CS170#06.2

Overload Resolution

Vadim Surov

Outline

- Function Overloading
- Overload Resolution
- Best Matching
- Exact Matches
- Exact Matches For Arrays
- Type Promotion
- Standard Conversions
- Conversion Constructors
- explicit
- User-Defined Conversions

Function Overloading

- In C++ you can have two or more functions with the same name so long as they differ in their parameter lists. This is called **function overloading**.
- The function is invoked whose parameter list **matches** the arguments in the call.
- Normally the compiler can deal with overloaded functions fairly easily by comparing the arguments with the parameter lists of the candidate functions.
- However, this is not always a straightforward matter.

Function Overloading

- Consider the following code:

```
void foo(double d1, int i1) { }
void foo(double d1, double d2) { }
int main( ) {
   foo(1.0, 2);
   return 0;
}
```

- How does the compiler know which version of foo() to call?

Overload Resolution

- The compiler works through the following checklist and if it still can't reach a decision, it issues an error:
  1. Gather all the functions in the current scope that have the same name as the function called.
  2. Exclude those that don't have the right number of parameters to match the arguments in the call.
     a. It has to be careful about parameters with default values; `void foo(int x, int y = 0)` is a candidate for the call `foo(25);`
  3. If no function matches, the compiler reports an error.
  4. If there is more than one match, select the '**best match**'.
  5. If there is no clear winner of the best matches, the compiler reports an error - ambiguous function call.

Best Matching

- In deciding on the best match, the compiler works on a rating system for the way the types passed in the call and the competing parameter lists match up.
- In decreasing order of goodness of match:

1. An exact match
2. A promotion
3. A standard type conversion, e.g. int to short
4. A conversion constructor or user-defined type conversion

Exact Matches

- An exact match is where the parameter and argument types match exactly.
- Example: argument is a double and parameter is a double.
- Note that, for the purposes of overload resolution a pointer to an array of type x exactly matches a pointer of type x.
    - This is because arrays are always passed by reference, meaning that you actually pass a pointer to the first element of the array.

Exact Matches For Arrays

```
void f(int y[ ]){}      // call this f1
void f(int* z){}        // call this f2
....
int x[ ] = {1, 2, 3, 4};
f(x);           // Both f1 and f2 are exact
matches, so the call is ambiguous.

void sf(const char s[]){}
void sf(const char*){}
....
sf("abc");    // Same problem; both sf
functions are exact matches.
```

Type Promotion

- The following are described as "promotions":
  - A char, unsigned char or short can be promoted to an int.
    - For example, **`void`** `foo(`**`int`**`);` can be a match for `foo('a');`
  - A float can be promoted to a double.
  - A bool can be promoted to an int
    - false counts as 0, true as 1.

Standard Conversions

- All the following are described as "standard conversions":
    - conversions between integral types, apart from the ones counted as promotions. Remember that bool and char are integral types as well as int, short and long.
    - conversions between floating types: double, float and long double, except for float to double which counts as a promotion.
    - conversions between floating and integral types
    - conversions of integral, floating and pointer types to bool (zero or NULL is false, anything else is true)
    - conversion of an integer zero to the NULL pointer.

Standard Conversions

- All of the standard conversions are treated as equivalent for scoring purposes.
  - A seemingly minor standard conversion, such as int to long, does not count as any "better" than a more drastic one such as double to bool.

Conversion Constructors

- Constructors that provides an implicit type conversions.
- This kind of conversion can only work when the constructor can be called with just one argument.
- Generally this means that the constructor will have just one parameter, but it could have more if all but the first of the parameters (or, indeed, all of them) had default values.

explicit

- It can happen that you have a constructor that can be called with just one argument, and which will therefore behave as a conversion constructor, but you don't want it to behave in this way.
- But implicit type-conversions are a common source of programming error, so you might decide to disallow that sort of conversion.
- We can do that simply by inserting the keyword **explicit** before the constructor prototype in the class definition

User-Defined Conversions

- Conversion member-functions allow you to specify how you want objects to respond if they are asked to behave as if they were objects of some other type.
- In following example, a Foo is being treated as if it were an int:

```
class Foo {
public:
    operator int() { return 0; }
}
void boo(int n) { }
Foo foo;
boo(foo);
```

Choosing a Winner

- A candidate function is only as strong as its weakest match;
  - a candidate requiring three promotions, for example, beats a candidate with two exact matches and a standard conversion.
- Candidates whose weakest matches are equivalently weak are compared on their next-weakest, and so on - a candidate with a standard conversion, a promotion and an exact match beats a candidate with a standard conversion and two promotions.