Overload Resolution

Ansel Sermersheim & Barbara Geller
Amsterdam C++ Group
March 2019

Introduction

- Prologue
- Definition of Function Overloading
- Determining which Overload to call
- How Overload Resolution Works
- Standard Conversion Sequences
- Examples
- Bonus Round

Who is CopperSpice

- Maintainers and Co-Founders of the following projects
  - CopperSpice
    - set of cross platform C++ libraries *(linux, os x, windows)*
  - DoxyPress
    - documentation program for multiple languages and outputs
  - CsString
    - support for UTF-8 and UTF-16, extensible to other encodings
  - CsSignal
    - thread aware signal / slot library
  - libGuarded
    - library which manages access to data shared between threads

Who is CopperSpice

- Credentials
  - products leverage modern C++ idioms
  - all of our libraries and applications are open source
  - source code hosted on github
  - experience spans multiple platforms and languages
  - active video series teaching C++ with over 35 videos

  - copperspice is an expanding team
    - C++ developers worldwide
    - technical documentation reviewers
    - test and quality assurance contributors
    - slack channel for team communication

Definitions

- Function Overloading
  - function overloading pertains to
    - free functions, constructors, and methods
    - developers commonly refer to all of these as “functions”
  - order of declaration for overloaded functions is not meaningful
  - two or more functions are overloads of each other when
    - they have the exact same function name
    - are visible from the same scope
    - have a different set of parameters

Definitions

- When is a Function Overload a Compile Error
  - two functions which differ only by the return type
    - will not compile
    - since using the return value is optional, the compiler treats this as defining the same function twice
  - two functions which differ only in their default arguments
    - will not compile
    - default values do not make the function signature different
  - two methods with the same name and parameters, where one method is declared “static”
    - will not compile, ambiguous

Example

- Example 1

```
void doThing1(int)              // overload 1
{ }


void doThing1(int, double)      // overload 2
{ }


int main() {
  doThing1(42);                 // calls overload 1
  doThing1(42, 3.14);           // calls overload 2
}
```

Overload Resolution

- Determining which overload to call
  - computed by the compiler
  - for simple cases this process is intuitive and usually results in calling the expected overload
  - however, it can get complicated very fast . . .

  - pointer / reference data types do not resolve as you initially expect
  - template functions can deduce arguments in unexpected ways
  - data type conversions can be messy

  - overload resolution is the name of the process in C++ for selecting which overload will be called

Overload Resolution

- From the standard
  - C++17 defines overload resolution in clause 16 (28 pages)

  - overload resolution depends on
    - name lookup, argument dependent lookup (40 pages)
    - fundamental types (10 pages)
    - value categories (30 pages)
    - standard conversions (15 pages)
    - user defined conversions (25 pages)
    - template argument deduction (80 pages)
    - SFINAE (35 pages)
    - special member functions (30 pages)

Overload Resolution

- What is Overload Resolution
  - a process which is used to select the most appropriate overload
  - the compiler must decide which overload to call
    - done at compile time
    - only considers (passed) argument data types and how they match the (received) parameter data types, never the actual values
    - if the compiler can not choose one specific function, the function call is considered ambiguous

  - template functions or methods
    - they participate in the overload resolution process
    - if two overloads are deemed equal, a non-template function will always be preferred over a template function

Overload Resolution

- Before Overload Resolution Starts
  - compiler must first run a procedure called name lookup
  - name lookup is the process of finding every function declaration which is visible from the current scope
    - name lookup may require argument dependent lookup
    - template functions may require template argument deduction

Overload Resolution

- Details of Overload Resolution
  - first step is for every overload to be put in a list of candidates
    - template argument deduction is done just prior to creating the candidate list or while it is being created
  - second step is to remove all invalid candidates
    - according to the C++ standard functions which are invalid are regarded as not viable
  - what makes a particular candidate invalid
    - passed argument count does not match the parameter list
      - passing fewer arguments than the function parameter list declared may still be a valid overload if default arguments exist
    - passed arguments are not a possible match to the received parameters, even when considering implicit conversions

Overload Resolution

- Type Conversions
  - also known as type casting and type coercion
  - a way of changing a value from one data type to another
    - int to a float
    - string literal to a pointer
    - enum to an int
    - timestamp to a long
    - int to a string
    - char * to a void *
    - std::any to std::vector (or anything)

```
doThing(38)                    // passed argument, int

void doThing(float data)       // received parameter, implicit type conversion
```

Overload Resolution

- Type Conversions
  - type conversions are either implicit or explicit
  - example of an implicit conversion

```
char foo[] = "ABC";
int bar    = foo[0];        // bar will equal 65
```

  - an explicit conversion would be a static_cast, dynamic_cast, reinterpret_cast, c style cast
  - another type of explicit conversion is a functional cast

```
if (std::string(“root”) == current_directory) {
  // do something
}
```

Overload Resolution

- Standard Conversion Sequences - Order of Ranking (1-3)
  - exact match
    - no conversion is required
  - lvalue transformations
    - lvalue to rvalue conversion
    - array to pointer conversion
    - function to pointer conversion
  - qualification adjustments
    - qualification conversion *(adding const or volatile)*
    - function pointer conversion *(new in C++17)*

Overload Resolution

- Standard Conversion Sequences - Order of Ranking (4-5)
  - numeric promotions
    - integral promotion
    - floating-point promotion
  - conversions
    - integral conversion
    - floating-point conversion
    - floating–integral conversion
    - pointer conversion
    - pointer-to-member conversion
    - boolean conversion

Overload Resolution

- LValue to RValue Conversion
  - based on value categories
  - according to the C++ standard
    - a glvalue of any non-function, non-array type T can be implicitly converted to a prvalue of the same type
    - if T is a non-class type, this conversion also removes cv-qualifiers

  - parameter which expects an rvalue and is passed an lvalue

Overload Resolution

- Qualification Conversion
  - according to the C++ standard
    - prvalue of type “pointer to T” can be converted to a prvalue of type “pointer to more cv-qualified T”
  - constness and volatility can be added to a pointer
  - this is an implicit “extra” first parameter to a member function
    - allows a const or volatile qualified member function to be a candidate for a call which passed an unqualified object

```
std::vector<int> data;
data.size();              // data is not const, size() method is const qualified
```

Overload Resolution

- Example 2

```
void doThing2(char value)       // overload 1
{ }


void doThing2(long value)       // overload 2
{ }


int main() {
  doThing2(42);                 // which overload is called?
}
```

Overload Resolution

- Example 2

```
void doThing2(char value)       // overload 1
{ }


void doThing2(long value)       // overload 2
{ }


int main() {
  doThing2(42);                 // ambiguous ( compile error )
}
```

Overload Resolution

- Argument Conversions
  - standard conversion - integral promotion
    - unsigned short promotable to unsigned int or int
      - depends on your platform
    - short promotable to int
    - char promotable to int or unsigned int
      - depends on your platform
    - bool promotable to int (0 or 1)
  - standard conversion - floating point promotion
    - float to double

Overload Resolution

- Example 3
  - integral conversion
    - from an int to a long

```
void count(long value)          // single candidate
{ }


int main() {
  count(42);
}
```

Overload Resolution

- Example 4
  - compile error message - “no matching function for call to”
  - error message will list the possible candidates

```
void doThing4(char x1)          // single candidate
{ }


int main() {
  doThing4(‘x’, nullptr);
}
```

Overload Resolution

- User Defined Conversions
  - standard conversions are part of C++
    - part of the language, used to convert between known types
  - implicit conversions in the STL are considered user defined
    - classes in the STL like std::string have implicit constructors
    - const_iterators are implicitly convertible to iterators
  - C++ has no knowledge about conversions between user defined data types which are defined in your classes or application
  - all user defined conversions have a lower ranking below the standard conversions

Overload Resolution

- Example 5

```
void doThing5(char value)          // candidate A
{ }


template <typename T>
void doThing5(T value)             // candidate B
{ }


int main() {
  doThing5(42);                    // which overload is called?
}
```

Overload Resolution

- Example 5

```
void doThing5(char value)          // candidate A
{ }


template <typename T>
void doThing5(T value)             // candidate B
{ }


int main() {
  doThing5(42);                    // candidate B wins
}
```

Overload Resolution

- Best Overload Selection Process
  - if exactly one function is better than all other functions in the candidate list, it is called the “best viable function” and wins the overload resolution process

  - create the candidate list
  - remove the invalid functions
  - rank the candidates
    - process of ranking the remaining candidates is how the compiler finds the single best match
    - best candidate match may be the least bad match
  - tie breakers

Overload Resolution

- Full List for Ranking Candidates
  - exact match
    - no conversion, lvalue to rvalue, cv qualification
  - numeric promotion
    - integral, floating point
  - conversion
    - integral, floating point, pointer, boolean
  - user defined conversion
    - convert a const char * to an std::string
  - ellipsis conversion
    - c style varargs function call

Overload Resolution

- Selecting a Candidate
  - tie breakers are used throughout overload resolution to decipher which candidate might be a better match
  - when a template and a non-template candidate are tied for first place the non-template function is selected
  - an implicit conversion which requires fewer “steps” is a better match than a candidate which takes more “steps”
  - if there is no best match or there is an unresolvable tie, a compiler error is generated

Overload Resolution

- When the Candidate List has no best Match
  - how to resolve an ambiguous function call
    - change your overload set
    - mark a constructor explicit to prevent an implicit conversion
    - template functions can be eliminated through SFINAE
      - a template function which can not be instantiated will not be placed in the candidate set
    - convert arguments before the call, using an explicit conversion
      - static_cast<> an argument being passed
      - explicitly construct an object
      - use std::string("some text") rather than pass a string literal

Overload Resolution

- When the Best Match is Not What You Wanted
  - overload resolution can be hard to debug since there is no clean way to ask the compiler why it chose a particular overload
  - overload resolution can be more complex than template argument deduction
  - given a template function with a template parameter T
    - where one of the parameters is const T&
    - this function will be an exact match for nearly every call
    - which may not be what you intended

Overload Resolution

- Example 6
  - CsString library has a constructor allowing a const char * or char * to be implicitly converted to a CsString
  - const char * is a C string which is not the same as a string literal
  - #define used in the implementation to inject a static_assert if this constructor should be disabled, thus disallowing a C string

```
#define CS_STRING_ALLOW_UNSAFE

template <typename T, typename =
       typename std::enable_if<std::is_same<T, const char *>::value ||
                               std::is_same<T, char *>::value>::type>
CsBasicString(const T &str, const A &a = A());
```

Overload Resolution

- Example 6
  - type trait test added to ensure the compiler does not evaluate the static_assert until the template is fully instantiated

```
template <typename E, typename A>
template <typename T, typename>
CsBasicString<E, A>::CsBasicString(const T &str, const A &a)
  : m_string(1, 0, a)
{

#ifndef CS_STRING_ALLOW_UNSAFE
   static_assert(! std::is_same<E, E>::value, "Unsafe operation not ...");
#endif

// constructor implementation source removed for simplicity
```

Overload Resolution

- Example 6
  - since template argument deduction runs before overload resolution
    - our templated constructor will be placed in the list of candidates when a C string is passed
  - since template instantiation occurs after overload resolution has selected the best match
    - the static_assert is only evaluated if this template is chosen

Overload Resolution

- Example 7

```
// A
void doThing_A(double, int, int) { }        // overload 1
void doThing_A(int, double, double) { }     // overload 2

int main() {
  doThing_A(4, 5, 6);                       // which overload is called?
}


// B
void doThing_B(int, int, double) { }        // overload 3
void doThing_B(int, double, double) { }     // overload 4

int main() {
  doThing_B(4, 5, 6);                       // which overload is called?
}
```

Overload Resolution

- Example 7

```
// A
void doThing_A(double, int, int) { }        // overload 1
void doThing_A(int, double, double) { }     // overload 2

int main() {
  doThing_A(4, 5, 6);                       // ambiguous ( compile error )
}


// B
void doThing_B(int, int, double) { }        // overload 3
void doThing_B(int, double, double) { }     // overload 4

int main() {
  doThing_B(4, 5, 6);                       // overload 3 wins
}
```

Overload Resolution

- Example 8

```
// D
void doThing_D(int &) { }                    // overload 1
void doThing_D(int)   { }                    // overload 2

int main() {
  int x = 42;
  doThing_D(x);                              // which overload is called?
}

// E
void doThing_E(int &) { }                    // overload 3
void doThing_E(int)   { }                    // overload 4

int main() {
  doThing_E(42);                             // which overload is called?
}
```

Overload Resolution

- Example 8

```
// D
void doThing_D(int &) { }                    // overload 1
void doThing_D(int)   { }                    // overload 2

int main() {
  int x = 42;
  doThing_D(x);                              // ambiguous ( compile error )
}

// E
void doThing_E(int &) { }                    // overload 3
void doThing_E(int)   { }                    // overload 4

int main() {
  doThing_E(42);                             // overload 4 wins
}
```

Overload Resolution

- Example 9

```
// F
void doThing_F(int &)  { }                    // overload 1
void doThing_F(int &&) { }                    // overload 2

int main() {
  int x = 42;
  doThing_F(x);                               // which overload is called?
}

// G
void doThing_G(int &)  { }                    // overload 3
void doThing_G(int &&) { }                    // overload 4

int main() {
  doThing_G(42);                              // which overload is called?
}
```

Overload Resolution

- Example 9

```
// F
void doThing_F(int &)  { }                    // overload 1
void doThing_F(int &&) { }                    // overload 2

int main() {
  int x = 42;
  doThing_F(x);                               // overload 1 wins
}

// G
void doThing_G(int &)  { }                    // overload 3
void doThing_G(int &&) { }                    // overload 4

int main() {
  doThing_G(42);                              // overload 4 wins
}
```

Overload Resolution

- Example 10 - Bonus Round

```
void doThing_10(int &) { }                  // overload 1, lvalue ref to int
void doThing_10(...)   { }                  // overload 2, c style varargs

struct MyStruct
{
  int m_data : 5;                           // bitfield, 5 bits stored in an int
};


int main() {
  MyStruct object;
  doThing_10(object.m_data);                // which overload is called?
}
```

Overload Resolution

- Example 10 - Bonus Round

```
void doThing_10(int &) { }                  // overload 1, lvalue ref to int
void doThing_10(...)   { }                  // overload 2, c style varargs

struct MyStruct
{
  int m_data : 5;                           // bitfield, 5 bits stored in an int
};


int main() {
  MyStruct object;
  doThing_10(object.m_data);                // overload 1 wins
}
```

Overload Resolution

- Example 10 - Bonus Round

```
void doThing_10(int &) { }                    // overload 1, lvalue ref to int
void doThing_10(...)   { }                    // overload 2, c style varargs

struct MyStruct
{
  int m_data : 5;                             // bitfield, 5 bits stored in an int
};


int main() {
  MyStruct object;
  doThing_10(object.m_data);                  // overload 1 wins
}
```

➔ Hang on, compile error “non const reference can not bind to bit field”
➔ adding an overload which takes a “const int &” does not change the result

Presentations

- Why CopperSpice
- Why DoxyPress
- Compile Time Counter
- Modern C++ Data Types (references)
- Modern C++ Data Types (value categories)
- Modern C++ Data Types (move semantics)
- CsString library (unicode)
- CsString library (library design)
- Multithreading in C++
- Multithreading using libGuarded
- Signals and Slots
- Build Systems
- Templates in the Real World
- Copyright Copyleft
- What's in a Container
- Modern C++ Threads
- C++ Undefined Behavior
- Regular Expressions
- Using DoxyPress
- Type Traits
- C++ Tapas (typedef, forward declarations)
- Lambdas in C++
- C++ Tapas (typename, virtual, pure virtual)
- Overload Resolution
- Futures & Promises
- Special Member Functions
- C++ in Review
- Thread Safety
- Constexpr Static Const
- What do you do when Your Codebase is Old Enough to Vote
- Sequencing
- Linkage
- Inheritance
- Evolution of Graphics Technology
- GPU, Pipeline, and the Vector Graphics API (pending release 14 March)

Please subscribe to our YouTube Channel
https://www.youtube.com/copperspice

Libraries

- CopperSpice
  - libraries for developing GUI applications
- CsSignal Library
  - standalone thread aware signal / slot library
- CsString Library
  - standalone unicode aware string library
- libGuarded
  - standalone multithreading library for shared data

Applications

- KitchenSink
  - one program which contains 30 demos
  - links with almost every CopperSpice library

- Diamond
  - programmers editor which uses the CopperSpice libraries

- DoxyPress & DoxyPressApp
  - application for generating source code and API documentation

Where to find CopperSpice

- www.copperspice.com

- ansel@copperspice.com
- barbara@copperspice.com

- source, binaries, documentation files
  - download.copperspice.com

- source code repository
  - github.com/copperspice

- discussion
  - forum.copperspice.com