Function Overloading

The material in this handout is collected from the following reference:

- Section 6.4 of the text book C++ Primer.

Introduction to overloaded functions

Suppose we author a function to compute the cubed value of an integer:

```
int cube(int n) {
  return n * n * n;
}
```

In addition to computing the cubed values of integers, we use the function to compute the cubed values of values with variety of types:

```
int    i {8};
long   l {50L};
float  f {2.5F};
double d {3.14};

std::cout << cube(i) << "\n"; // works fine: 512
std::cout << cube(l) << "\n"; // may or may not work: 125000
std::cout << cube(f) << "\n"; // not quite what we want: 8
std::cout << cube(d) << "\n"; // not quite what we want: 27
```

Obviously, the results are not as expected for types other than `int`. The problem with using `cube` to compute the cube of a value of type `long` is that values of type `long` are downgraded to type `int`. Same is the case with floating-point types.

Our first attempt to fix this would entail defining a separate function for each specific type for which cubed values are required:

```
int cube_int(int n) {
  return n * n * n;
}

long cube_long(long n) {
  return n * n * n;
}

float cube_float(float n) {
  return n * n * n;
}

double cube_double(double n) {
  return n * n * n;
}
```

These definitions will work as expected:

```
int    i {8};
long   l {50L};
float  f {2.5F};
double d {3.14};

std::cout << cube_int(i) << "\n";    // works fine: 512
std::cout << cube_long(l) << "\n";   // Works fine: 125000
std::cout << cube_float(f) << "\n";  // works fine: 15.625
std::cout << cube_double(d) << "\n"; // works fine: 30.9591
```

This C-style way of dealing with multiple functions implementing the same algorithm quickly becomes tedious and unmanageable as we write additional functions to handle other types such as `unsigned int`, `unsigned long`, `char`, as well as user-defined types that might come along.

The C++ mechanism that provides notational convenience of using the same name for operations on different types is called *overloading*. The technique is already used for basic operations in C++. That is, there is only one name for addition, `+`, yet it can be used to add values of integer and floating-point types and combinations of such types. This idea is easily extended to functions defined by the programmer and is called *function overloading*. In function overloading, functions can share the same name as long as their parameter declarations are sufficiently different.

Since function overloading eliminates the need to invent – and remember – names that exist only to help the programmer and compiler figure out which function to call, we can define a `cube` function for each specific type:

```
int cube(int n) {
  return n * n * n;
}

float cube(float n) {
  return n * n * n;
}

double cube(double n) {
  return n * n * n;
}

long cube(long n) {
  return n * n * n;
}
```

and call these `cube` functions without needing to choose the right function:

```
int    i {8};
long   l {50L};
float  f {2.5F};
double d {3.14};

std::cout << cube(i) << "\n"; // works fine: 512
std::cout << cube(l) << "\n"; // Works fine: 125000
std::cout << cube(f) << "\n"; // works fine: 15.625
std::cout << cube(d) << "\n"; // works fine: 30.9591
```

Now, if we decide we need to handle another data type, we simply *overload* the `cube` function to handle the new type. Clients using the `cube` library have no idea that we implement `cube` as separate functions. Each call to function `cube` has an argument type that exactly matches the parameter type of a particular definition of function `cube`. Therefore, it is straightforward for the compiler to match a function call to a specific function definition. However, the situation gets complicated when we make the following call:

```
char ch = 'a';
cube(ch);
```

Since a `cube(char)` function was not defined, the compiler will have to choose one of the previously defined `cube` functions or flag the call `cube(ch)` as an error. The next section discusses C++'s response when it encounters such functions.

Overload resolution

Consider the following example:

```
#include <iostream>
#include <cmath>

int divide(int a, int b) {
  return a/b;
}

float divide(float a, float b) {
  return std::floor(a/b);
}

int main() {
  int x = 5, y = 2;
  float n = 5.f, m = 2.f;
  std::cout << divide(x, y) << "\n";
  std::cout << divide(n, m) << "\n";
  std::cout << divide(x, m) << "\n";
}
```

Here we define function `divide` twice: with two `int` and two `float` parameters. When we call `divide`, the compiler performs an *overload resolution*:

1. Is there an overload that matches the argument type(s) exactly? Take it; otherwise:
2. Are there overloads that match *after conversion*? How many?
    1. Zero matches: Error. No matching function found.
    2. One match: Take it.
    3. $>$ one matches: Error. Ambiguous call.

Let's apply overload resolution to our example. The calls `divide(x, y)` and `divide(n, m)` are exact matches. For `divide(x, m)`, no overload matches exactly and both functions match by *implicit conversion* and therefore it is an ambiguous call.

What does implicit conversion mean? When an expression contains operands of built-in types, and no explicit casts are present, the compiler uses built-in standard conversions to convert one of the operands so that the types match. The implicit conversions among the arithmetic types are defined to preserve precision, if possible. Most often, if an expression has both integral and

floating-point operands, the integer is converted to floating-point. The rules governing implicit conversions are detailed and are described here. When we later define our own types, we can implement a conversion from another type to it or conversely from our new type to an existing one. We've seen such a conversion from the type `std::ifstream` to `bool` when we implemented code to compute the average of an unknown number of `int`s from a text file:

```
#include <iostream>
#include <fstream>

int main(int argc, char *argv[]) {
  std::ifstream ifs {argv[1]};
  int sum {0}, n{0}, value;
  while (ifs >> value) {
    sum += value;
    ++n;
  }
  double avg = static_cast<double>(sum)/n;
}
```

More formally, function overloads must differ in their *signature*. The signature consists in C++ of

- The function name;
- The number of parameters, called *arity*; and
- The types of the parameters (in their respective order).

In contrast, overloads varying only in the `return` type or the parameters names have the same signature and are considered as forbidden redefinitions:

```
void f(int x) { /* code */ }
void f(int y) { /* code */ } // redefinition: only argument name
                             // different
long f(int z) { /* code */ } // redefinition: only return type different
```

Even if function overloads differ in their signature, ambiguity can be a problem. Clearly, the following two functions are distinct and can coexist:

```
void foo(double d) { std::cout << "foo(double): " << d << "\n"; }
void foo(float f)  { std::cout << "foo(float): " << f << "\n"; }
```

However, one of the following call is in error:

```
foo(1.0F); // calls foo(float)
foo(1.0);  // calls foo(double)
foo(1);    // which one?
```

Here, the call `foo(1.0F)` matches `foo(float)`; the call `foo(1.0)` matches `foo(double)`; the call `foo(1)` is ambiguous because an `int` value can be implicitly converted to a `float` value or a `double` value.

That functions with different names or *arity* are distinct goes without saying. The presence of a reference symbol turns the parameter type into another type. Thus, functions `f(int)` and `f(int&)` can coexist:

```
void f(int)  { std::cout << "int\n"; }
void f(int&) { std::cout << "int&\n"; }
```

Which function is called in the following code fragment?

```
int i = 1;
int const j = 2;

f(5);      // which one?
f(i);      // which one?
f(j);      // which one?
f(i + j);  // which one?
```

The call `f(5)` matches `f(int)` because `5` is an rvalue; the call `f(i)` is ambiguous because it match `f(int)` and `f(int&)`; the call `f(j)` matches `f(int)` and not `f(int&)` because the parameter of `f(int&)` is an lvalue reference and thus cannot be a reference to `j` (since `j` is a `const int`); the call `f(i+j)` evaluates to an rvalue and therefore matches `f(int)`.

The following three overloads have different signatures:

```
void f(int x)        { /* some code here */ }
void f(int& x)       { /* some code here */ }
void f(int const& x) { /* some code here */ }
```

Although the three function definitions compile, problems will arise when we call `f`:

```
int i {3};
int const ci {4};

f(3);  // ERROR: ambiguous
f(i);  // ERROR: ambiguous
f(ci); // ERROR: ambiguous
```

All three function call are ambiguous: the call `f(3)` matches `f(int)` and `f(int const&)`; the call `f(i)` matches `f(int)`, `f(int&)`, and `f(int const&)`; the call `f(ci)` matches `f(int)` and `f(int const&)`.

Mixing overloads of reference and value parameters almost always fails. Thus, when one overload has a reference-qualified parameter, then the corresponding parameter of the other overloads should be reference-qualified as well. We can achieve this in our example by omitting the value-parameter overload. Then `f(3)` and `f(ci)` will resolve to the overload with the constant reference and `f(1)` to that with the plain reference.

> ***Mixing overloads of reference and value parameters almost always fails. Thus, when one overload has a reference-qualified parameter, then the corresponding parameter of the other overloads should be reference-qualified as well.***

Finally, we can have problems when we mix overloading and default parameters:

```
void bar(int a, int b = 10);
void bar(int a);
bar(5, 6); // ok
bar(5);    // ambiguous
```

Consider the following set of function declarations and a function call:

```
void f();
void f(int);
void f(int, int);
void f(double, double = 3.14);

f(5.6);
```

Analyze the overload resolution mechanism for function call `f`.