C++ Overload Inspector
A tool for analyzing and profiling overloaded function calls

By Botond Horvath
Eötvös Lóránd University
Budapest
Overloading in C++

- Same named functions with different types
- Usually constructors or operators
- Often used in metaprogramming
- Picking the wrong overload can cause
  - Bugs
  - Bad performance (copy instead of move)
- The “Best matching” function will be called
The rules for overloading ([over.match])

- Rules are in priority
- F1 is better than F2 if:
  - **F1 has a better conversion then F2 and F2 hasn’t better conversion**
    Note if both have a better conversion in different parameters we have ambiguity regardless of the rules later
- Inside a User-defined conversion: has better conversion after the function
- Inside a conversion: better reference binding (lvalue, rvalue)
- **F2 is a template F1 is not**
Overloading with templates in C++ ([over.match])

- **F1 is a more specialized template**
- Non templated F1, F2, same types (with same class) and same ObjectParam and F1 is more constrained template
- F1 is constructor for derived class (F2’s constructor for base)
- F2 is rewritten and F1 is not (see C++20 => → <)
- F1 and F2 is rewritten, F2 is reversed (C++20 A=>B → B<A)
- More rules for deduction guides
- Also more rules for surrogates
A confusing example

```cpp
#include <iostream>
struct Any{
    Any(){
        std::cout<"Made\n";
    }
    Any(const Any& o){
        std::cout<"Copied\n";
    }
    Any(Any&& o){
        std::cout<"Moved\n";
    }
    template <class T>
    Any(T&& a) {
        std::cout<"New any from T\n";
    }
};
int main(){
    Any a(1);
    const Any b = std::move(a);
    Any c(b);
    Any d(c);
}
```

<table>
<thead>
<tr>
<th>Param type</th>
<th>Any c(b)</th>
<th>Any d(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&amp;&amp;</td>
<td>const Any&amp;</td>
<td>Any&amp;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conversion vs copy constructor</th>
<th>Any → const Any&amp; vs Any → const Any&amp;</th>
<th>Any → Any&amp; vs Any → const Any&amp;</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Reason of selection</th>
<th>templated</th>
<th>Better conversion</th>
</tr>
</thead>
</table>

New any from T
Moved
Copied
New any from T
A 2nd example

#include <iostream>

template<class T>
void foo(T a){ // #1
    std::cout << "T":;
}

template<> void foo(int* a){ // #2
    std::cout << "int*":;
}

template<class T>
void foo(T* a){ // #3
    std::cout << "T*:;"
}

int main(){
    int iptr;
    foo(iptr);
}

#2 is a specialization of #1
Candidates for overloading are #1 and #3

#3 is more specialized than #1 (T* vs T)
If we put #3 to the top
    #2 will be a specialization of #3

Even the order of definitions can impact the code

Will print “T*”

Will print “int*”
Overloading in metaprogramming

- Often used in metaprogramming
- Metaprogramming “runtime” = compile time
- Metaprogramming “values” = types and functions
- We have good profilers and debuggers for runtime
- We don’t really have good tools for metaprogramming
So overloading is

- Sometimes hard to get right
- Can cause bugs
- Used a lot in metaprogramming
- No good tools to rely on for debugging/profiling
My Overload-Inspector tool

- Fork of clang
  - Can build on infrastructure
  - Knows all the features
  - Same result as clang (including bugs)
  - Not portable to other compilers :(  
  - Can only work with information present at overload resolution
- Inserted hooks to get info with minimal overhead if disabled
My Overload-Inspector tool

- Prints info about the overloads
- Prints optionally YAML for other tools
- Can measure overloading time for each function (in progress)
  - At least clang’s overloading time
- Get parameters from flags
- Long term goal: make into clang or be an easily addable patch
An other tool related to this

- My tool is for overloads is what is templight to templates
- See more on templight: c++Now 2013
What we print (possibly)

- Lists all candidates in categories (Best / Viable / Non viable / Ambiguous)
  - What the templates are deduced to
- Gives reason why something is non viable.
- Prints all the conversions needed.
- Prints the reason why the best candidate is better than the all others.
  - Or if ambiguous why
- Verbose by default can be turned down
- Filter for function name/ location in file
- The time overloading took
Example output

tmp2.cpp:7:5: remark: Overload resulted with OR_Success With types [double[temporary]]
  7 |   foo(1.0);
  
    Success

With types [double[temporary]]

tmp2.cpp:4:1: note: Best candidate: foo (int)
  4 | void foo(int){
  
    1

    2 void foo(int){

    3 }

    4 void foo(float &){

    5 }

    6 int main(){

    7   foo(1.0);

    8 }

    9

    1

    2 void foo(float &){

    3 }

    4 void foo(int){

    5 }

    6 int main(){

    7   foo(1.0);

    8 }
Example output for ambiguity

- tmp2.cpp:7:5: remark: Overload resulted with OR_Ambiguous With types [double[temporary]]
  7 | foo(1.0);
  tmp2.cpp:4:1: note: Ambiguous candidate: foo (int)
  4 | void foo(int){
  tmp2.cpp:7:9: note: Conversions:
     StandardConversion conversion:
     double[temporary] -> int
     Floating-integral conversion
  7 | foo(1.0);
  tmp2.cpp:2:1: note: Ambiguous candidate: foo (float)
  2 | void foo(float){
  tmp2.cpp:7:9: note: Conversions:
     StandardConversion conversion:
     double[temporary] -> float
     Floating conversion
  tmp2.cpp:7:1: note: Comparing candidates resulted in The first is not better (reason: inconclusive)
My tool's output for the first example:

conf.cpp:23:6: remark: Overload resulted with OR_Success With types [Any]
   23 |   Any d(c);
   24 |   
Best candidate: Any::Any (Any &)
   13 |     template<class T>
   14 |     Any(T&& a){
Conversions:
      StandardConversion conversion:
        Any -> Any & = T &&
Viable candidate: Any::Any (const Any &)
   7 |     Any(const Any& o){
Conversions:
      StandardConversion conversion:
        Any -> const Any &
Non viable candidate: Any::Any ()
Comparing candidates resulted in The first is better (reason: betterConversion)
  Conversions:[ ( Any -> Any & = T &&)     >     ( Any -> const Any &)]
My tool’s output for the 2nd example:

```
x.cpp:16:5: warning: Explicit specialization ignored
   19 |     foo(iptr);
The ignored specialization:
   9 |     template<>
  10|     void foo(int* a) { // #2
General declaration:
   4 |     void foo(T a){ // #1
x.cpp:16:5: remark: Overload resulted with OR_Success With types [int *]
  20 |     foo(iptr);
Best candidate: foo (int *)
   Template params: [T = int]
  14 |     void foo(T* a){ // #3
Viable candidate: foo (int *)
   Template params: [T = int *]
  4 |     void foo(T a){ // #1
note: 1 explicit template specializations found
Comparing candidates resulted in The first is better (reason: moreSpecialized)
```
Relevancy: C++23 deducing this

- Has no new overloading rules
- Just “regular” function templates
- They can match on the object parameter perfectly.
- Can lead to unexpected results in overloading if mixed with “regular” member functions.
```cpp
struct A{
    template <typename Self>
    void bar(this Self&& s, int i) {} //#1
    void bar(float f) const {} //#2
};

int main(){
    A a;
    const A ca;
    a.bar(1.0f); // #3
    a.bar(1); // #4
    ca.bar(1.0f); // #5
    ca.bar(1); // #6
}
```

<table>
<thead>
<tr>
<th>#3</th>
<th>Ambiguous</th>
<th>(A -&gt; A) &gt; (A -&gt; const A)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(float -&gt; int) &lt; (float -&gt; float)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#4</th>
<th>#1</th>
<th>(A -&gt; A) &gt; (A -&gt; const A)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(int -&gt; int) &gt; (int -&gt; float)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#5</th>
<th>#2</th>
<th>(const A -&gt; const A) = (const A -&gt; const A)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(float -&gt; int) &lt; (float -&gt; float)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#6</th>
<th>#1</th>
<th>(const A -&gt; const A) = (const A -&gt; const A)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(int -&gt; int) &gt; (int -&gt; float)</td>
</tr>
</tbody>
</table>
struct A{
  template <typename Self>
  void bar(this Self&& s, int i) {} //#1
  void bar(float f) {} //#2
};

int main()
{
  A a;
  const A ca;
  a.bar(1.0f); // #3
  a.bar(1); // #4
  ca.bar(1.0f); // #5
  ca.bar(1); // #6
}

Better overload | Conversions compared
---|---
#3 | #2 | (A -> A) = (A -> A) (float -> int) < (float -> float)
#4 | #1 | (A -> A) = (A -> A) (int -> int) > (int -> float)
#5 | #1 | (const A -> const A) >! (const A -> A) (float -> int) ? (float -> float)
#6 | #1 | (const A -> const A) >! (const A -> A) (int -> int) ? (int -> float)
Settings

- We can filter by line or function name
- We can show/hide details like conversions, non viable candidates…
- We can switch between profiling and explanation (possibly both)
- More explanation in github readme
Timing/profiling

- Each candidate has to be looked at each function call
- With N overloads and M calls → resolution N*M times evaluated
- Can be a lot (operator<<)

- Goal: measure the time each function takes
- Can be useful for library writers and big projects.
- Can be useful for metaprogramming
- In progress
- The order of the decelerations can impact the overloading time
Profiling

- Same filters apply
- Measures overload-time for each function
- Makes a summary (function name; call count; time it took)
- Sorted by overall time
- Example:
  
  ```
  <<:  count:3  overload time:  1.013567e-01s  from this in children:  9.815693e-03s  
  foo:  count:2  overload time:  1.670122e-03s  from this in children:  7.796288e-05s  
  ```
- So far only used in artificial examples, and in debug build.
- Right now prints walltime
How to use profiling

- For metaprogrammers
  - Just run the tool filtering for the interesting functions
  - See which overloads take too long

- For library-writers:
  - Create a test cpp file with calls for the overloaded functions (called by realistic types)
  - Run the tool for the test file
  - Look for relatively high overload times (Compared to the call count)
Conclusion

- Can help debugging
- Can do profiling on the time of overloading (in progress)
  - helpful for libraries and metaprograms
  - Ideas for good metrics will be appreciated
- 2 clang bugs found
- Code can be downloaded from: github
Thank you for the attention.