Overloading OVERLOADING Overloading Overloading

Overloading in C++: How Does It *Really* Work?

Overloading Overloading Overloading OVERLOADING

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >

A little about me

- B.A. (math's); M.S., Ph.D. (computer science).
- Professional programmer for over 50 years, programming in C++ since 1982.
- Experienced in industry, academia, consulting, and research:
  - Founded a Computer Science Dept.; served as Professor and Dept. Head; taught and mentored at all levels.
  - Managed and mentored the programming staff for a reseller.
  - Lectured internationally as a software consultant and commercial trainer.
  - Retired from the Scientific Computing Division at Fermilab, specializing in C++ programming and in-house consulting.
- Not dead — still doing training & consulting. (Email me!)

 3

Emeritus participant in C++ standardization

- Written ~175 papers for WG21, proposing such now-standard C++ library features as gcd/lcm, cbegin/cend, common_type, and void_t, as well as all of headers <random> and <ratio>.
- Influenced such core language features as *alias templates*, *contextual conversions*, and *variable templates*; recently worked on *requires-expressions*, operator<=>, and more!
- Conceived and served as Project Editor for *Int'l Standard on Mathematical Special Functions in C++* (ISO/IEC 29124), now incorporated into C++17's <cmath>.
- Be forewarned: Based on my training and experience, I hold some rather strong opinions about computer software and programming methodology — these opinions are not shared by all programmers, but they should be! ☺

 4

Not this kind of overloading ☺

 5

Nor this kind of overloading ☺

 6

Nope, still wrong kind of overloading ☺

 7

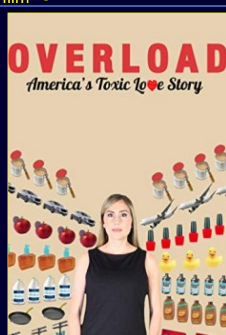
No, not there yet ☺

 8

Nor this kind of overloading, either ☺

 10

No, not this 2018 film ☺



 11

Getting a bit closer … ☺

 12

An observation ☺

Isn't it remarkable how often the word "OVERLOAD" has itself been overloaded?

 13

In today's talk we'll explore …

- Origins of Overloading
- Principles of Overloading
- Selecting among Overloaded Functions
- Scenarios and Subtleties of Overloading
- An Advanced Case Study (or Two)

 14

Origins of Overloading

Let's consider the expression x + y

- What does the + operator mean in this context?
  - Without more information, it's simply unclear how to obtain the intended result.
  - In mathematics, we would evaluate the expression according to the kinds of entities denoted by x and by y.
  - *E.g.*, summing whole numbers vs. fractions vs. matrices.
- *I.e.*, the same notation can imply different techniques:
  - The method to be used depends on solely the operands.
  - So operators are …

From *The Design and Evolution of C++* [1994, reformatted]

- "Operators are used to provide notational convenience. …
- "When variables can be of different types, we must decide whether to allow mixed-mode arithmetic or to require explicit conversion to a common type [instead]. …
- "By choosing the former — as [other languages] have — C++ entered a difficult area without perfect solutions. …
- "This … results in a fundamentally difficult problem.
- "The desire for flexibility and freedom of expression clashes with wishes for safety, predictability, & simplicity."

The net outcome

- So C++ supports operator overloading, just as many other programming languages (*e.g.*, 1957's FORTRAN) do.
- But C++ supports, too, the more general feature that we term function overloading.
- By treating operators as "functions with funny names," we obtain a coherent set of rules for all overloading:
  - Whether we spell a function plus to call it as plus(x, y), …
  - Or spell the function operator + and call it as x + y.
- So this talk will focus mostly on named functions.

Principles of Overloading

C++ has overloaded declarations

- A C++ name is described as overloaded if:
  - ✓ The name is declared at least twice in a single scope, …
  - ✓ Each such declaration introduces either a function or a primary (unspecialized) function template, and …
  - ✓ The declarations are mutually distinguishable.
- Examples of indistinguishable declarations:
  - × A redeclaration is not distinguishable from its initial decl.
  - × Fctn decl's are not distinguishable if their sole difference is in their { return types, noexcept specifications }.
  - × Member fctn decl's are not distinguishable if they differ by only the presence/absence of { static, ref-qual }.

What's left?

- Like-named function decl's are distinguishable when:
  - ✓ One declares a primary function template and the other declares an ordinary function, or …
  - ✓ They have different numbers of parameters, or …
  - ✓ Any corresponding param's have distinguishable types.
- Examples of indistinguishable fctn parameters:
  - × Differ only in name or in default value (not part of type).
  - × One type is an alias for the other's (alias is not a new type).
  - × One type is the decayed form of the other's type (*e.g.*, E[⋯] ⇒ E∗ or R (⋯) ⇒ R (∗) (⋯) or T const ⇒ T).

 21

Overloaded?

- `char calc1( char c );`
  `namespace ns { long calc1( unsigned char u ); }`
  - ✗ No; declarations are in (inhabit) different scopes.
- `char calc2( char c );`
  `char calc2( unsigned char c );`
  - ✓ Yes; these parameters' types are distinguishable. (While char might natively be an unsigned type, it's not an alias.)
- `char calc3( char c );`
  `char calc3( char d = 'a' );`
  - ✗ No; param names and defaults do not affect the param types, so the 2nd declaration merely redeclares the 1st.

 22

Overloaded?

- `char calc4( char );`
  `template< class T = char > char calc4( T );`
  - ✓ Yes; a fctn and a fctn template can overload each other.
- `double * calc5( double p (double ) );`
  `double * calc5( double (*q) (double ) );`
  - ✗ No; the 2nd redeclares the 1st. A param of fctn type is indistinguishable from its decayed (ptr-to-fctn) type.
- `float calc6( float );`
  `float calc6( float, double = 3.14 );`
  - ✓ Yes; a 1-param fctn and a 2-param fctn can overload. (However, a single-arg call is likely ambiguous here.)

 23

Overloaded?

- `struct B { int calc7( int ); }`
  `struct D : B { char calc7( float ); };`
  - ✗ No; these declarations inhabit different scopes.
  - ✗ Further, D::calc7 hides B::calc7; never are both visible.
- `struct B { int calc8( int ); }`
  `struct D : B { using B::calc8; char calc8( float ); };`
  - ✓ Yes; D::calc8 is overloaded.
  - ✓ The using declaration brings B::calc8 into D's scope, where there is also a distinguishable calc8. However, …
  - ✗ If the two calc8's were indistinguishable, B::calc8 would (despite the using) be hidden by D::calc8.

 24

Selecting among Overloaded Functions

Preamble to clause "Overloading" [reformatted]

- "When a function name is used in a call, which function declaration is being referenced [is] determined by comparing
  - "the types of the arguments at the point of use with
  - "the types of the parameters in the declarations that are visible [at that point of use].
- "This function selection process is [termed] *overload resolution*.…"

 26

Overload resolution: compiler's initial steps

1) Initiated from a named function use. (Not via fctn ptrs!)
2) Prepare a list of candidate fctn decl's via appropriate name lookup(s) (unqualified, qualified, arg-dep, …):
   - Found a fctn template? Synthesize a fctn decl from it, but silently discard that decl if it's ill-formed (SFINAE).
   - C'tor? Consider deduction guides, too (since C++17).
3) For each candidate c, determine c's viability, namely:
   - ✓ Do the call's arg's match c's param's in number (after accounting for ellipsis param. and default arguments, if any)?
   - ✓ Can each arg (directly, or via promotion/conversion/decay, or via reference binding) initialize c's corresponding param?
   - ✓ Are c's associated constraints satisfied (since C++20)?

 27

Overload resolution: compiler's remaining steps

4) Seek the best of the viable candidate fctn decl's.
   - (Details on the next page.)
5) Success iff:
   - "there is exactly one viable [candidate] that is a better function than all other viable [candidates]" and …
   - That candidate is accessible in the context of the use.
6) If overload resolution succeeds:
   - The fctn def'n (instantiated if necessary) corresponding to the chosen decl will be applied in the use context.
   - Else the program is ill-formed.

 28

Sample criteria to decide the better of two viable candidates

- How many conversion steps are needed to init a param with its corresponding arg?
  - ✓ Prefer the candidate needing fewer conversion steps.
- A param of rvalue type vs. a param of lvalue type:
  - ✓ Prefer the candidate with the rvalue parameter type.
- A param of derived vs. a param of its base class type:
  - ✓ Prefer the candidate with the derived parameter type.
- An ordinary fctn vs. one synthesized from a template:
  - ✓ Prefer the candidate that's an ordinary function.

 29

In brief …

- "[T]he candidate function whose parameters match the arguments most closely is the one that is called."
- But overload resolution may be needed wherever a function name may appear, not only in function calls:
  - As an initializer in an obj or ref declaration, or …
  - On the right-hand side of an assignment , or …
  - As an argument to a function, to a user-defined operator, or to a static or explicit cast , or …
  - As the operand of a return statement, or …
  - As a non-type template argument.

 30

Scenarios and Subtleties of Overloading

Scenarios ①

- What if a candidate function (or function template specialization) is defined as deleted?
  - *E.g.*, `double g( double ) = delete;` *// a deleted definition*
- The overload resolution algorithm considers only declarations, not any definition, …
  - So how (or even whether) a function is defined is not relevant to the O.R. algorithm.
  - But if overload resolution selects a deleted definition as its best viable candidate, the program is ill-formed (analogous to a function that's declared but not defined).

 32

Scenarios ②

- What if a candidate member function (or member function template specialization) is declared private?
  - *E.g.*, class C { double g(double); } *// implicitly private g*
- Overload resolution considers only the declaration, not its accessibility:
  - So where it's declared is not relevant to the algorithm.
  - But the program is ill-formed if overload resolution yields an inaccessible declaration as its best viable candidate.
  - (*I.e.*, even private members affect a class' interface!)

 33

Scenarios ③ (adapted from CWG2169)

- Consider:
  - struct wrapped_long { long k; };
    struct wrapped_short { short k; };
  - void g(wrapped_long) { ⋯ }
    void g(wrapped_short) { ⋯ }
  - ⋯ g({ 1'000'000 }) ⋯ *// call is ambiguous!*

This rule is being reconsidered for C++23!

- The wrapped_short parameter can't be initialized from the arg { 1'000'000 } due to the narrowing conversion:
  - But overload resolution inspects only an arg's type, not its value, so the above fact does not affect the outcome.
  - (Not all compilers today correctly implement this rule.)

 34

Scenarios ④

- Valid code?
  - int S;
    struct S { ⋯ };

```
class S { ⋯ };
S S;          // ok, but IMO perverse
S t;          // no! S is not a type
class S u;    // ok, here S is a type
```

  - Yes, S is multiply-declared in a single scope, but S is not overloaded because no functions are declared.
  - Nonetheless, it is valid C++ (because it was valid C code).
- This (mis?)feature is termed an elaborated type:
  - The variable's name hides the type's name.
  - But the type name becomes visible when it's preceded by struct/class/union (*i.e.*, as an elaborated type specifier).
  - (Please avoid such code whenever possible.)

 35

Scenarios ⑤

- Overload resolution is sometimes performed twice!
- Example:
  - ```
    struct copy_only {             // has no move c'tor, so not movable
      copy_only( );                // default c'tor
      copy_only( copy_only & );    // copy c'tor, so no implicit move
    };
    ```
  - ```
    copy_only go( ) {
      copy_only x;
      return x;      // even if elided: prefer to move, fall back to copy
    }
    ```

This rule, too, is being reconsidered for C++23!

  - Treat x as an rvalue during overload resolution; if that fails, treat x as an lvalue and repeat overload resolution.

 36

Scenarios ⑥

- Does an explicit specialization overload its primary function template?
  - No, neither the primary template nor any specialization is ever a candidate for overload resolution.
  - Overloading considers function declarations only: a template does not declare any function, although function declarations can be synthesized from a primary template.
  - When function templates are involved, declarations (not definitions!) synthesized from the primary template become candidates considered by overload resolution.

 37

Scenarios ⑦

- Suppose overload resolution selects a synthesized (from a function template) declaration: now what?
- Then we need that declaration's corresponding specialization (definition):
  - Either the programmer has explicitly provided such a corresponding explicit specialization, or …
  - Else the compiler must instantiate such a specialization from the definition of the primary template.

 38

Scenarios ⑧

- What if overload resolution yields a tie between candidate declarations that were each synthesized, but from distinct function templates?
- Such ties are resolved via a partial ordering algorithm:
  - *E.g.*: a more specialized candidate is better than one that is less specialized.
  - *E.g.*: a constrained candidate (C++20) is better than one that is unconstrained or less constrained. (Applies iff all corresponding parameters have identical type.)
- (The algorithm is termed partial because not all ties can be broken.)

Scenarios ⑨

- Suppose some declarations originate elsewhere:
  - `double calc9( double );`
  - `extern long calc9( long );`
  - `using yonder :: calc9;`
- Is this a valid set of overloaded calc9 declarations?
  - Yes, provided that …
  - Each calc9 declaration from namespace yonder is of a function (or fctn template) that is distinguishable from the first two declarations.

Scenarios ⑩

- Compare use of a default argument vs. overloading:

1)
```
… calc10( string s = "Hello"s ) { … }
```
2)
```
… calc10( string s ) { … }                    // 1st of two overloads
… calc10( ) { return calc10( "Hello"s ); }    // forwards
```

- Recall that a default argument, like every argument, is supplied at each call site:
  - *I.e.*, default arguments are always inlined.
  - Thus, if you have many calls to calc10( ), …
  - You potentially have many copies of the default arg …
  - But just one (out-of-line) copy in the overload set.

From *The Design & Evolution of C++* [reformatted]

- "Given general function overloading, …
  - default [function] arguments are logically redundant …
  - and at best a minor notational convenience.
- "However, C with Classes …
  - had default argument lists for years …
  - before general overloading became available in C++."
- Another reason to prefer overloading:

✗
```
template< class T > void g( T = 0 ) { }
                       // won't infer T as int when defaulted!
```
✓
```
template< class T = int > void g( T = 0 ) { }
```


Default arguments interact with virtual [A. Tomazos]

```
struct B {
  virtual void g( int x = 42 ) = 0;
};

struct D : B {
  void g( int x = 43 ) override { std::cout << x; }
};

int main( ) {
  D d;        d.g( );     // no vtable involved; displays 43

  B & b = d;  b.g( );     // vtable dispatch; displays 42 (!)
}
```


Scenario adapted from recent posted question

- Which function g will be called in each case?

```
void g( int           p1 ) { … }
void g( int const &&  p2 ) { … }
```
```
int k = 0;
g( k );              // #1
g( std::move(k) );   // #2
```

Recall that C++ expressions never have a reference type!

  - Hint: consider the type of each call's argument.
- At #1, argument k has type int, which can equally well initialize param's p1 and p2, ∴ call #1 is ambiguous.
- At #2, the argument expression std::move(k) also has type int (not int &&), ∴ #2 is identically ambiguous.

Scenario excerpted from [over.ics.list] Example 1

- struct A { int x, y; };
  struct B { int y, x; };
- void g(A a);
  void g(B b);
- g({ .x = 1, .y = 2 }); // aggregate initialization; uses
  // a (C++20) designated-initializer-list
- Ambiguous: "Aggregate initialization does not require that the members are declared in designation order."
  - But: "If, after overload resolution, the order does not match for the selected overload, the initialization of the parameter will be ill-formed."
  - (Not all compilers yet correctly implement this rule.)

45

Scenario adapted from a recent compiler bug report

- Start with:
  - template< class > void foo( ) { } // #1
    void use1( ) { foo<int>( ); }
  - Instantiates foo<int> from primary template #1, right?
- Later in the same translation unit, continue with:
  - template< class > requires true void foo( ) { } // #2
    void use2( ) { foo<int>( ); } // which foo<int>?
  - O.R. selects foo<int> from template #2 — its constraint breaks the tie with unconstrained #1.
  - But "you can't change the result of O.R. for a given call", so this "program is ill-formed, no diagnostic required."

46

[over.match.best.general]/4 and Example 8 (excerpted)

- "If … multiple declarations were found [in] different scopes and specify a default argument that made the function viable, the program is ill-formed."
- *I.e.*, default arguments can affect viability, if used:
  - namespace A { extern "C" void g(int = 5); }
    namespace B { extern "C" void g(int = 5); }
  - using A::g, B::g; // bring both g declarations into our scope
  - void use( ) {
      g(3); // OK: no default argument was used for viability
      g( ); // error: default argument found twice
    }

47

An Advanced Case Study

Can we overload callables that aren't functions?

- *I.e.*, given several function objects:
  - Each of whose type has the form*:
    struct … { … **R operator ( ) (…) { … }** … };
  - Can we provide them under some common name …
  - So as to allow overload resolution to apply to their call?
- auto go = overload{ *your, function, object, instances, …* };
- ⋮
  go(…); // after overload resolution of go's operators ( ),
  // will call the corresponding function object instance

*Note that all lambdas' closure types have this form, as do all function objects of class type (*e.g.*, std::function)!

49

Yes, via parameter packs, inheritance, and using

- We'll ① inherit from each of the function objects, and ② bring each of their call operators into our scope:
  - template< class... Fs > // treat the fctn objs' types as a pack
    struct overload
    : Fs … // ① inherit from each fctn obj's type
    { // no c'tor needed — rely on aggregate initialization
      using Fs::operator( ) … ; // ② bring call op's into this scope
    };
- Example (being careful to ensure distinguishability):
  - auto go = overload{ [ ] (int k) { return k + 1; }
    , [ ] (string s) { return s + s; }
    };

50

Can we overload, yet control the order of consideration?

- I.e., given the same kinds of function obj's as before:
  - Can we provide them under some common name …
  - So that, of the function objects that we provide, we will call the first-listed one that's viable?
- Yes; let's design a class template first_viable:
  - We'll distinguish its first parameter, f of type F, from the rest of its parameters, a pack fs of types Fs….
  - (It's a rather Lisp-like approach, treating our list of function object parameters as having a head and a tail.)
  - If f is viable when supplied with arg's, we'll call f(args…).
  - Else, we'll call first_viable<Fs…>{fs…}(args…).

The big picture

- First, an empty primary template to handle cases when nothing is viable, so we call nothing:
  - `template< class… > class first_viable { };`
- Then a specialization that will check viability in order:

```
template< class F, class… Fs >
class first_viable<F, Fs…> {
private:
  using  Rest = first_viable<Fs…>;
  F      first;      // head of the list
  Rest   rest;       // tail of the list
public:
  ⋮
```


Fleshing out details

- We need a c'tor:

```
constexpr first_viable( F f, Fs... fs )
    : first( move(f) ), rest( move(fs)... )  { }
```

- Let Case1 denote the call first(forward<Args>(args)…):

```
template< class... Args >
constexpr auto operator ( ) ( Args && ... args ) const
    noexcept( noexcept( Case1 ) ) –> decltype( Case1 )
    requires requires { Case1; }   // satisfied iff a viable call
{ return  Case1; }
```

- Let Case2 denote the call rest(forward<Args>(args)…):
  - As above, changing all Case1 –> Case2, except …
  - `requires ( not requires { Case1; } )`

A few last touches

- A deduction guide can be useful:

```
template< class... Fs >
first_viable( Fs... )  –>  first_viable <Fs…>;
```

- Could consolidate the operator ( ) overloads:
  - `{ if constexpr( requires { Case1; } )  return Case1;`
    `else return Case2; }`
  - But the return type and the noexcept(···) clauses become messier (although certain type traits can help with these).
- Finally, consider using std::invoke instead of bare calls:
  - Understands calling members (both functions and data), and reference_wrappers as well as function objects.

Overloading in C++:

How Does It *Really* Work?

FIN

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >