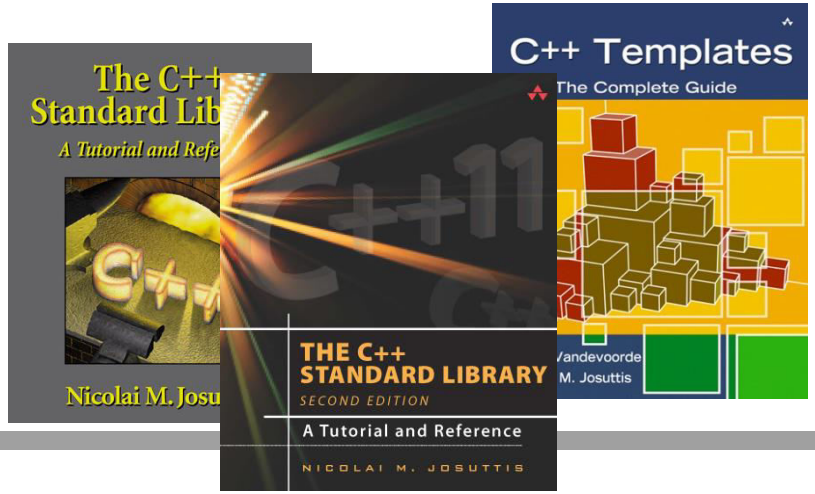
The Nightmare of Move Semantics

Nicolai M. Josuttis
IT-communication.com

04/17

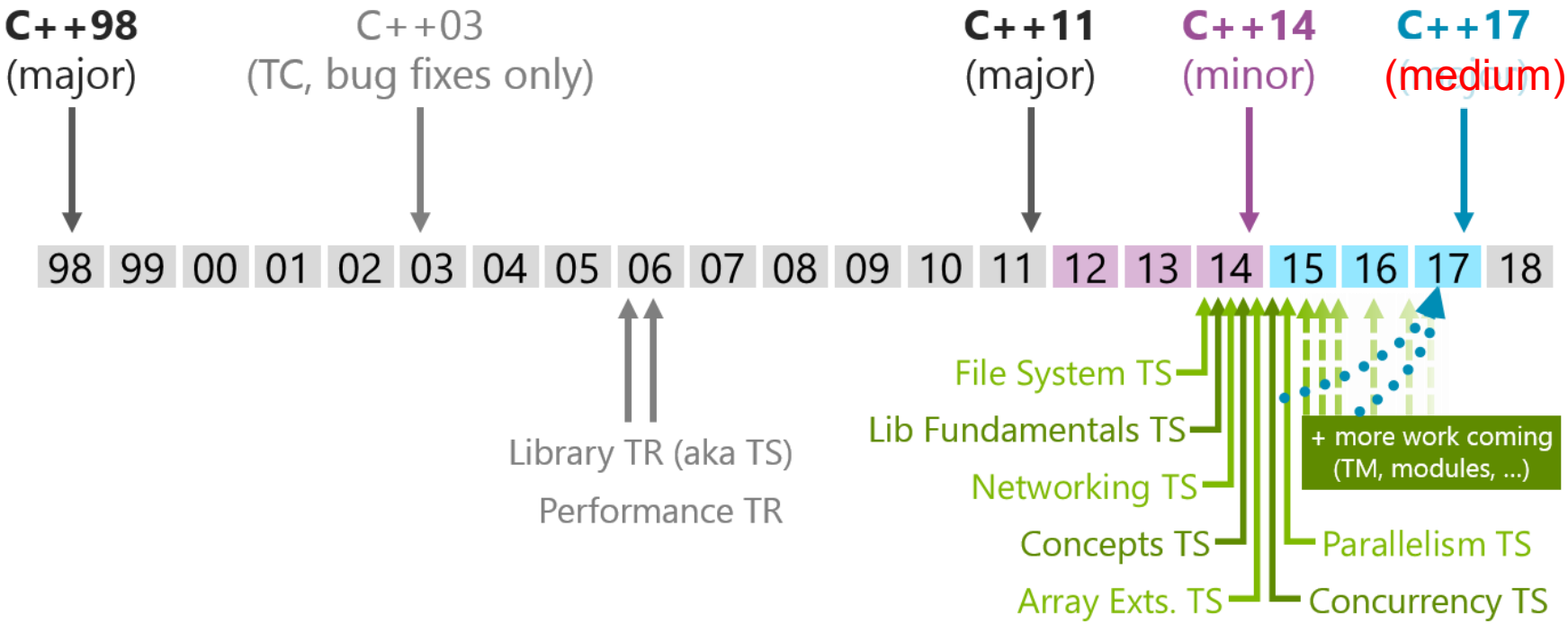
Nicolai M. Josuttis

- **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)



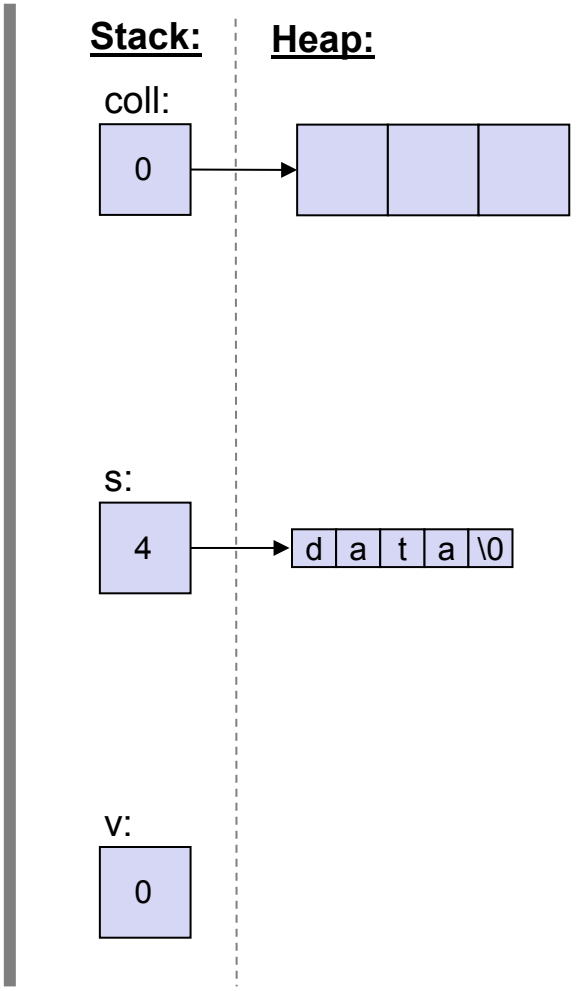
C++ Timeframe

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



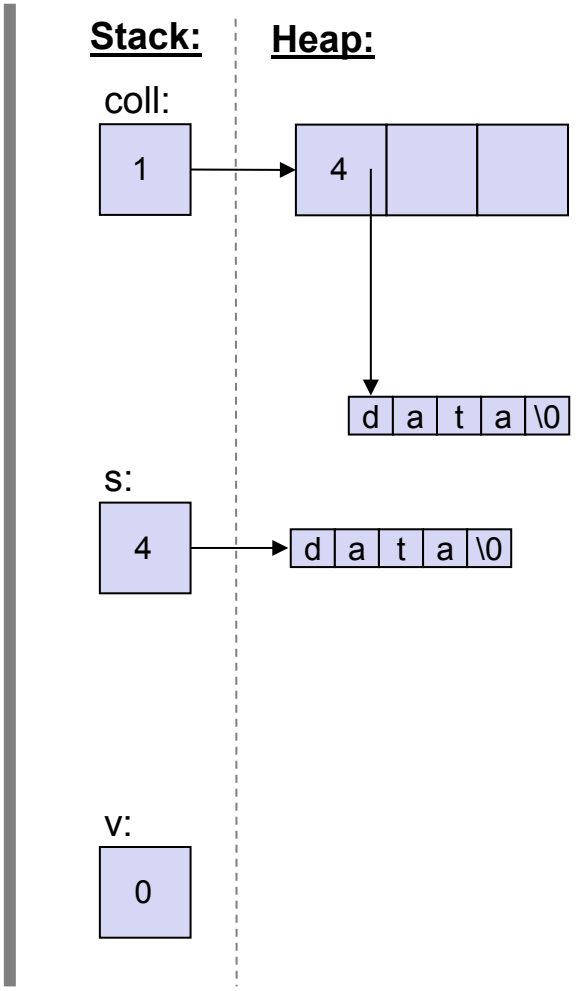
Move Semantics of C++11

```
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();
```



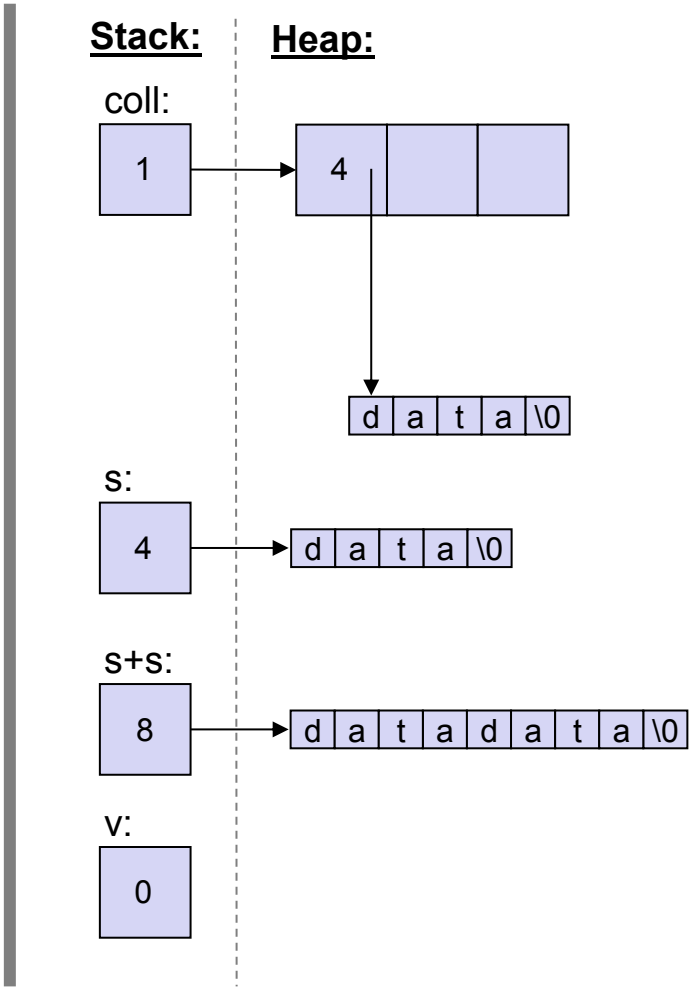
Move Semantics of C++11

```
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();
```



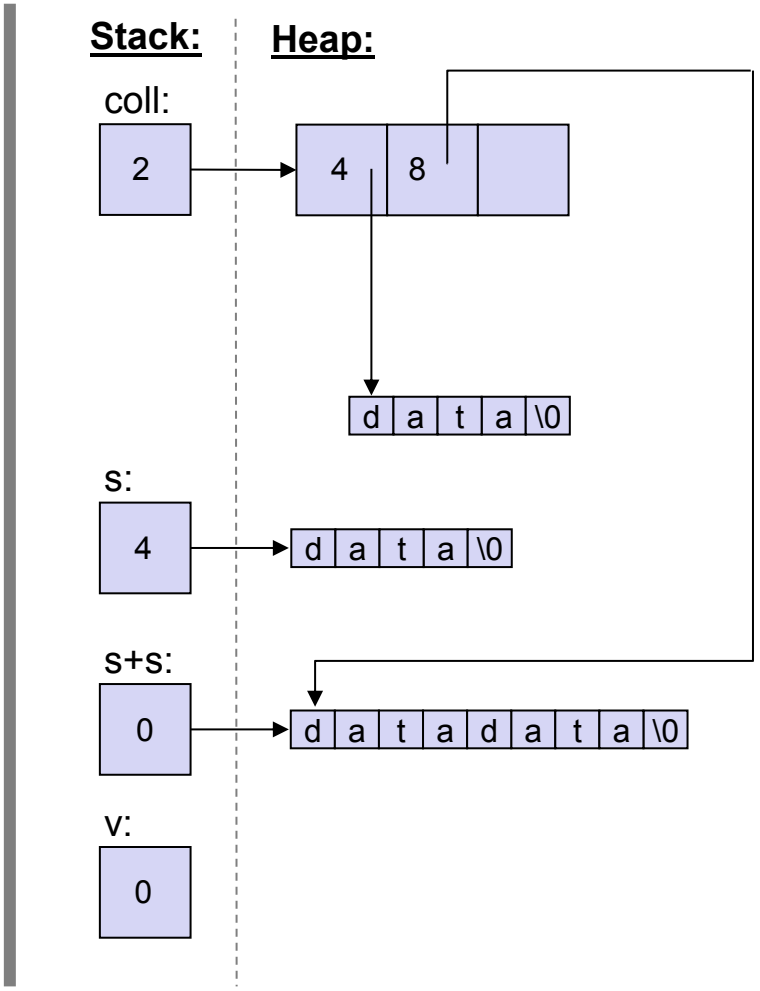
Move Semantics of C++11

```
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();
```



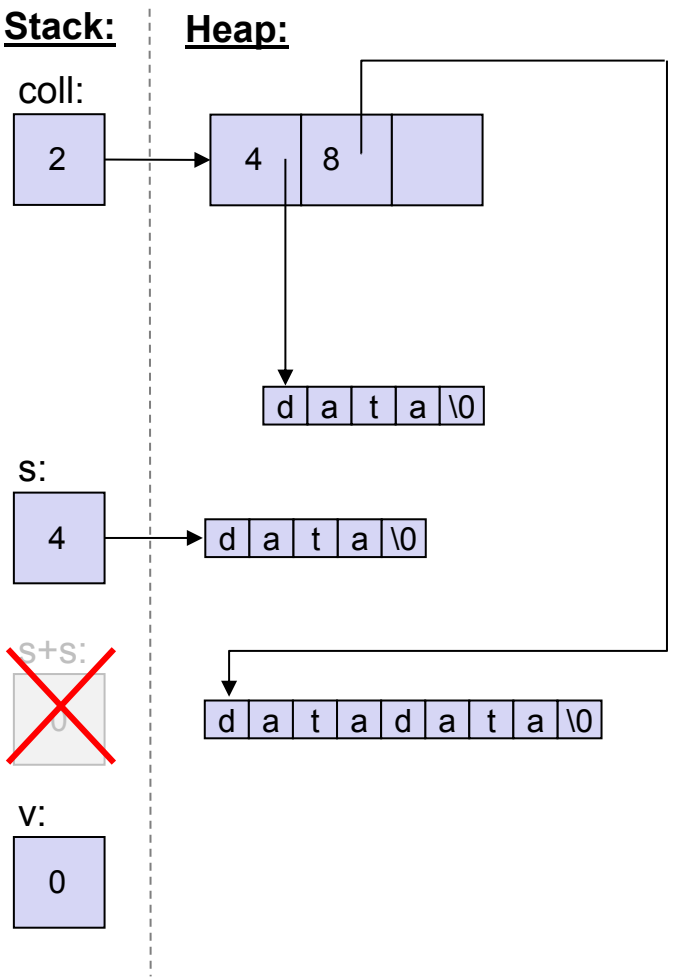
Move Semantics of C++11

```
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();
```



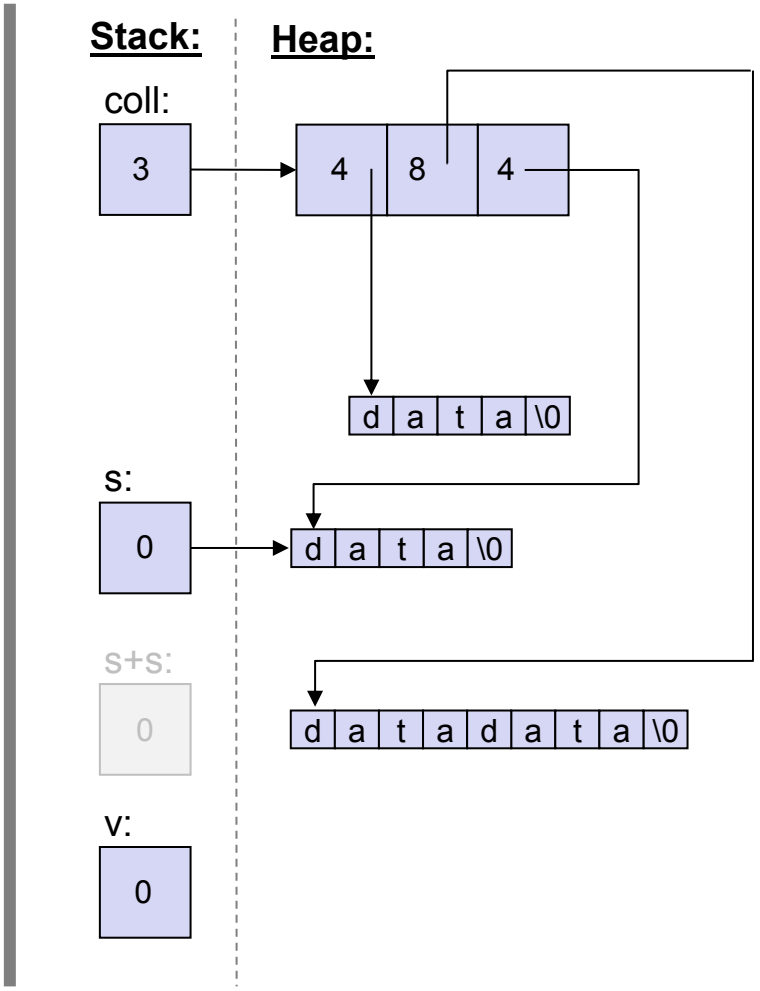
Move Semantics of C++11

```
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

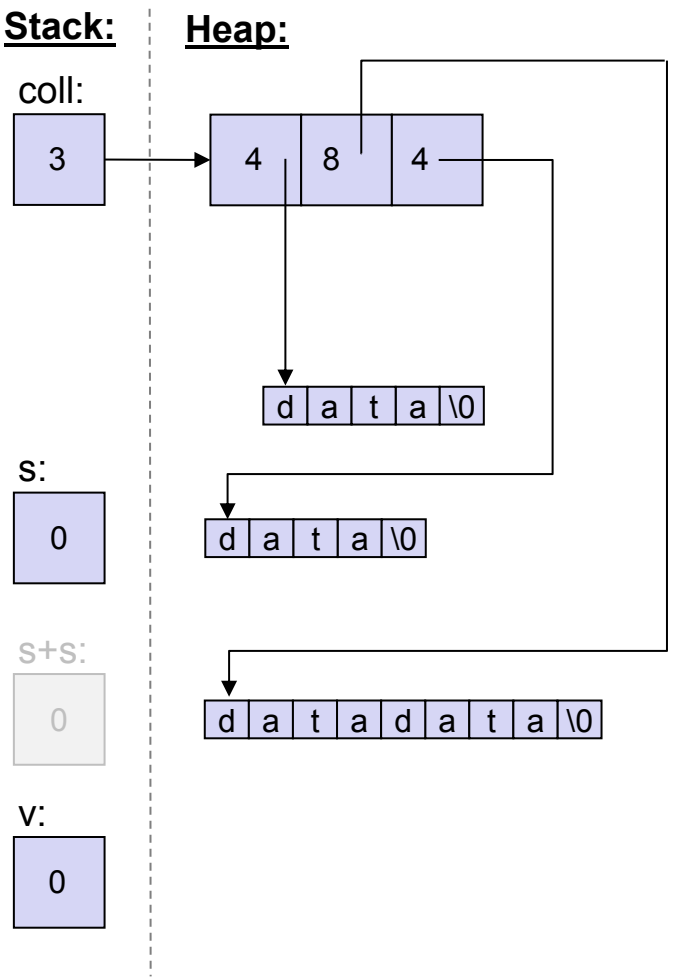
```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    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();
```

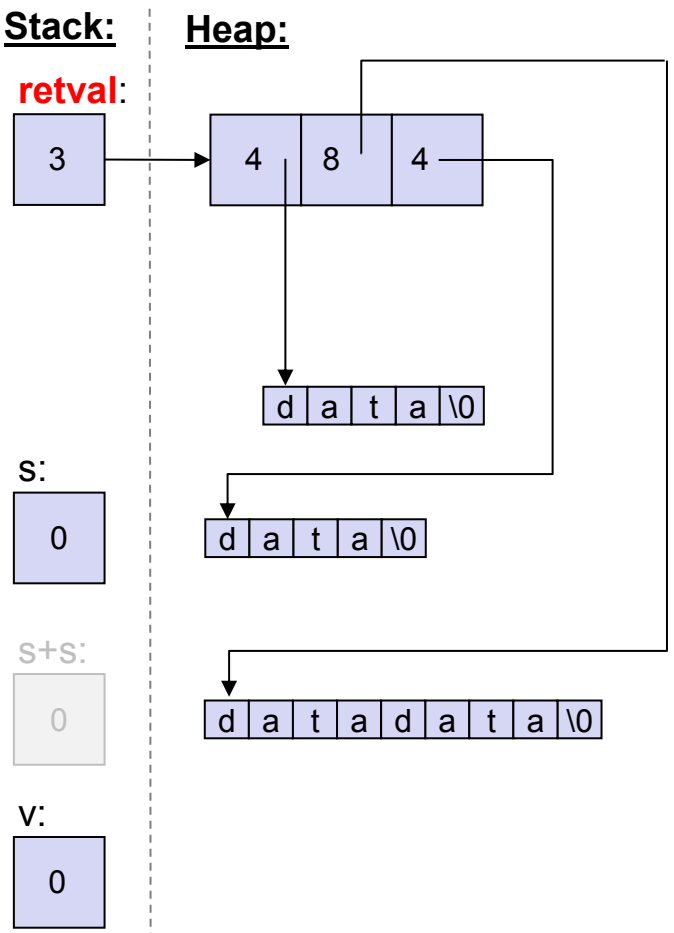
MAY move coll



Move Semantics of C++11

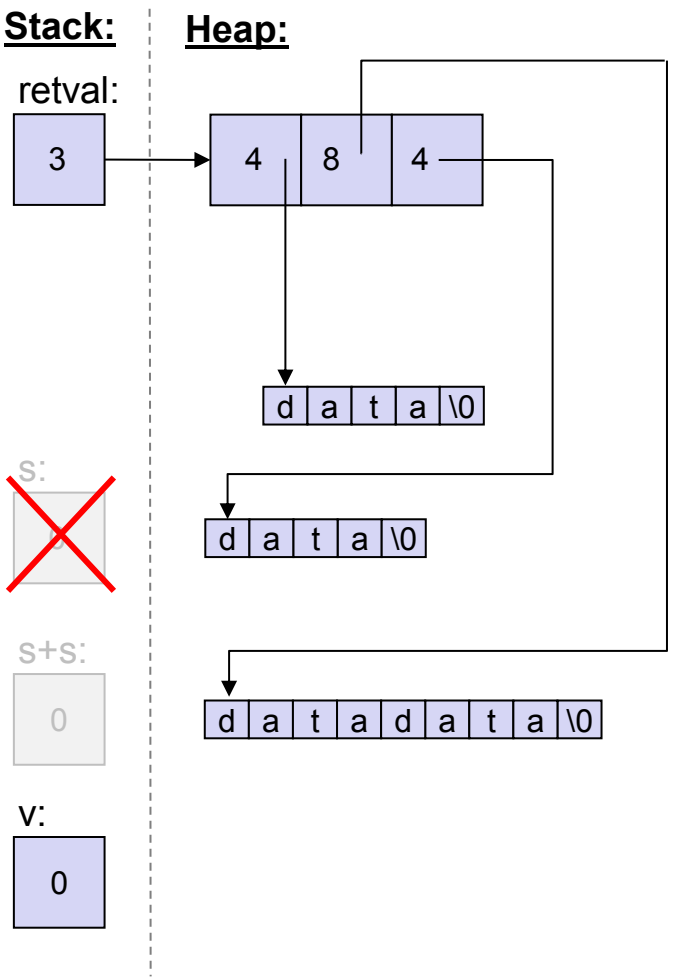
```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    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



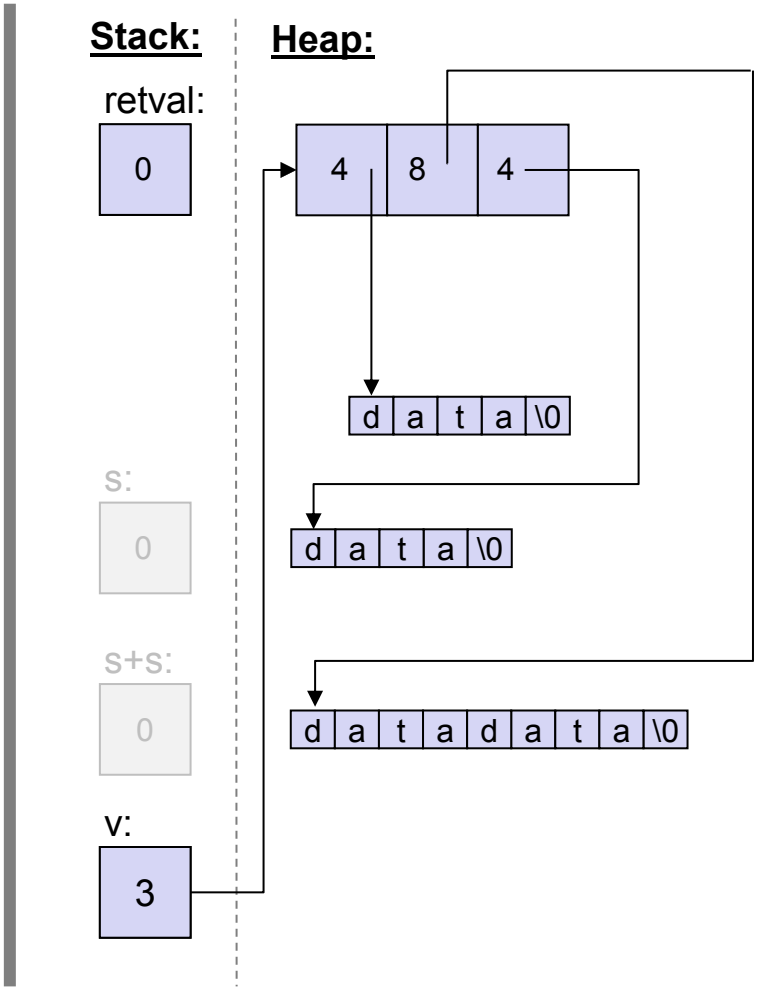
Move Semantics of C++11

```
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();
```



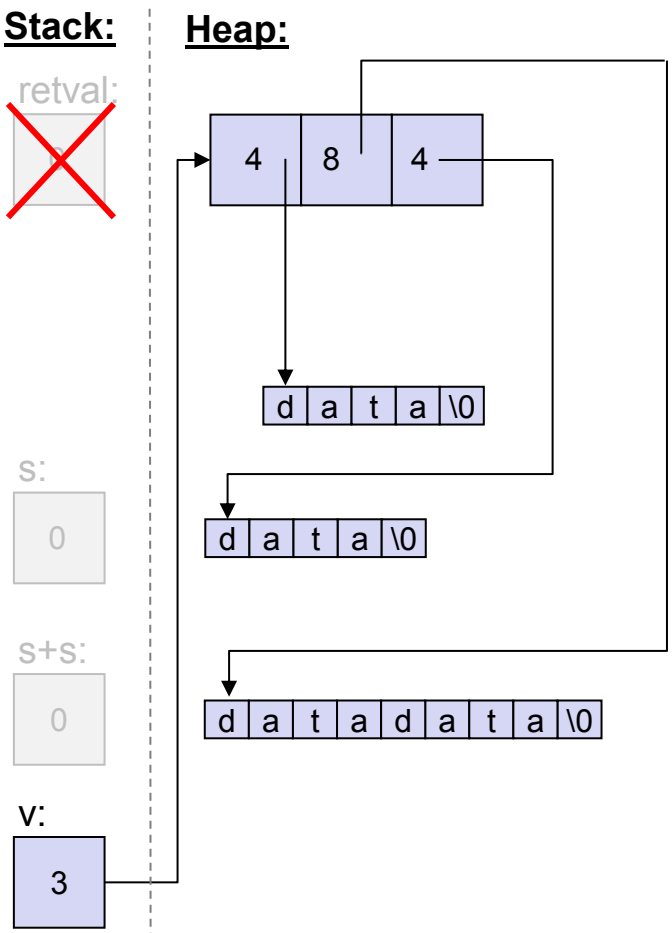
Move Semantics of C++11

```
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();
```



~~10~~ 4 malloc/new
~~8~~ 0 free/delete

So:

What changed with C++11?

What are the consequences?

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

So:

Dealing with Move Semantics

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;

v.push_back(Cust("jim","coe",42));

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

v.push_back(std::move(c));
std::cout << "c: " << c << std::endl;
```

How many expensive calls?

- i.e. potential memory allocations
- i.e. copy constructors or copy assignments for std::string
- with gcc

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;

v.push_back(Cust("jim","coe",42));

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

v.push_back(std::move(c));
std::cout << "c: " << c << std::endl;
```

C++11:

4 exp (cr+cp+mv)

4 exp (cr+cp)
2 exp (cp+mv)
`c: [77: joe fix]`

0 exp (mv+mv)
`c: [77: ??? ???]`

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: ??? ???]
```

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;

v.push_back(Cust("jim","coe",42));

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

v.push_back(std::move(c));
std::cout << "c: " << c << std::endl;
```

How many expensive calls now?

- **We had 10**

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

Perfect Forwarding in Detail

- **For "*Universal/Forwarding References*" special rules apply:**
  - Passed lvalues become lvalue references
    while passed rvalues become rvalue reference
  - Rules for reference collapsing

```
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)          // Iff lvalues were passed: T is type& => t is type&
{                       // Otherwise: T is type => t is type&&
    g(std::forward<T>(t)); // Converts t to rvalue iff T is an rvalue (reference)
}                          // (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));
```

arg is:	T is:	t is:	forward<T>(t):
lvalue			
lvalue			
prvalue	X	X&&	
xvalue	X	X&&	

Perfect Forwarding in Detail

- For "*Universal/Forwarding References*" spe
  - Passed lvalues become lvalue references while passed rvalues become rvalue reference
  - Rules for reference collapsing

Rule in §14.8.2.1 [temp.deduct.call]:

If the parameter type is an rvalue reference to a cv-unqualified template parameter and the argument is an lvalue, the type "lvalue reference to T" is used in place of T for type deduction.

Collapsing rule in C++ §8.3.2 [dcl.ref]:

`Type& &`	becomes	`Type&`
`Type& &&`	becomes	`Type&`
`Type&& &`	becomes	`Type&`
`Type&& &&`	becomes	`Type&&`

Forward definition in §20.2.4 [forward]:

Returns: `static_cast<T&&>(t)`

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

template <typename T>
void f (T&& t)        // Iff lvalues were passed: T is type&
{                     // Otherwise: T is type => t is type&&
    g(std::forward<T>(t)); // Converts t to rvalue iff T is a
}                          // (without forward<>, only calls g(const X&) or g

X v;
const X c;

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

arg is:	T is:	t is:	forward<T>(t):
lvalue	`X&`	`X& && => X&`	`t`
lvalue	`const X&`	`... => const X&`	`t`
prvalue	`X`	`X&&`	`std::move(t)`
xvalue	`X`	`X&&`	`std::move(t)`

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;

v.push_back(Cust("jim","coe",42));

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

v.push_back(std::move(c));
std::cout << "c: " << c << std::endl;
```

Covers:

```
Cust(const string&, const string&,...)
Cust(const string&, string&&,...)
Cust(string&&, const string&,...)
Cust(string&&, string&&,...)
```

How many expensive calls now?

- **We had 6**

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 << "]";
    }
};
```

Covers:

```
Cust(const string&, const string&,...)
Cust(const string&, string&&,...)
Cust(string&&, const string&,...)
Cust(string&&, string&&,...)
```

`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: ??? ???]`

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
```

Beware of Template Copy Constructors

- **Don't use forwarding template member functions that can be used as special member functions**

```
class Cust {
  private:
    std::string name;
    int         value;
  public:
    template <typename S>
    Cust(S&& n, int v = 0)
     : name(std::forward<S>(n)), value(v) {
    }

    ...
};
```

Better match than (default) copy constructor for non-const objects:
- `Cust` **objects**
- **objects derived from** `Cust`

```
Cust c("tom",42);
Cust d(c);          // Error: can't initialize name with a Cust
```

enable_if<>

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
```

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
```

Summary

- **The safest way:**

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

- **gcc/g++ with its C++17 support is awesome**
  - Thanks to Jonathan Wakely and all the others
- **Type traits are tricky**
  - See "***C++ Templates, 2nd ed.***"
  - will be out in September 2017
- **C++ is tricky**
  - You can do everything
  - You can even make every mistake
- **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**

Looking for a better title!