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*

Idea: Function Overloading

- 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 that we don't care about return type

Idea: Function Overloading


```
// 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;
}

}
```

Idea: Function Overloading


```
// 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";
}
```

Idea: Function Overloading

- 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 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

Overload Resolution: Simplified Version [3/3]

- If exact match is found, the compiler will take it.
- Otherwise:
  - If two 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: 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
```

Overload Resolution: Multiple Parameters

- 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

- 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
```