C++
Overload resolution

Robert Piszczatowski

Tls technologie (tls.pl)

DCC Labs (dcclabs.com)

Warsaw C++ Users Group, 2015

Overloading in programming languages

Overloadling is not supported in some programming languages:

Overloading in programming languages

Overloadling is not supported in some programming languages:

- c - overloading supported for builtin operators only

Overloading in programming languages

Overloadling is not supported in some programming languages:

- c - overloading supported for builtin operators only
- ocaml - overloading supported for relational operators only

Overloading in programming languages

Overloadling is not supported in some programming languages:

- c - overloading supported for builtin operators only
- ocaml - overloading supported for relational operators only
- haskell - no overloading at all (typeclasses as alternative)

Why use overloading?

```

template <typename T>
void myFunction(T &t1, T &t2) {
  using std::swap;
  swap(t1, t2);
}
```

Overloadable declarations

```
struct A {
  void f(int);
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int);
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*);
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int);
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int); /* error: static and non-static
      overloads of f(int) */
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int); /* error: static and non-static
      overloads of f(int) */
  static int f(double);
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int); /* error: static and non-static
      overloads of f(int) */
  static int f(double); /* ok: static but different
      argument types */
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int); /* error: static and non-static
      overloads of f(int) */
  static int f(double); /* ok: static but different
      argument types */
  int f(char) &;
```

Overloadable declarations

```
struct A {
 void f(int);
 void f(const int); /* error: same as above */
 void f(int*);
 void f(const int*); /* ok: not same as above */
 void f(int[8]);  /* error: same as f(int*) (decay) */
 static void f(int); /* error: static and non-static
     overloads of f(int) */
 static int f(double); /* ok: static but different
     argument types */
 int f(char) &;
 int f(char) &&;
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int); /* error: static and non-static
      overloads of f(int) */
  static int f(double); /* ok: static but different
      argument types */
  int f(char) &;
  int f(char) &&;  /* ok: difference in ref qualifiers
      of implicit object */
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int); /* error: static and non-static
      overloads of f(int) */
  static int f(double); /* ok: static but different
      argument types */
  int f(char) &;
  int f(char) &&;  /* ok: difference in ref qualifiers
      of implicit object */
  void f(const char) &;
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int); /* error: static and non-static
      overloads of f(int) */
  static int f(double); /* ok: static but different
      argument types */
  int f(char) &;
  int f(char) &&;  /* ok: difference in ref qualifiers
      of implicit object */
  void f(const char) &; /* error: difference in return
      type only */
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int); /* error: static and non-static
      overloads of f(int) */
  static int f(double); /* ok: static but different
      argument types */
  int f(char) &;
  int f(char) &&;  /* ok: difference in ref qualifiers
      of implicit object */
  void f(const char) &; /* error: difference in return
      type only */
  void f(int) &;
```

Overloadable declarations

```
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int); /* error: static and non-static
      overloads of f(int) */
  static int f(double); /* ok: static but different
      argument types */
  int f(char) &;
  int f(char) &&;  /* ok: difference in ref qualifiers
      of implicit object */
  void f(const char) &; /* error: difference in return
      type only */
  void f(int) &;   /* error: ref qualifiers for f(int)
      is all or none feature */
```

Overloadable declarations

```cpp
struct A {
  void f(int);
  void f(const int); /* error: same as above */
  void f(int*);
  void f(const int*); /* ok: not same as above */
  void f(int[8]);  /* error: same as f(int*) (decay) */
  static void f(int); /* error: static and non-static overloads of f(int) */
  static int f(double); /* ok: static but different argument types */
  int f(char) &;
  int f(char) &&;  /* ok: difference in ref qualifiers of implicit object */
  void f(const char) &; /* error: difference in return type only */
  void f(int) &;   /* error: ref qualifiers for f(int) is all or none feature */
};
```

Overloading

If you really want to distinguish int and const int…

Overloading

If you really want to distinguish int and const int…

…you can use templates:

```
template <typename T> void f(T);
```

Overloading

If you really want to distinguish int and const int…

…you can use templates:

```
template <typename T> void f(T);

f(5);
```

Overloading

If you really want to distinguish int and const int...

...you can use templates:

```
template <typename T> void f(T);

f(5);          // T = int

```

Overloading

If you really want to distinguish int and const int…

…you can use templates:

```
template <typename T> void f(T);

f(5);             // T = int
f((const int)5);
```

Overloading

If you really want to distinguish int and const int...

...you can use templates:

```
template <typename T> void f(T);

f(5);            // T = int
f((const int)5); // T = int

```

Overloading

If you really want to distinguish int and const int...

...you can use templates:

```
template <typename T> void f(T);

f(5);              // T = int
f((const int)5); // T = int
f<const int>(5);
```

Overloading

If you really want to distinguish int and const int...

...you can use templates:

```
template <typename T> void f(T);

f(5);               // T = int
f((const int)5);  // T = int
f<const int>(5);  // T = const int

```

Overloading

If you really want to distinguish int and const int…

…you can use templates:

```
template <typename T> void f(T);

f(5);               // T = int
f((const int)5); // T = int
f<const int>(5); // T = const int
f("ala ma kota");
```

Overloading

If you really want to distinguish int and const int…

…you can use templates:

```
template <typename T> void f(T);

f(5);              // T = int
f((const int)5); // T = int
f<const int>(5); // T = const int
f("ala ma kota"); // T = const char *

```

Overloading

If you really want to distinguish int and const int. . .

. . . you can use templates:

```
template <typename T> void f(T);

f(5);               // T = int
f((const int)5); // T = int
f<const int>(5); // T = const int
f("ala ma kota"); // T = const char *
f<const char[22]>("ala ma kota");
```

Overloading

If you really want to distinguish int and const int…

…you can use templates:

```
template <typename T> void f(T);

f(5);             // T = int
f((const int)5); // T = int
f<const int>(5); // T = const int
f("ala ma kota"); // T = const char *
f<const char[22]>("ala ma kota"); // T = const char [22]
```

Overloading and object initialization

```
struct A;
struct B {
  B();
  B(const A&);
};

struct A {
  operator B();
};

int main() {
  A a;
  (B)a;

  B b = a;
}
```

Overloading and object initialization

```
struct A;
struct B {
  B();
  B(const A&);
};

struct A {
  operator B();
};

int main() {
  A a;
  (B)a; /* call B::B(const A&), A::operator B() not
      considered */
  B b = a; /* call cast operator of A */
}
```

How overloading can be joined with other language features.

```
template <typename T> void f(T);
template <typename T> void f(T*);
template <> void f(int *);

int *p{};
f(p);
```

How overloading can be joined with other language features.

```
template <typename T> void f(T);
template <typename T> void f(T*);
template <> void f(int *);

int *p{};
f(p);    /* calls f(int*) */
```

How overloading can be joined with other language features.

```
template <typename T> void f(T);
template <typename T> void f(T*);
template <> void f(int *);

int *p{};
f(p);    /* calls f(int*) */
```

```
template <typename T> void f(T);
template <> void f(int *);
template <typename T> void f(T*);

int *p{};
f(p);
```

How overloading can be joined with other language features.

```
template <typename T> void f(T);
template <typename T> void f(T*);
template <> void f(int *);

int *p{};
f(p);    /* calls f(int*) */
```

```
template <typename T> void f(T);
template <> void f(int *);
template <typename T> void f(T*);

int *p{};
f(p);    /* calls f(T*) */
```

Overloading

Overloading and templates specializations (2)

C++ standard (n3797), §14.7.3.7

„When writing a specialization, be careful about its location; or to make it compile will be such a trial as to kindle its self-immolation”

Overloading
Overload resolution contexts

- invocation of a function named in the function call syntax;

Overloading
Overload resolution contexts

- invocation of a function named in the function call syntax;
- invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax;

- invocation of a function named in the function call syntax;
- invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax;
- invocation of the operator referenced in an expression;

- invocation of a function named in the function call syntax;
- invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax;
- invocation of the operator referenced in an expression;
- invocation of a constructor for direct-initialization of a class object;

- invocation of a function named in the function call syntax;
- invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax;
- invocation of the operator referenced in an expression;
- invocation of a constructor for direct-initialization of a class object;
- invocation of a user-defined conversion for copy-initialization of a class object;

Overloading

Overload resolution contexts

- invocation of a function named in the function call syntax;
- invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax;
- invocation of the operator referenced in an expression;
- invocation of a constructor for direct-initialization of a class object;
- invocation of a user-defined conversion for copy-initialization of a class object;
- invocation of a conversion function for initialization of an object of a nonclass type from an expression of class type;

Overloading
Overload resolution contexts

- invocation of a function named in the function call syntax;
- invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax;
- invocation of the operator referenced in an expression;
- invocation of a constructor for direct-initialization of a class object;
- invocation of a user-defined conversion for copy-initialization of a class object;
- invocation of a conversion function for initialization of an object of a nonclass type from an expression of class type;
- invocation of a conversion function for conversion to a glvalue or class prvalue to which a reference will be directly bound.

Overloading
Member signature adjustment

For purpose of overload resolution a member function is considered to have an extra parameter, called the implicit object parameter, which represents the object for which the member function has been called. Type of the implicit object parameter is

- „lvalue reference to cv X” for functions declared without a ref-qualifier or with the ref-qualifier
- „rvalue reference to cv X” for functions declared with the ref-qualifier

```
class X {
int f(int); // int f(A &, int) binds to rvalue
int f(int) volatile; // int f(volatile A &, int) binds
    to rvalue
int f(int) volatile &; // int f(volatile A &, int)
    cannot bind to rvalue
int f(int) const &&; // int f(const A &&, int)
};
```

Overloading
Overload resolution algorithm

Overloading
Overload resolution algorithm

- create set of candidate functions and their arguments;

Overloading
Overload resolution algorithm

- create set of candidate functions and their arguments;
- create set of viable functions

Overloading
Overload resolution algorithm

- create set of candidate functions and their arguments;
- create set of viable functions
    - correct arguments number

- create set of candidate functions and their arguments;
- create set of viable functions
    - correct arguments number
    - implicit conversion sequence for each argument should be available

Overloading

Overload resolution algorithm

- create set of candidate functions and their arguments;
- create set of viable functions
  - correct arguments number
  - implicit conversion sequence for each argument should be available
- find best viable function

Overloading

Overload resolution algorithm

- create set of candidate functions and their arguments;
- create set of viable functions
  - correct arguments number
  - implicit conversion sequence for each argument should be available
- find best viable function
  - ordering on conversion sequences

Overloading

Overload resolution algorithm

- create set of candidate functions and their arguments;
- create set of viable functions
    - correct arguments number
    - implicit conversion sequence for each argument should be available
- find best viable function
    - ordering on conversion sequences
    - F is better than G if for all arguments conversions for F are not worse than those for G and

Overloading
Overload resolution algorithm

- create set of candidate functions and their arguments;
- create set of viable functions
  - correct arguments number
  - implicit conversion sequence for each argument should be available
- find best viable function
  - ordering on conversion sequences
  - F is better than G if for all arguments conversions for F are not worse than those for G and
    - for some argument it is better or

Overloading
Overload resolution algorithm

- create set of candidate functions and their arguments;
- create set of viable functions
    - correct arguments number
    - implicit conversion sequence for each argument should be available
- find best viable function
    - ordering on conversion sequences
    - F is better than G if for all arguments conversions for F are not worse than those for G and
        - for some argument it is better or
        - in case of user defined conversion conversion of return type does matter

Overloading
Overload resolution algorithm

- create set of candidate functions and their arguments;
- create set of viable functions
  - correct arguments number
  - implicit conversion sequence for each argument should be available
- find best viable function
  - ordering on conversion sequences
  - F is better than G if for all arguments conversions for F are not worse than those for G and
    - for some argument it is better or
    - in case of user defined conversion conversion of return type does matter
    - non-template function is preferred before specializations

- create set of candidate functions and their arguments;
- create set of viable functions
  - correct arguments number
  - implicit conversion sequence for each argument should be available
- find best viable function
  - ordering on conversion sequences
  - F is better than G if for all arguments conversions for F are not worse than those for G and
    - for some argument it is better or
    - in case of user defined conversion conversion of return type does matter
    - non-template function is preferred before specializations
    - more specialized template is preffered before less specialized

Overloading
Overload resolution algorithm

- create set of candidate functions and their arguments;
- create set of viable functions
  - correct arguments number
  - implicit conversion sequence for each argument should be available
- find best viable function
  - ordering on conversion sequences
  - F is better than G if for all arguments conversions for F are not worse than those for G and
    - for some argument it is better or
    - in case of user defined conversion conversion of return type does matter
    - non-template function is preferred before specializations
    - more specialized template is preffered before less specialized
- if selected function is not available program is ill-formed

Overloading and object initialization, explanation

```
struct A;
struct B {
  B();
  B(const A&);
};

struct A {
  operator B();
};

int main() {
  A a;
  B b = a;
  // candidate functions: B(const A&) and operator B(A&)
  // operator B(A&) is better match
}
```

Thank you

Thank You