error 2
gmake: *** [Makefile:91: all] Error 2
```

More about default function arguments

- every parameter to the right of the first default argument must have default arguments
- There's no syntax yet to support explicitly passing overridden values while leaving args to the left default
- default args cannot be redeclared
- default args must be declared before use (it's best practice to put it in the forward declaration)

```
int add(int x = 0, int y, int z); // not allowed
```

```
void some_func(int x = 4, int y = 6, int z = 10) {}

some_func(42);
```

```
int add(int a, int b = 4);
// not allowed: redefinition of default arg b
int add(int a, int b = 17);
```

Templates – motivation

- Writing essentially the same function signature and definition across multiple types is annoying and tedious

```
int max(int a, int b) {
    return (a > b) ? a : b;
}

double max(double a, double b) {
    return (a > b ? a : b);
}
```

Introducing templates and generic programming

- Luckily, C++ has **templates** to solve this problem
- Write a generic function template once, and let the compiler generate function definitions for any applicable type
- Function instantiation is *deferred* to the first time a function template is called with a unique parameter

```
template<typename T>
T max (T a, T b) {
    return (a > b) ? a : b;
}
```

Template instantiation

- Pre-C++17, need to specify the type using angle brackets
- Post-C++17, Class Template Argument Deduction (CTAD) allows the compiler to determine what type the template is
  - Called either like a regular function call, or with an empty template parameter
- Sometimes it may still make sense or read better to specify the type in your expression

```cpp
// instantiates and calls max<int>(int, int)
cout << max<int>(12, 5) << endl;

// instantiates and calls max<double>(double, double)
cout << max(1.4, 6.2) << endl;

// Uses the already instantiated definition max<int>(int, int)
cout << max(1, 64) << endl;

// instantiates and calls max<unsigned>(unsigned, unsigned)
cout << max<>(6U, 49U) << endl;

// instantiates and calls max<char>(char, char)
cout << max('b', 'a') << endl;
```

Multiple template args

- We can also have templates with multiple types.
- In this example, because we would return one of two types, we declare 'auto' as the return type to avoid narrowing conversions

```
template <typename T1, typename T2>
auto max(T1 a, T2 b) {
    return (a > b) ? a : b;
}
```

```
// this particular set of types is a bad idea for this example
cout << max<int, unsigned int>(-5, 4) << endl;
```

```
4294967291
```

Template specialization

- We can explicitly declare a template specialization for a given type
- e.g. in this example, it’s nice to specify that a boolean should be printed with boolalpha

```cpp
template<typename T>
void print(T thing) {
    cout << thing << endl;
}

// template specialization of print() with type bool
template<>
void print(bool thing) {
    cout << std::boolalpha << thing << endl;
}

print(42);
print("secrets");
print(9 + 10 == 21);
```

Templates and project file structure

- Unlike non-template functions, templates can't be forward-declared.
- The model of spec → .hpp, impl → .cpp falls apart
- Your options are
  - define your template inside the header file
  - put the definitions inside a '.ipp' file and #include that in the header
  - specifically instantiate your templated function with type args inside your .cpp file (brittle)

forward_declared_templates.hpp U

code-examples > W5-overloading-and-templates > include > forward_declared_templates.hpp > ...

```hpp
#pragma once

template<typename T>
T mul(T a, T b);

```

forward_declared_templates.cpp 2, U

code-examples > W5-generic-programming-and-overloads > src > lib > forward_declared_templates.cpp > ...

```cpp
#include "forward_declared_templates.hpp"

template<typename T>
T mul(T a, T b) {
    return a * b;
}

// explicit instantiations of mul for int and float.
// if our template definition is in a .cpp file,
// we'll have to do this for every type we intend to use it with
template int mul(int a, int b);
template float mul(float a, float b);
```

```cpp
mul(1, 4);
mul(2.5f, 4.6f);
// mul wasn't instantiated for unsigned int
mul(5U, 7);
```

Crash course on classes and operator overloading

- We’ve alluded in the course to overloading operators in custom types
- Now we’ll create a type that can evaluate at compile time, and implements the operators for +, += and <<

Our custom Point3d class

- Point of type double with x, y, and z
- Constructor with constexpr initialization
- Member methods are defined as constexpr
- Addition returns a Point3d. Incrementing by another Point3d returns a reference to self (since incrementing is inplace)
- member variables are private and (in this example) denoted with the m_ prefix.
- Any function declared as a “friend function” can access its private members. Useful for defining the stream insertion operator for our Point3d

```cpp
class Point3d {
    public:
    constexpr Point3d(double x, double y, double z)
    : m_x(x), m_y(y), m_z(z) {}

    constexpr bool operator==(const Point3d& other) const {
        return
        m_x == other.m_x &&
        m_y == other.m_y &&
        m_z == other.m_z;
    }

    constexpr Point3d operator+(const Point3d& other) {
        return Point3d(
            m_x + other.m_x,
            m_y + other.m_y,
            m_z + other.m_z
        );
    }

    constexpr Point3d& operator+=(const Point3d& other) {
        m_x += other.m_x;
        m_y += other.m_y;
        m_z += other.m_z;
        return *this;
    }

    // We declare the stream insertion operator a "friend" function.
    // It's external to the class, but allowed to access its private members
    friend std::ostream& operator<<(std::ostream& out, const Point3d& pt);

    private:
    double m_x;
    double m_y;
    double m_z;

}; // class Point3d

// Outside of the class, we define the stream insertion operator for Point3d,
// allowing us to use cout on it
std::ostream& operator<<(std::ostream& out, const Point3d& pt) {
    out <<"Point3d (x=" << pt.m_x << ", y=" << pt.m_y << ", z=" << pt.m_z << ")";
    return out;
}
```

Our custom Point3d class

- All of these expressions will evaluate at compile-time because we declared the results as constexpr.

```
constexpr Point3d a (3.5, 4.6, 9.17);
constexpr Point3d b  = Point3d{12.4, 5.7, 7.5} + a;
constexpr Point3d c (3.5, 4.6, 9.17);
constexpr bool these_points_are_the_same{a == c};
constexpr Point3d d = Point3d(1.8, 91.5, 9814.07) += c;

cout << a << endl;
cout << b << endl;
cout << "These points are the same: "<<these_points_are_the_same<<endl;
cout << d << endl;
```

```
Point3d (x=3.5, y=4.6, z=9.17)
Point3d (x=15.9, y=10.3, z=16.67)
These points are the same: 1
Point3d (x=5.3, y=96.1, z=9823.24)
```

Additional Resources

- https://en.cppreference.com/w/cpp/language/operators.html
- https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/
- https://en.cppreference.com/w/cpp/language/class_template_argument_deduction.html
- https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/