Modernes C++
Träume und Alpträume

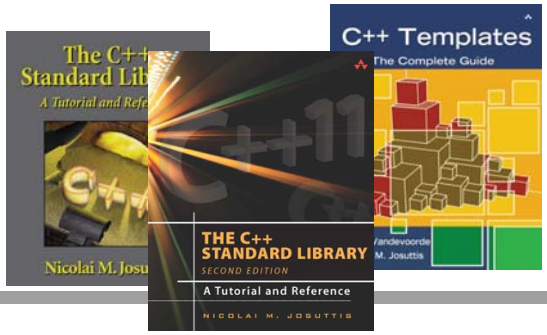
Nicolai M. Josuttis

05/17

©2017 by IT-communication.com

Nicolai M. Josuttis

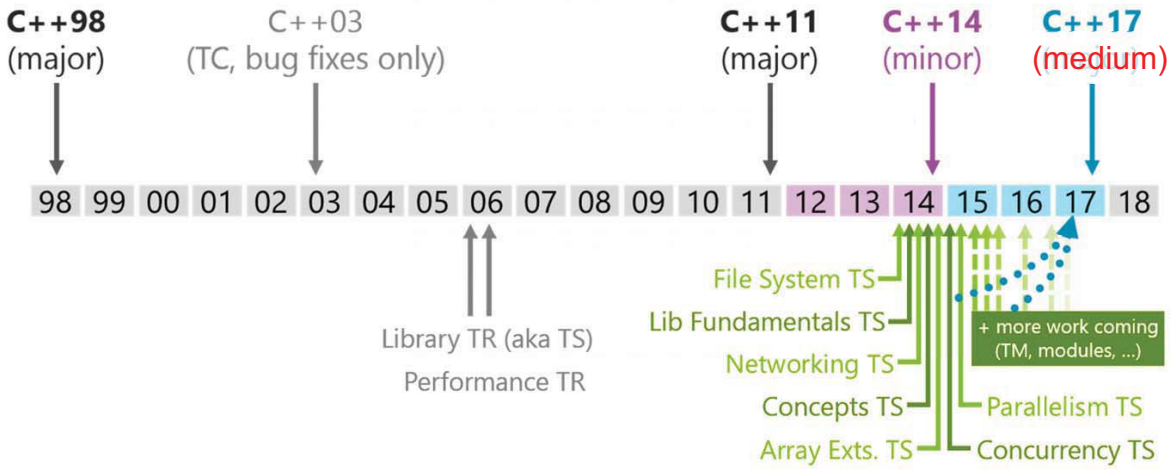
- **Independent consultant**
  - continuously learning since 1962
- **Systems Architect, Technical Manager**
  - finance, manufacturing, automobile, telecommunication
- **Topics:**
  - C++
  - SOA (Service Oriented Architecture)
  - Technical Project Management
  - Privacy (contributor of Enigmail)



©2017 by IT-communication.com

C++ Timeframe

http://isocpp.org/std/status:

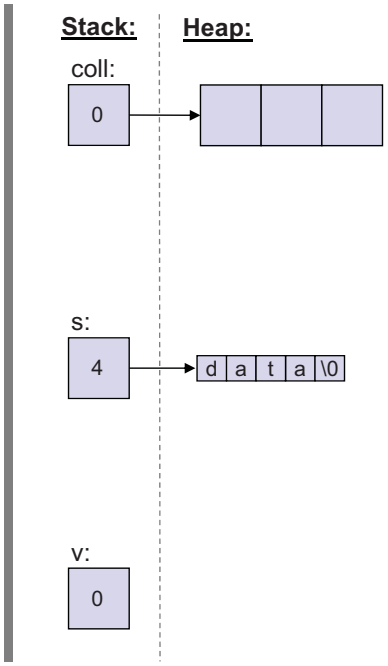


The Power of Move Semantics

Use a Naive Function returning a Vector of Strings in C++11

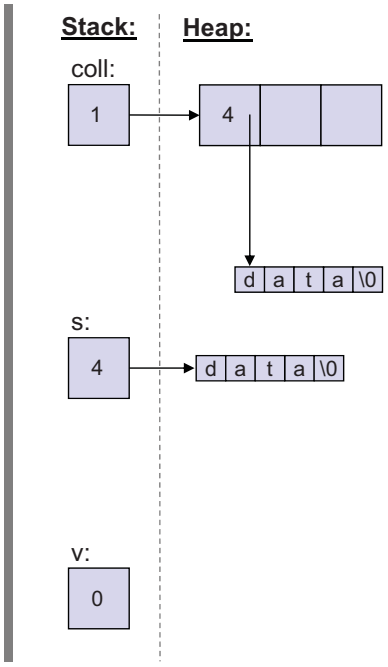
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



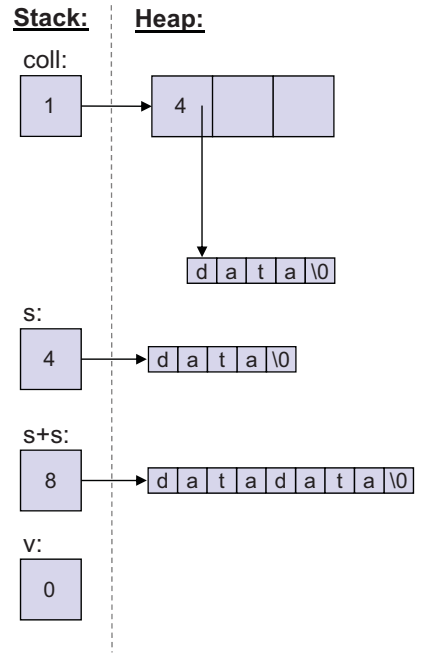
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



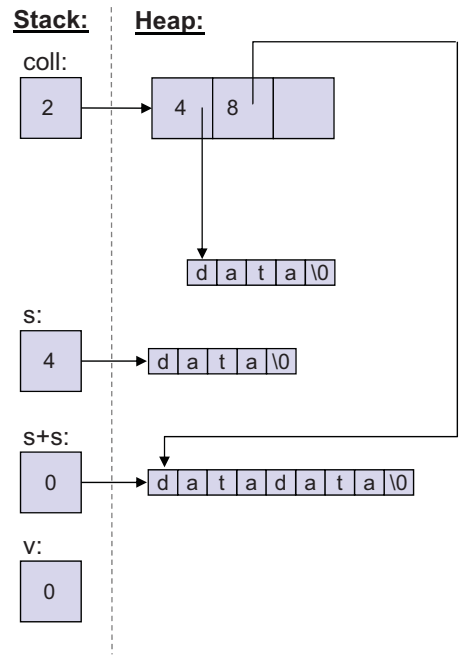
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



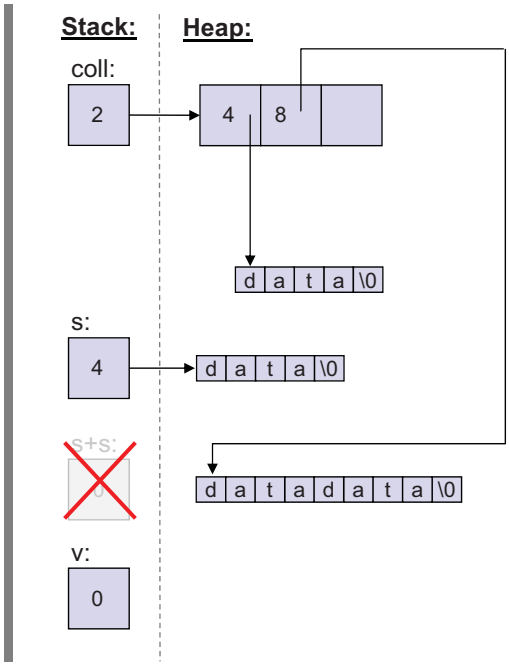
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



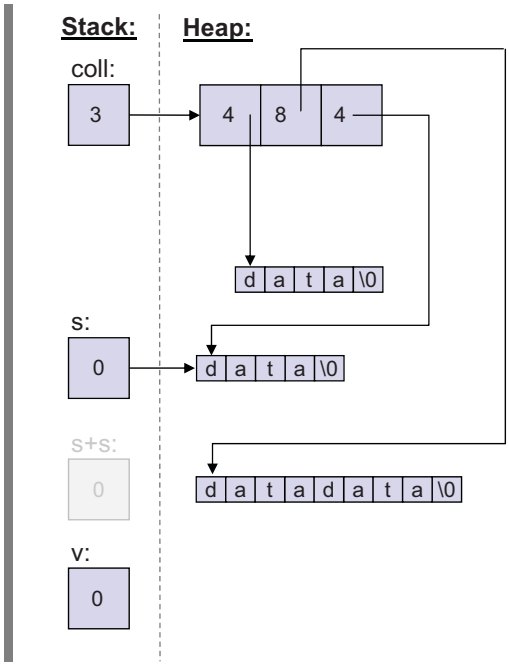
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::
                   destruct temporary

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



Move Semantics of C++11

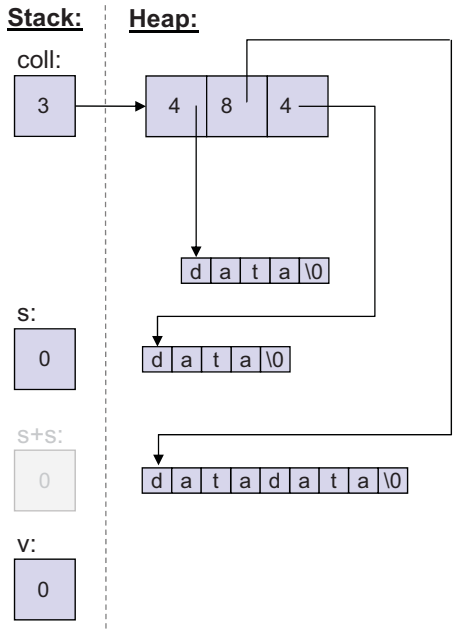
```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```

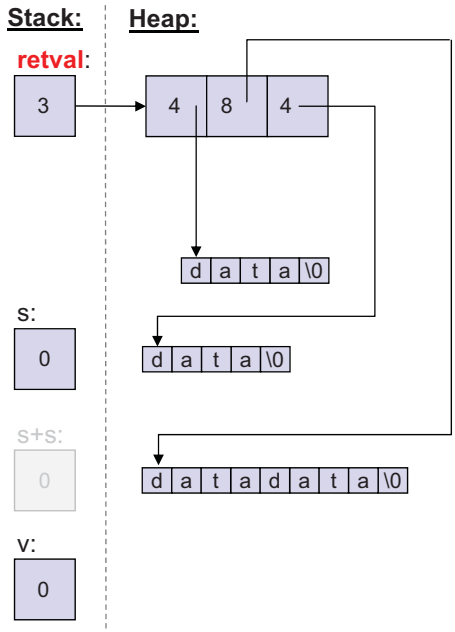
MAY move coll



Move Semantics of C++11

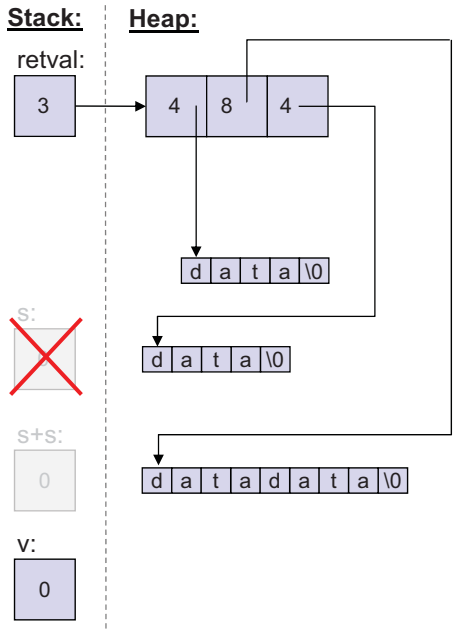
```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```

MAY move coll



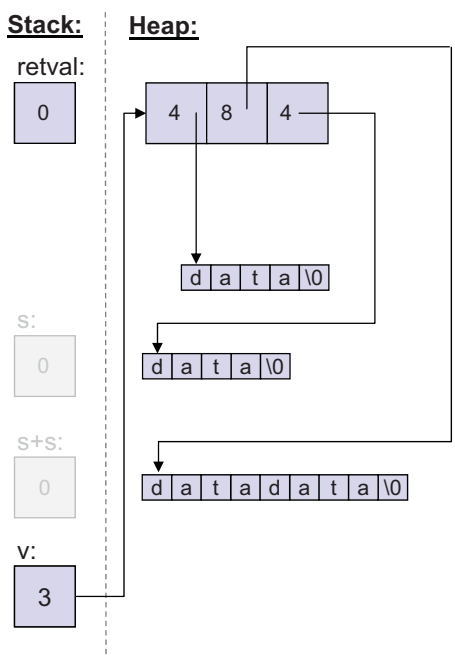
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



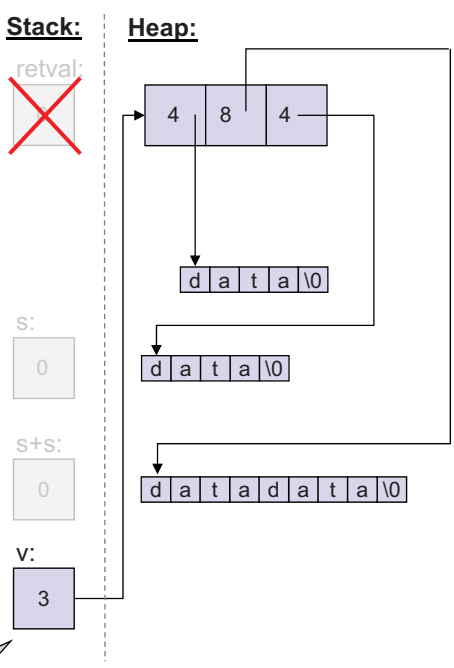
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



~~4~~ malloc/new
~~0~~ free/delete

So:

What changed with C++11?

What are the consequences?

Without Move Semantics

- **Containers have value semantics**
  - copy passed new elements into their containers
  - allows to pass rvalues (such as temporaries)
- **This leads to unnecessary copies with C++98/C++03**

```
template <typename T>
class vector {
  public:
    ...
    // insert a copy of elem:
    void push_back (const T& elem);
    ...
};
```

```
std::vector<std::string> createAndInsert ()
{
    std::vector<std::string> coll;
    std::string s("data");     // create a string s
    ...
    coll.push_back(s);         // insert a copy of s into coll
    ...                        // s is used and modified afterwards

    coll.push_back(s+s); // insert copy of temporary rvalue

    coll.push_back(s);         // insert copy of s again
                               // but s is no longer used here
    return coll;               // may copy coll
}
```

unnecessary copies in C++98 / C++03

With Move Semantics

- **With rvalue references you can provide move semantics**
- **RValue references represent modifiable object where the value is no longer needed so that you can *steal* their content**

```
template <typename T>
class vector {
  public:
    ...
    // insert a copy of elem:
    void push_back (const T& elem);
    ...
    // insert elem with its content moved:
    void push_back (T&& elem);
    ...
};
```

declares rvalue reference

```
#include <utility>         // declares std::move()

std::vector<std::string> createAndInsert ()
{
    std::vector<std::string> coll;
    std::string s("data");     // create a string s
    ...
    coll.push_back(s);         // insert a copy of s into coll
    ...                        // s is used and modified afterwards

    coll.push_back(s+s); // move temporary into coll

    coll.push_back(std::move(s));
                               // move s into coll
                               // OK, because s is no longer used
    return coll;               // may move coll
}
```

With Move Semantics

- **To support move semantics for non-trivial types you should:**
  - provide a **move constructor**
  - provide a **move assignment operator**

  where the move version is optimized to
  - steal contents from the passed object
  - and set the assigned object in a valid but undefined (or initial) state

```
class string {
  private:
    int len;        // current number of characters
    char* elems;    // array of characters

  public:
    // create a full copy of s:
    string (const string& s)
     : len(s.len) {
        elems = new char[len+1]; // new memory
        memcpy(elems,s.elems,len+1);
    }
```

```
  public:
    // create a copy of s with its content moved:
    string (string&& s)
     : len(s.len),
       elems(s.elems) {     // copy pointer to memory
        s.elems = nullptr;  // otherwise destructor of s
                            // frees stolen memory
        s.len = 0;
    }
    ...
};
```


Basic Move Support

- **Guarantees for library objects (**§17.6.5.15 [lib.types.movedfrom]**):**
  - "Unless otherwise specified, ... moved-from objects shall be placed in a valid but unspecified state."
- **Copy as Fallback**
  - If no move semantics is provided, copy semantics is used
    - unless move operations are explicitly deleted
- **Default move operations are generated**
  - Move constructor and Move assignment operator
    - pass move semantics to member

  but only if this can't be a problem
  - Only if there is no special member function defined
    - copy constructor
    - assignment operator
    - destructor

Effect of Default Move Semantics

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    Cust(const std::string& fn, const std::string& ln = "", long i = 0)
     : first(fn), last(ln), id(i) {
    }

    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.id << ": " << c.first << " " << c.last << "]";
    }
};
```

```
std::vector<Cust> v;                        // C++03:                  C++11:

v.push_back(Cust("jim","coe",42));          // 6 exp (cr+cp+cp)        4 exp (cr+cp+mv)

Cust c("joe","fix",77);                     // 4 exp (cr+cp)           4 exp (cr+cp)
v.push_back(c);                             // 2+2 exp (cp+cp)         2 exp (cp+mv)
std::cout << "c: " << c << std::endl;       // c: [77: joe fix]        c: [77: joe fix]

v.push_back(std::move(c));                  // ----                    0 exp (mv+mv)
std::cout << "c: " << c << std::endl;       // ----                    c: [77: ??? ???]
```


(Perfect) Forwarding

Forwarding Move Semantics

- **You can and have to forward move semantics explicitly:**

```
class X;

void g (X&);          // for variable values
void g (const X&);    // for constant values
void g (X&&);         // for values that are no longer used (move semantics)

void f (X& t) {
    g(t);             // t is non const lvalue => calls g(X&)
}
void f (const X& t) {
    g(t);             // t is const lvalue     => calls g(const X&)
}
void f (X&& t) {
    g(std::move(t)); // t is non const lvalue => needs std::move() to call g(X&&)
}                     // - When move semantics would always be passed,
                      //   calling g(t) twice would be a problem
X v;
const X c;
f(v);             // calls f(X&)       => calls g(X&)
f(c);             // calls f(const X&) => calls g(const X&)
f(X());           // calls f(X&&)      => calls g(X&&)
f(std::move(v)); // calls f(X&&)      => calls g(X&&)
```


Example of Improvements for Move Semantics

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    Cust(const std::string& fn, const std::string& ln = "", long i = 0)
     : first(fn), last(ln), id(i) {
    }
    Cust(std::string&& fn, std::string&& ln = "", long i = 0)
     : first(std::move(fn)), last(std::move(ln)), id(i) {
    }
    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.id << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;                          // C++03 (old class):   C++11:

v.push_back(Cust("jim","coe",42));            // 6 exp (cr+cp+cp)     2 exp (cr+mv+mv)

Cust c("joe","fix",77);                       // 4 exp (cr+cp)        2 exp (cr+mv)
v.push_back(c);                               // 2+2 exp (cp+cp)      2 exp (cp+mv)
std::cout << "c: " << c << std::endl;         // c: [77: joe fix]     c: [77: joe fix]

v.push_back(std::move(c));                    // ----                 0 mallocs (mv+mv)
std::cout << "c: " << c << std::endl;         // ----                 c: [77: ??? ???]
```

Perfect Forwarding

- **Special semantics for && with template types**
  - For temporaries, constants, and variables and the template type knows what they are
  - You can use `std::forward<>()` to keep this semantics
- **Term "*Universal Reference*"** (introduced by Scott Meyers)
  - Standard term: "*Forwarding Reference*" (introduced for C++17 with N4262)

```
void g (X&);          // for variable values
void g (const X&);    // for constant values
void g (X&&);         // for values that are no longer used (move semantics)

template <typename T>
void f (T&& t)        // t is universal/forwarding reference
{
    g(std::forward<T>(t)); // forwards move semantics
}                          // (without forward<>, only calls g(const X&) or g(X&))

X v;
const X c;

f(v);
f(c);
f(X());
f(std::move(v));
```


Example of Generic Improvements for Move Semantics

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR1, typename STR2>
    Cust(STR1&& fn, STR2&& ln = "", long i = 0)
     : first(std::forward<STR1>(fn)), last(std::forward<STR2>(ln)), id(i) {
    }

    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.id << ": " << c.first << " " << c.last << "]";
    }
};
```

`std::vector<Cust> v;`	// C++03 (old class):	C++11:
`v.push_back(Cust("jim","coe",42));`	// 6 exp (cr+cp+cp)	2 exp (cr+mv)
`Cust c("joe","fix",77);`	// 4 exp (cr+cp)	2 exp (cr)
`v.push_back(c);`	// 2+2 exp (cp+cp)	2 exp (cp+mv)
`std::cout << "c: " << c << std::endl;`	// c: [77: joe fix]	c: [77: joe fix]
`v.push_back(std::move(c));`	// ----	0 exp (mv+mv)
`std::cout << "c: " << c << std::endl;`	// ----	c: [77: ??? ???]

Fixing Broken Usage

Deducing from Default Call Arguments

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR1, typename STR2>
    Cust(STR1&& fn, STR2&& ln = "", long i = 0)
     : first(std::forward<STR1>(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.id << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;
v.push_back(Cust("jim","coe",42));

Cust c("joe","fix",77);
v.push_back(c);
std::cout << "c: " << c << std::endl;   // outputs: c: [77: joe fix]

Cust d1{"Tim"};                         // Error: can't deduce from default call arguments
Cust d2("Tim");                         // Error: can't deduce from default call arguments
```

Deducing from Default Call Arguments

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR1, typename STR2>
    Cust(STR1&& fn, STR2&& ln = std::string{""}, long i = 0)
     : first(std::forward<STR1>(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.id << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;
v.push_back(Cust("jim","coe",42));

Cust c("joe","fix",77);
v.push_back(c);
std::cout << "c: " << c << std::endl;  // outputs: c: [77: joe fix]

Cust d1{"Tim"};                       // Error: can't deduce from default call arguments
Cust d2("Tim");                       // Error: can't deduce from default call arguments
```

same error with:
STR2&& ln = ""s

Default Template Arguments

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR1, typename STR2 = std::string>
    Cust(STR1&& fn, STR2&& ln = "", long i = 0)
     : first(std::forward<STR1>(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.id << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;
v.push_back(Cust("jim","coe",42));

Cust c("joe","fix",77);
v.push_back(c);
std::cout << "c: " << c << std::endl;  // outputs: c: [77: joe fix]

Cust d1{"Tim"};                       // OK
Cust d2("Tim");                       // OK
```

Default Template Arguments

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR1, typename STR2 = std::string>
    Cust(STR1&& fn, STR2&& ln = "", long i = 0)
     : first(std::forward<STR1>(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.id << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;
v.push_back(Cust("jim","coe",42));

Cust c("joe","fix",77);
v.push_back(c);
std::cout << "c: " << c << std::endl;  // outputs: c: [77: joe fix]

Cust d1{"Tim"};                       // OK
Cust e1{d1};                          // Error: can't convert Cust to std::string
const Cust d2{"Tim"};                 // OK
Cust e1{d2};                          // OK
```


Default Template Arguments

Better match than pre-defined copy constructor for non-const objects:
- `Cust` objects
- objects derived from `Cust`

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR1, typename STR2 = std::string>
    Cust(STR1&& fn, STR2&& ln = "", long i = 0)
     : first(std::forward<STR1>(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
      return strm << "[" << c.id << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;
v.push_back(Cust("Tim","Coe",42));

Cust c("Joe","Fix",77);
v.push_back(c);
std::cout << "c: " << c << std::endl;  // outputs: c: [77: Joe Fix]

Cust d1{"Tina"};                      // OK
const Cust d2{"Bill"};                // OK
Cust e1{d1};                          // Error: "can't convert Cust to std::string"
Cust e2{d2};                          // OK
```

Using enable_if<>

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR1, typename STR2 = string,
              typename std::enable_if<!std::is_same<Cust,STR1>::value,
                                      void*>::type = nullptr>
    Cust(STR1&& fn, STR2&& ln = "", long i = 0)
     : first(std::forward<STR1>(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
    …
};

std::vector<Cust> v;
v.push_back(Cust("jim","coe",42));

Cust c("joe","fix",77);
v.push_back(c);
std::cout << "c: " << c << std::endl;  // outputs: c: [77: joe fix]

Cust d1{"Tim"};                        // OK
Cust e1{d1};                           // Error: can't convert Cust to std::string
const Cust d2{"Tim"};                  // OK
Cust e1{d2};                           // OK
```


Using enable_if<>

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template<typename STR1, typename STR2 = string,
             typename std::enable_if<!std::is_same
                                     <Cust,
                                      typename std::remove_reference
                                       <typename std::remove_const<STR1>::type
                                       >::type
                                     >::value,
                                   void*>::type = nullptr>
    Cust(STR1&& fn, STR2&& ln = "", long i = 0)
     : first(std::forward<STR1>(fn)), last(std::fo          TR2>(ln)), id(i) {
    }
    …
};
```

Wrong order:

```
typename std::remove_reference<
  typename std::remove_const<const int&>::type>::type       // => const int

typename std::remove_const<
  typename std::remove_reference<const int&>::type>::type   // => int
```

Type Traits Details

- std::**is_constructible<***T*, *Args*...**>**
  - checks whether you can construct *T* from *Args*...

```
T t(declval<Args>()...);      // must be valid
```

- std::**is_convertible<***From*, *To***>**
  - checks whether you can convert *From* to *To*

```
To test() {
   return declval<From>(); // must be valid
}
```

```
class C {
  public:
    explicit C(const C&);
}

std::is_constructible_v<C,C>      // yields true
std::is_convertible_v<C,C>        // yields false
```


Using enable_if<>

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR1, typename STR2 = string,
              typename std::enable_if<std::is_constructible<std::string,STR1>
                                      ::value, void*>::type = nullptr>
    Cust(STR1&& fn, STR2&& ln = "", long i = 0)
     : first(std::forward<STR1>(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
    …
};

std::vector<Cust> v;
v.push_back(Cust("jim","coe",42));

Cust c("joe","fix",77);
v.push_back(c);
std::cout << "c: " << c << std::endl;  // outputs: c: [77: joe fix]

Cust d1{"Tim"};                       // OK
Cust e1{d1};                          // OK
const Cust d2{"Tim"};                 // OK
Cust e1{d2};                          // OK
```

C++17: Using enable_if<>

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR1, typename STR2 = std::string,
              std::enable_if_t<std::is_constructible_v<std::string,STR1>,
                               void*> = nullptr>
    Cust(STR1&& fn, STR2&& ln = "", long i = 0)
     : first(std::forward<STR1>(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
    ...
};

std::vector<Cust> v;
v.push_back(Cust("jim","coe",42));

Cust c("joe","fix",77);
v.push_back(c);
std::cout << "c: " << c << std::endl;  // outputs: c: [77: joe fix]

Cust d1{"Tim"};                        // OK
Cust e1{d1};                           // OK
const Cust d2{"Tim"};                  // OK
Cust e1{d2};                           // OK
```


C++20: Using Concepts

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR1, typename STR2 = std::string>
    requires std::is_constructible_v<std::string,STR1>
    Cust(STR1&& fn, STR2&& ln = "", long i = 0)
     : first(std::forward<STR1>(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
    ...
};

std::vector<Cust> v;
v.push_back(Cust("jim","coe",42));

Cust c("joe","fix",77);
v.push_back(c);
std::cout << "c: " << c << std::endl;  // outputs: c: [77: joe fix]

Cust d1{"Tim"};                        // OK
Cust e1{d1};                           // OK
const Cust d2{"Tim"};                  // OK
Cust e1{d2};                           // OK
```

Default Template Arguments

```
class Cust {
  private:
    std::string first;
    std::string last;
    long        id;
  public:
    template <typename STR2 = std::string>
    Cust(const std::string& fn, STR2&& ln = "", long i = 0)
     : first(fn), last(std::forward<STR2>(ln)), id(i) {
    }
    template <typename STR2 = std::string>
    Cust(std::string&& fn, STR2&& ln = "", long i = 0)
     : first(std::move(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
    …
};

std::vector<Cust> v;
v.push_back(Cust("Tim","Coe",42));      // OK
Cust c("Joe","Fix",77);
v.push_back(c);                         // OK

Cust d1{"Tina"};                        // OK
const Cust d2{"Bill"};                  // OK
Cust e1{d1};                            // OK
Cust e2{d2};                            // OK
```

If member templates can be used as copy/move constructor or assignment operator, overload first argument instead of using a template parameter.

Summary

- **The safest way:**

```
class Cust {
    Cust(std::string fn, std::string ln = "", long i = 0)
     : first(std::move(fn)), last(std::move(ln)), id(i) {
    }
};
```

- **The common way:**

```
class Cust {
    Cust(const std::string& fn, const string& ln = "", long i = 0)
     : first(fn), last(ln), id(i) {
    }
};
```

- **The best performing way:**
  - Overload only the first argument:

```
class Cust {
    template <typename STR2 = std::string>
    Cust(const std::string& fn, STR2&& ln = "", long i = 0)
     : first(fn), last(std::forward<STR2>(ln)), id(i) {
    }
    template <typename STR2 = std::string>
    Cust(std::string&& fn, STR2&& ln = "", long i = 0)
     : first(std::move(fn)), last(std::forward<STR2>(ln)), id(i) {
    }
};
```

Summary

- **C++ is tricky**
  - You can do everything
  - You can even make every mistake
- **We have C++17 support now**
  - gcc/g++ v7.1 has full language support
  - VS2017.x has some support (will grow with updates in C++17)
- **Type traits are tricky**
  - See "***C++ Templates, 2nd ed.***"
  - will be out in September 2017, see **www.tmplbook.com**
- **C++17 is an improvement**
  - See "***Programming with C++17***"
  - probably out this year, see/register at: **www.cppstd17.com**
- **C++20 will be an improvement**

Contact

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com