HIGH-LEVEL PROGRAMMING 2

Function Overloading

by Prasanna Ghali

Functions: Parameters and Arguments

this variable is called *formal parameter* or just *parameter*

```
int myabs(int number) {
  return number < 0 ? -number : number;
}
```

client calls function `myabs` using function call operator ( )

```
int num = 10;
num = myabs(-num)
```

this expression is called *function argument*

Function Overloading: Idea (1/2)

- You'll often find that you need multiple functions that essentially do the same thing but with parameters of different types
- C requires such functions to have distinct names
- Idea behind function overloading is to avoid having to invent and remember unique function names and instead use single name

Function Overloading: Idea (2/2)

- In *same scope*, there can be multiple functions with same name provided their sets of parameters differ
- More formally, function overloads must differ in their *signatures* which consists of:
  - Function name
  - Number of parameters, and
  - Types of parameters [in their respective order]
- Notice we don't care about return type

Are Function Signatures Different?


```
struct S { /* ... */ };

void boo(S);
void boo(S const);
```

```
struct S { /* ... */ };

void coo(S const &rs);
void coo(S const&);
```

```
struct X { /* ... */ };
typedef X TypeX;

X doo(X const&);
X doo(TypeX  const&);
```

```
struct X { /* ... */ };

int doo(X const&);
X   doo(X  const&);
```

Function Overloading: Declaration and Definitions


```
// declarations ...
namespace misc_stuff {

  void print(int);
  void print(double);
  void print(char const*);

}
```

```
// definitions ...
namespace misc_stuff {

void print(int x) {
  std::cout << x;
}

void print(double x) {
  std::cout << x;
}

void print(char const *x) {
  std::cout << x;
}

}
```

Function Overloading: Usage


```
// use of overloaded functions ...
int main() {
  misc_stuff::print(123);
  std::cout << "\n";

  misc_stuff::print(123.456);
  std::cout << "\n";

  misc_stuff::print("123.456789");
  std::cout << "\n";
}
```

Function Overloading: Advantage

- Eliminates need for programmers to invent – and remember – names that exist only to help figure out which function to call for specific argument types

Name Mangling

- But how do compilers deal with multiple functions having same name?
- Compilers mangle same function names to create unique functions with decorated names
- Linux program *nm* gives decorated names for your functions

Overload Resolution

- Compiler's job is to pick right function according to language rules
- Process used by compiler to choose function to call based on a set of arguments is called *overload resolution*
- Unfortunately, in order to cope with complicated cases, language rules are quite complicated!!!
- We'll worry less about language rules and more about few basic guidelines!!!

Overload Resolution: Simplified Version [1 /3]

- Finding right version to call from set of overloaded functions is done by looking for *best match* between type of arguments and corresponding parameters

Overload Resolution: Simplified Version [2/3]

- Series of following criteria is tried in order:
  - Exact match: no or only trivial conversions [array name to pointer, function name to pointer to function, `T` to `T const`]
  - Match using *promotions*: integral promotions [`bool` to `int`, `char` to `int`, `short` to `int`, and their unsigned counterparts] and `float` to `double`
  - Match using *standard conversions*: `int` to `double`, `double` to `int`, `double` to `long double`, `int` to `unsigned int`, …
  - Match using user-defined conversions: for example `char const*` to `std::string`

Overload Resolution: Simplified Version [3/3]

- If exact match is found, the compiler will take it
- Otherwise:
  - If multiple matches are found at highest level where a match is found, call is rejected as ambiguous
  - If no match is found, call will fail

Overload Resolution: Examples (1/2)


```
void foo(int);          // 1
void foo(int&);         // 2
void foo(int const&); // 3

int i{5}, &ri{i};
int const& cri{i};

foo(5);   // ???
foo(i);   // ???
foo(ri);  // ???
foo(cri); // ???
```

Overload Resolution: Example (2/2)


```
void foo(int);          // 1
void foo(int&);         // 2
void foo(int const&); // 3

int i{5}, &ri{i};
int const& cri{i};

foo(5);   // ???
foo(i);   // ???
foo(ri);  // ???
foo(cri); // ???
```

Overload Resolution: Built-In Types

- Function overloading for small number of built-in types leads to surprising results!!!
- See *cube.hpp*, *cube.cpp*, *cube-driver.cpp* for more details
- *Well designed system must not include function overloads with parameters that are closely related*
- *If you wish to overload functions for built-in types, provide overloads for all built-in types.*
- This is what standard library does!!!

Overload Resolution: Reference and Value Parameters

- Mixing overloads of reference and value parameters almost always fails!!!
- See *refval.hpp*, *refval.cpp*, *refval-driver.cpp* for more details
- *Well designed system must not include function overloads with value and reference parameters*
- *When one overload has reference-qualified parameter, corresponding parameter of other overloads should be reference-qualified as well*

Overload Resolution: Don't Mix Overloading & Default Parameters

- We get surprising results when we mix overloading and default parameters

```
void bar(int a);
void bar(int a, int b = 1);

bar(5, 6); // ok
bar(1);    // ambiguous
```

Overloading and Scope

- Functions declared in different non-namespace scopes do not overload!!!

```
void foo(int);

void boo() {
  void foo(double);
  foo(1); // ???
}
```

Overloading Advice (1/7)

- We can implement following overloads

```
void foo(X);          // 1) in parameters: inexpensive
void foo(X&);         // 2) in-out parameters
void foo(X const&); // 3) in parameters: expensive
```

- We learnt all three overloads should not be implemented
- Which functions should be declared in what circumstances?

Overloading Advice (2/7)

- If we implement only:

```
void foo(X);          // 1) in parameters: inexpensive
void foo(X&);         // 2) in-out parameters
void foo(X const&); // 3) in parameters: expensive
```

- `foo` can be called on everything: *rvalues*, `const` *rvalues*, *lvalues*, and `const` *lvalues*,
- Pass by value is expensive; therefore should be used only for small inexpensive values such as built-in types and small structures!!!

Overloading Advice (3/7)

- If we implement only:

```
void foo(X);          // 1) in parameters: inexpensive
void foo(X&);         // 2) in-out parameters
void foo(X const&); // 3) in parameters: expensive
```

- `foo` can be called for *lvalues* but not for `const` *lvalues*, *rvalues*, and `const` *rvalues*
- Pass by reference is inexpensive but only reason to implement this function is to modify values in caller!!!

Overloading Advice (4/7)

- If we implement only:

```
void foo(X);          // 1) in parameters: inexpensive
void foo(X&);         // 2) in-out parameters
void foo(X const&); // 3) in parameters: expensive
```

- `foo` can be called for *lvalues* , `const` *lvalues*, *rvalues*, and `const` *rvalues*
  - However, not possible to distinguish between *lvalues* and *rvalues*
- Use when you only want to pass expensive values with read-only access to functions

Overloading Advice (5/7)

- If we implement these two:

```
void foo(X);          // 1) in parameters: inexpensive
void foo(X&);         // 2) in-out parameters
void foo(X const&); // 3) in parameters: expensive
```

- Don't do this!!!
- Overload resolution fails for *lvalues*

Overloading Advice (6/7)

- If we implement these two:

```
void foo(X);          // 1) in parameters: inexpensive
void foo(X&);         // 2) in-out parameters
void foo(X const&); // 3) in parameters: expensive
```

- Ideal set of overloads!!! Can be called for *lvalues*, `const` *lvalues*, *rvalues*, and `const` *rvalues*
  - You can distinguish between *lvalues* AND `const` *lvalues*, *rvalues*, and `const` *rvalues*

Overloading Advice (7/7)

- If we implement these two:

```
void foo(X);          // 1) in parameters: inexpensive
void foo(X&);         // 2) in-out parameters
void foo(X const&); // 3) in parameters: expensive
```

- Unnecessary!!!
- Maintaining two overloads with similar behavior not worth trouble!!!

Overload Resolution: Multiple Parameters (1/3)

- We first find best match for each argument
- If one function is at least as good a match as all other functions for every argument and is a better match than all other functions for one argument, that function is chosen; otherwise call is ambiguous

Overload Resolution: Multiple Parameters (2/3)


```
void f(int, std::string const&, double); // 1
void f(int, char const*, int);           // 2

f(1, "hello", 1);                  // ???
f(1, std::string("hello"), 1.0); // ???
f(1, "hello", 1.0);                // ???
```

Overload Resolution: Multiple Parameters (3/3)

- In last call:
  - `"hello"` matches `char const*` without a conversion and `std::string const&` only with conversion
  - On other hand, `1.0` matches `double` without conversion but `int` only with conversion
  - So neither function is better match than the other

```
void f(int, std::string const&, double); // 1
void f(int, char const*, int);           // 2

f(1, "hello", 1);                  // Ok: call 2
f(1, std::string("hello"), 1.0); // Ok: call 1
f(1, "hello", 1.0);                // Error: ambiguous
```

Summary

- In C++, functions may be overloaded
- The same name may be used to define different functions as long as their signatures differ
- Compiler will automatically figure out which function to call based on arguments in call
- Process of selecting right function from set of overloaded functions is called *overload resolution* or *function matching*