Overload Resolution
Back To Basics

Ansel Sermersheim & Barbara Geller
CppCon - October 2021

Introduction

- Prologue
- Overload vs Override
- Why is Overload Resolution Required
- Definitions
- What is Overload Resolution
- When will declaring an Overload fail
- Overview of the Process
- When to Use Overloads vs Using a Template
- Process to find the Best Overload
- When the Best Match is Not What You Wanted
- Examples

Prologue

- Co-Founders of the following projects
    - CopperSpice
        - cross platform C++ GUI libraries
    - DoxyPress
        - documentation generator for C++ and other languages
    - CsString
        - support for UTF-8 and UTF-16, extensible to other encodings
    - CsSignal
        - thread aware signal / slot library
    - CsLibGuarded
        - library for managing access to data shared between threads

Prologue

- Credentials
  - every library and application is open source
  - projects are developed using cutting edge C++ technology
  - all source code hosted on github
  - prebuilt binaries available on our download site
  - documentation is generated by DoxyPress

  - youtube channel with videos focused mostly on C++
  - speakers at multiple conferences
    - CppCon, CppNow, emBO++, MeetingC++, code::dive
  - numerous presentations for C++ user groups
    - United States, Germany, Netherlands, England

Overload Resolution

- Overload vs Override
  - definition of polymorphism
    - having many forms
    - when there are two or more possibilities
  - compile time polymorphism
    - function, method, or constructor overloading
    - invoking the correct function is based on the data type
  - run time polymorphism
    - method overriding
    - invoking the correct method based on the object

Overload Resolution

- Overload vs Override
  - function, method, or constructor overloading
    - multiple declarations which have different signatures
    - main topic for this presentation

Overload Resolution

- Overload vs Override
  - method overriding
    - used with inheritance, method in the child class has the same name as a method in the parent class
    - example: class Fruit and class Apple where both declare a method called canYouEatThePeel()
    - *unrelated to the subject of overload resolution*

Overload Resolution

- Overload vs Override
    - operator overloading
        - invokes a different operation for the given operator
        - example: operator+()
            - might add two numbers or concatenate two strings
        - implemented using the overload resolution process

Overload Resolution

- Why is Overload Resolution Required
    - an overload occurs when two or more functions have the same name and are all visible where the call is being made
    - this avoids long names like doFunctionStr(), doFunctionInt(), doFunctionBool(), doFunctionDouble(), doFunctionPairIntString(), or doFunctionFunctionPtr(), simply to support multiple data types

```
doFunction(“mountain”);
doFunction(17);
```

Overload Resolution

- Declaring Overloads
  - applies to free functions, methods, and constructors
    - term function will be used to refer to all of these
  - function declarations are overloads of each other when . . .
    - they have the exact same name
    - visible from the same scope
    - have a different set of parameter types
  - order of declarations is not meaningful

Overload Resolution

- 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

- What is Overload Resolution
  - process of selecting the most appropriate overload

  - compiler decides at compile time which overload to call
    - only considers (passed) argument types and how they match the (received) parameter types, never the actual values
    - if the compiler can not choose one specific overload, the function call is considered ambiguous

  - template functions or methods
    - 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

- When will declaring an Overload fail
  - two functions which differ only by the return type
    - does 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
    - does not compile
    - default values do not make the function signature different
  - two methods with the same signature, one is marked as “static”
    - does not compile

Overload Resolution

- Overview of the Process
  - computed by the compiler
  - for many cases this process results in calling the expected overload
  - however, it can get complicated very fast . . .

  - data type conversions can be messy
  - pointer / reference data types may not resolve as expected
  - template functions can deduce arguments in unexpected ways
  - if the “wrong” overload is selected it can be difficult to debug

Overload Resolution

- When to Use Overloads vs Using a Template
  - should you write an overload set or a single template?

  - prefer overloaded functions when the implementation changes for different data types
    - example: constructors for *std::string*
    - (const char *), (std::string &&), (size_type, char), etc

  - templates are the correct choice when the body of the function does the same thing for all data types
    - example: *std::sort(data.begin(), data.end())*
    - data can be any type such as std::vector, std::string, etc

Overload Resolution

- From the C++ Standard
  - C++17 defines overload resolution in clause 16 (32 pages)
    - name lookup, argument dependent lookup (44 pages)
    - fundamental types (33 pages)
    - value categories (31 pages)
    - declarations (45 pages)
    - standard conversions (15 pages)
    - user defined conversions (25 pages)
    - template argument deduction (80 pages)
    - SFINAE (35 pages)
    - special member functions (30 pages)
  - *C++20 defines overload resolution in clause 12 (35 pages)*

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
  - full list of visible function declarations is called the overload set

Overload Resolution

- Details of Overload Resolution
  - first step, entire overload set is put in a list of candidates
  - second step, remove all invalid candidates
    - according to the C++ standard invalid overloads are referred to as "not viable"

Overload Resolution

- What Makes a Candidate Not Viable or Invalid (1 of 2)
    - number of passed arguments does not match the declaration
        - passing too many arguments is always considered invalid
        - passing fewer arguments is invalid unless default arguments exist in the function declaration

```
doThing(38);

void doThing();                          // candidate A
void doThing(int, bool = true);          // candidate B
```

- What Makes a Candidate Not Viable or Invalid (2 of 2)
  - data type of passed arguments can not be converted to match the declaration
    - even when considering implicit conversions

```
doThing(38);

void doThing();                        // candidate A
void doThing(int);                     // candidate B
void doThing(std::string);             // candidate C
```

Overload Resolution

- Process to find the Best Overload
  - create the candidate list
  - remove the invalid overloads

  - rank the remaining 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

  - if exactly one function in the candidate list ranks higher than all others, it wins the overload resolution process

  - if there is a tie for the highest ranking, then tie breakers are used

Overload Resolution

- Type Conversions
  - data type casting to change a value from one type to another
    - int to float
    - string literal to pointer
    - enum to int
    - timestamp to long
    - int to string
    - char * to void *
    - type X to type Y (depending on your code base)

```
doThing(38)                    // passing an int

void doThing(float data)       // receiving a float using an implicit conversion
```

Overload Resolution

- Type Conversions
  - example of an implicit conversion

```
char str[] = "ABC";
int data   = str[0];        // data will equal 65
```

  - explicit conversion
    - static_cast, dynamic_cast, reinterpret_cast, or c style cast

  - another type of explicit conversion called an functional cast

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

Overload Resolution

- Standard Conversion Categories
  - exact match (1)
    - no conversion is required
  - lvalue transformations (2)
    - lvalue to rvalue conversion *(based on value categories)*
    - array to pointer conversion
    - function to pointer conversion
  - qualification adjustments (3)
    - qualification conversion *(adding const or volatile)*
    - function pointer conversion *(new in C++17)*

Overload Resolution

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

Overload Resolution

- Qualification Adjustments (category 3)
  - qualification conversion -- const or volatile can be added to any pointer data type
  - passed argument
    - pointer to an std::string
  - received parameter
    - (A) ptr to a const std::string vs (B) ptr to std::string

```
std::string * myString = new std::string(“text”);
int value = lookUp(myString);


int lookUp(const std::string * key);            // candidate A
int lookUp(std::string * key);                  // candidate B
```

Overload Resolution

- Example 2

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

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

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

Overload Resolution

- Example 2

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


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


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

Overload Resolution

- Numeric Promotions (category 4)
  - integral promotion
    - short to int
    - unsigned short to unsigned int or int
    - bool to int (0 or 1)
    - char to int or unsigned int
    - a few more however it <u>must</u> be defined in the standard
  - floating point promotion
    - float to double

Overload Resolution

- Example 3
  - integral conversion (category 5)
    - integral data types are defined by the C++standard
    - examples: bool, char, short, int, long
    - if the standard defines converting between integral type A and integral type B is a promotion, it is not a conversion

```
void count(long value);            // int to long conversion, valid candidate

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

Overload Resolution

- Full List for Conversions In Ranking Order
  - no conversion (1-3)
    - exact match, lvalue transformations, qualification adjustments
  - numeric promotion (4)
    - integral promotions, floating point promotions
  - numeric conversion (5)
    - 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

- User Defined Conversion
  - user defined conversions have a lower ranking than any standard conversion

  - definition: implicit conversion to or from any class
    - class declaration may be located in the standard library, a third party library, or your application

```
void showMsg(std::string value) { }                // valid candidate

int main() {
  const char *msg = “Text”;
  showMsg(msg);
}
```

Overload Resolution

- Selecting a Candidate
  - tie breakers are used as the last step in overload resolution to decipher which candidate is 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

- New Tie Breaker
  - C++20 introduced Concepts which are used with templates to add a constraint on the T, thus limiting the set of allowed types

  - having a constraint on a template does not change the meaning of the template in regards to overload resolution

  - if a passed argument satisfies the concept for more than one overloaded template, the more constrained template is chosen
    - this rule is used only as a tie breaker

Overload Resolution

- Example 4

```
void doThing4(char value)          // candidate A
{ }

template <typename T>
void doThing4(T value)             // candidate B
{ }

int main() {
  doThing4(42);                    // which overload is called?
}
```

Overload Resolution

- Example 4

```
void doThing4(char value)           // candidate A, int to char conversion
{ }


template <typename T>
void doThing4(T value)              // candidate B, exact match
{ }


int main() {
  doThing4(42);                     // candidate B wins
}
```

Overload Resolution

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

Overload Resolution

- Example 5
  - compile error message - “no matching function for call”
  - error message will list the possible candidates, even though there are no viable candidates

```
void doThing5(char value)                    // not a viable candidate
{ }


int main() {
  doThing5(‘x’, nullptr);
}
```

Overload Resolution

- When the Best Match is Not What You Wanted
    - overload resolution can be complicated to debug since there is no clean way to ask the compiler why it chose a particular overload
    - it would be helpful if compilers provided a verbose mode
    - by intentionally adding an ambiguous overload to the candidate list, the resulting error message may help in deciphering why
    - try changing the data type of some passed argument

Overload Resolution

- Example 6

```
// 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 6

```
// 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 7

```
// 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 7

```
// 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 8

```
// 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 8

```
// 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
}
```

- Example 9 - Bonus Round

```
void doThing_9(int &) { }             // overload 1, lvalue ref to int
void doThing_9(...)   { }             // overload 2, c style varargs

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


int main() {
  MyStruct obj;
  doThing_9(obj.m_data);              // which overload is called?
}
```

Overload Resolution

- Example 9 - Bonus Round

```
void doThing_9(int &) { }                       // overload 1, lvalue ref to int
void doThing_9(...)   { }                       // overload 2, c style varargs

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


int main() {
  MyStruct obj;
  doThing_9(obj.m_data);                        // overload 1 wins
}
```

Overload Resolution

- Example 9 - Bonus Round

```
void doThing_9(int &) { }                    // overload 1, lvalue ref to int
void doThing_9(...)   { }                    // overload 2, c style varargs

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


int main() {
  MyStruct obj
  doThing_9(obj.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

Overload Resolution

- Back to the Basics . . .
  - understanding overload resolution requires knowing more of the C++ standard than almost any other feature
  - learn the difference between promotions and conversions
  - try to avoid mixing overloaded functions with a template of the same name
  - debugging an ambiguous overload error can be frustrating and time consuming

Things every C++ programmer should know . . .

- ❑ Modern C++ Data Types (data types, references)
- ❑ Modern C++ Data Types (value categories)
- ❑ Modern C++ Data Types (move semantics, perfect forwarding)

- ❑ Learn Programming, then Learn How to Be a Programmer (CppOnSea Keynote)
  https://www.youtube.com/watch?v=jIa17JCaNvo

- ❑ What is the C++ Standard Library
- ❑ CsString library - Intro to Unicode
- ❑ char8_t

- ❑ Multithreading in C++
- ❑ Modern C++ Threads
- ❑ C++ Memory Model

Presentations - www.youtube.com/copperspice

- Why CopperSpice, Why DoxyPress
- Compile Time Counter
- Multithreading using CsLibGuarded
- Signals and Slots
- Templates in the Real World
- Copyright Copyleft
- What's in a Container
- C++ Undefined Behavior
- Regular Expressions
- Type Traits
- C++ Tapas (typedef, forward declarations)
- C++ Tapas (typename, virtual, pure virtual)
- Lambdas in C++
- Overload Resolution
- Futures & Promises
- Thread Safety
- Constexpr Static Const
- When Your Codebase is Old Enough to Vote
- Sequencing
- Linkage
- Inheritance
- Evolution of Graphics Technology
- GPU, Pipeline, and the Vector Graphics API
- Declarations and Type Conversions
- Lambdas in Action
- Any Optional
- Variant
- std::visit
- CsPaint Library
- Moving to C++17
- Attributes
- Copy Elision
- Time Complexity
- Qualifiers
- Concepts in C++20
- Atomics
- Memory Model to Mutexes
- Mutexes + Lock = CsLibGuarded
- Variable Templates
- Paradigms and Polymorphism

Libraries

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

Libraries

- CsCrypto
  - C++ interface to the Botan and OpenSSL libraries
- CsPaint Library
  - standalone C++ library for rendering graphics on the GPU

Applications

- KitchenSink
  - contains over 30 demos, uses 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
- twitter: @copperspice_cpp

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

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

- source code repository
  - github.com/copperspice

- discussion
  - forum.copperspice.com