Training Material

Move-Semantik verständlich erklärt

Dynamischen Speicher bestmöglich nutzen

ESE Kongress
Sindelfingen
2022-12-06

Andreas Fertig

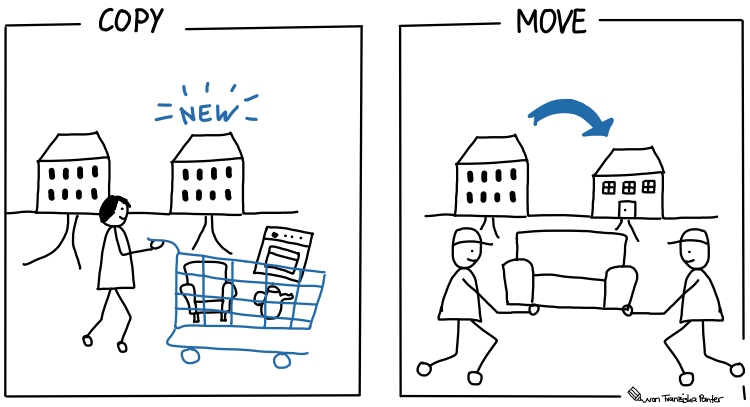
Write unique code!

Version: v1.0

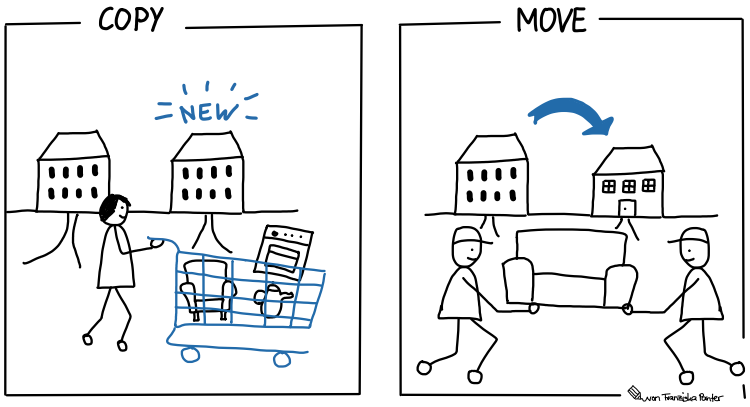
Planung, Satz und Einbandentwurf:
Andreas Fertig

Herstellung und Verlag:
Andreas Fertig

Move Semantik: Verschieben oder Duplizieren



Move Semantik: Verschieben oder Duplizieren




```
1 void Copy(char** dst, char** src, size_t size)
2 {
3   *dst = new char[size];
4   memcpy( *dst, *src, size);
5 }
```

```
1 void Move(char** dst, char** src)
2 {
3   *dst = *src;
4   *src = nullptr;
5 }
```

Überladungen

```cpp
void Fun(std::vector<int>& byRef)
{
  std::cout << "byRef\n";
}

void Fun(const std::vector<int>& byConstRef)
{
  std::cout << "byConstRef\n";
}
```


Überladungen

```cpp
void Fun(std::vector<int>& byRef)
{
  std::cout << "byRef\n";
}

void Fun(const std::vector<int>& byConstRef)
{
  std::cout << "byConstRef\n";
}
```

Using Fun

```cpp
void Use()
{
  std::vector       v{2, 3, 4};
  const std::vector cv{5, 6, 7};

  Fun(v);           // A We pass a lvalue
  Fun(cv);          // B We pass a const lvalue
  Fun({3, 5, 6});   // C We pass a temporary
}
```

Überladungen

```
1 void Fun(std::vector<int>& byRef)
2 {
3   std::cout << "byRef\n";
4 }
5
6 void Fun(const std::vector<int>& byConstRef)
7 {
8   std::cout << "byConstRef\n";
9 }
```

Using Fun

```
1 void Use()
2 {
3   std::vector       v{2, 3, 4};
4   const std::vector cv{5, 6, 7};
5
6   Fun(v);           A We pass a lvalue
7   Fun(cv);          B We pass a const lvalue
8   Fun({3, 5, 6});   C We pass a temporary
9 }
```

```
$ ./a.out
byRef
byConstRef
byConstRef
```


Überladungen

```
1  void Fun(std::vector<int>& byRef)
2 {
3   std::cout << "byRef\n";
4 }
5
6 void Fun(const std::vector<int>& byConstRef)
7 {
8   std::cout << "byConstRef\n";
9 }
10
11 void Fun(std::vector<int>&& byRvalueRef)
12 {
13   std::cout << "byMoveRef\n";
14 }
```

Überladungen

```
 1  void Fun(std::vector<int>& byRef)
 2 {
 3   std::cout << "byRef\n";
 4 }
 5
 6 void Fun(const std::vector<int>& byConstRef)
 7 {
 8   std::cout << "byConstRef\n";
 9 }
10
11 void Fun(std::vector<int>&& byRvalueRef)
12 {
13   std::cout << "byMoveRef\n";
14 }
```

```
$ ./a.out
byRef
byConstRef
byMoveRef
```

Die rvalue-Überladung aufrufen

```
1 void Fun(std::vector<int>& byRef)
2 {
3   std::cout << "byRef\n";
4 }
5
6 void Fun(const std::vector<int>& byConstRef)
7 {
8   std::cout << "byConstRef\n";
9 }
10
11 void Fun(std::vector<int>&& byRvalueRef)
12 {
13   std::cout << "byMoveRef\n";
14 }
```

```
1 void Use()
2 {
3   std::vector v{2, 3, 4};
4
5   Fun(static_cast<std::vector<int>&&>(v));
6 }
```

```
$ ./a.out
byMoveRef
```

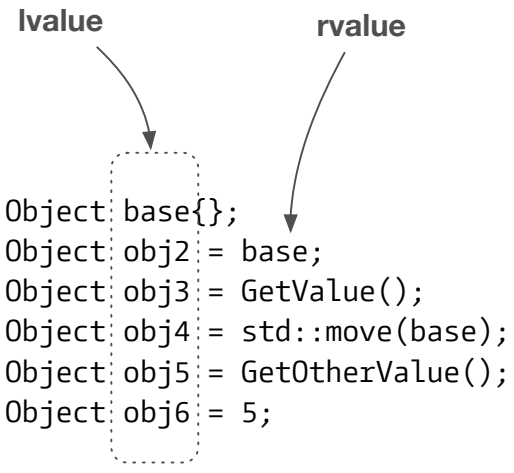
Die rvalue-Überladung aufrufen

```
void Fun(std::vector<int>& byRef)
{
  std::cout << "byRef\n";
}

void Fun(const std::vector<int>& byConstRef)
{
  std::cout << "byConstRef\n";
}

void Fun(std::vector<int>&& byRvalueRef)
{
  std::cout << "byMoveRef\n";
}
```

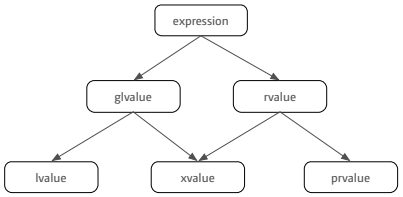
```
#include <utility>

void UseMove()
{
  std::vector v{2, 3, 4};

  Fun( std::move(v) );
}
```

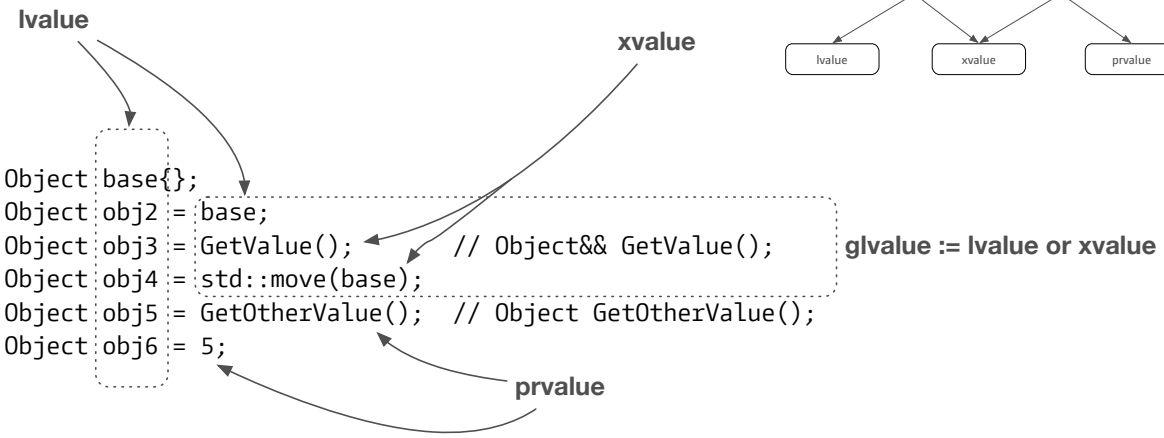
```
$ ./a.out
byMoveRef
```


Die Wertekategorien



Die Wertekategorien





Der Ort eines potenziellen Leistungsgewinns

```
1  class string {
2    size_t                  mLen{};
3    std::unique_ptr<char[]> mData{};
4
5  public:
6    string(const char* data);
7
8    string(const string& rhs);              A Copy constructor
9    string& operator=(const string& rhs);   B Copy assignment operator
10
11   string(string&& rhs);                   C Move constructor
12   string& operator=(string&& rhs);        D Move assignment operator
13
14   char* c_str() const { return mData.get(); }
15 };
```

Der Ort eines potenziellen Leistungsgewinns

```
1  string::string(const string& rhs)
2  : mLen{rhs.mLen}, mData{std::make_unique<char[]>(rhs.mLen)}
3  {
4    std::copy_n(rhs.mData.get(), mLen, mData.get());
5  }
6
7  string& string::operator=(const string& rhs)
8  {
9    if (&rhs != this) {
10     mLen  = rhs.mLen;
11     mData = std::make_unique<char[]>(mLen);
12     std::copy_n(rhs.mData.get(), mLen, mData.get());
13   }
14
15   return *this;
16 }
```


Der Ort eines potenziellen Leistungsgewinns

```
1  string::string(string&& rhs)
2  : mLen{std::exchange(rhs.mLen, 0)}, mData{std::exchange(rhs.mData, nullptr)}
3  {}
4
5  string& string::operator=(string&& rhs)
6  {
7    if (&rhs != this) {
8      mLen  = std::exchange(rhs.mLen, mLen);
9      mData = std::exchange(rhs.mData, std::move(mData));
10   }
11
12   return *this;
13 }
```

Ein verschobenes Objekt ist nichts Besonderes

```
1 string src{"Hello"};  A
2
3 string other{std::move(src)};  B
4
5 std::cout << src.c_str();  C
```


Ein verschobenes Objekt ist nichts Besonderes

Entscheidungen, Entscheidungen, Entscheidungen...

```
1  string::string(string&& rhs)
2  : mLen{std::exchange(rhs.mLen, 0)}, mData{std::exchange(rhs.mData, nullptr)}
3  {}
4
5  string& string::operator=(string&& rhs)
6  {
7    if (&rhs != this) {
8      mLen  = std::exchange(rhs.mLen, 0);
9      mData = std::move(rhs.mData);
10   }
11
12   return *this;
13 }
```

Die STL, move und ein eigenes Objekt

```
1  struct Object {
2    Object()                          { printf("ctor\n"); }
3    Object(const Object&)             { printf("copy ctor\n"); }
4    Object& operator=(const Object&) { printf("copy assign\n"); return *this; }
5    Object(Object&&)                  { printf("move ctor\n"); }
6    Object& operator=(Object&&)       { printf("move assign\n"); return *this; }
7  };
8
9  int main()
10 {
11   std::vector<Object> v{};
12
13   v.push_back(Object{});
14
15   printf("second element\n");
16   v.push_back(Object{});
17 }
```

Die STL, move und ein eigenes Objekt

```
1  struct Object {
2    Object()                          { printf("ctor\n"); }
3    Object(const Object&)             { printf("copy ctor\n"); }
4    Object& operator=(const Object&) { printf("copy assign\n"); return *this; }
5    Object(Object&&)                  { printf("move ctor\n"); }
6    Object& operator=(Object&&)       { printf("move assign\n"); return *this; }
7  };
8
9  int main()
10 {
11   std::vector<Object> v{};
12
13   v.push_back(Object{});
14
15   printf("second element\n");
16   v.push_back(Object{});
17 }
```

```
$ ./a.out
ctor
move ctor
second element
ctor
move ctor
copy ctor
```

Die STL, move und ein eigenes Objekt

```
1  struct Object {
2    Object()                                  { printf("ctor\n"); }
3    Object(const Object&)                     { printf("copy ctor\n"); }
4    Object& operator=(const Object&)          { printf("copy assign\n"); return *this; }
5    Object(Object&&)                noexcept  { printf("move ctor\n"); }
6    Object& operator=(Object&&)     noexcept  { printf("move assign\n"); return *this; }
7  };
8
9  int main()
10 {
11   std::vector<Object> v{};
12
13   v.push_back(Object{});
14
15   printf("second element\n");
16   v.push_back(Object{});
17 }
```

```
$ ./a.out
ctor
move ctor
second element
ctor
move ctor
move ctor
```


Compiler-generierte spezielle Mitgliedsfunktionen

```
1  class Object {
2    int* _data{};
3
4  public:
5    Object() : _data{new int{6}} {}
6    ~Object() { delete _data; }
7  };
```

Compiler-generierte spezielle Mitgliedsfunktionen

```
1  class Object {
2    int* _data{};
3
4  public:
5    Object() : _data{new int{6}} {}
6    ~Object() { delete _data; }
7
8    Object(const Object& rhs) : _data{new int{*rhs._data}} {}
9
10   Object& operator=(const Object& rhs)
11   {
12     if (&rhs != this) {
13       delete _data;
14       _data = new int{*rhs._data};
15     }
16
17     return *this;
18   }
19 };
```


Compiler-generierte spezielle Mitgliedsfunktionen

			compiler implicitly declares					
					copy		move	
			default ctor	dtor	ctor	assignment	ctor	assignment
user declares		**Nothing**	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
		Any ctor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
		default ctor	*user declared*	defaulted	defaulted	defaulted	defaulted	defaulted
		dtor	defaulted	*user declared*	defaulted	defaulted	not declared	not declared
	copy	**ctor**	not declared	defaulted	*user declared*	defaulted	not declared	not declared
		assignment	not declared	defaulted	defaulted	*user declared*	not declared	not declared
	move	**ctor**	not declared	defaulted	deleted	deleted	*user declared*	not declared
		assignment	not declared	defaulted	deleted	deleted	not declared	*user declared*

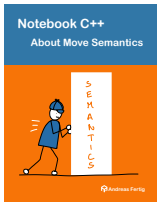
Quelle: [1]

Nutzen Sie `std::move` nur selten

- Im Allgemeinen ist der Compiler unser Freund.
- Temporäre Objekte werden automatisch verschoben.
- Für Rückgabewerte kann der Compiler Optimierungen wie Copy-Elision anwenden. Sie schlagen Copy-Elision nicht mit Verschieben!

}

Ich bin Fertig.



fertig.to/babm

Verwendete Compiler & Typografie

Verwendete Compiler

- Compiler welche zum Übersetzen des (meisten) Codes verwendet wurden.
  - g++ 11.1.0
  - clang version 14.0.0 (https://github.com/tru/llvm-release-build fc075d7c96fe7c992dde351695a5d25fe084794a)

Typografie

- Hauptschrift:
  - Camingo Dos Pro by Jan Fromm (https://janfromm.de/)
- Code-Schrift:
  - CamingoCode by Jan Fromm licensed under Creative Commons CC BY-ND, Version 3.0 http://creativecommons.org/licenses/by-nd/3.0/

Quellen

[1] Hinnant H., "Everything You Ever Wanted To Know About Move Semantics (and then some)", *ACCU*, Apr. 2014. https://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf

Bilder:

2: Franziska Panter
3: Franziska Panter
28: Franziska Panter

Nächste Events

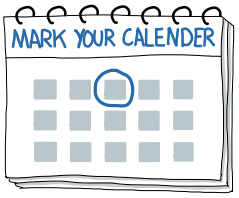
Vorträge

- *C++ Coroutines from scratch*, OOP, 08. Februar 2023

Schulungen

- *C++ Clean Code – Best Practices für Programmierer*, golem Akademie, 27. - 29. März 2023

Zukünftige Vorträge: https://andreasfertig.com/de/vortraege/.
Mehr zu meinen Schulungsangeboten gibt es hier: https://andreasfertig.com/de/schulungen/.
Immer aktuell Informiert? Hier geht es zum Newsletter: https://andreasfertig.com/newsletter/.



Über Andreas Fertig

Foto: Kristijan Matic www.kristijanmatic.de

Andreas Fertig, Geschäftsführer der Unique Code GmbH, ist erfahrener Trainer und Dozent für C++ für die Standards 11 bis 20.
Andreas engagiert sich im C++ Standardisierungskomitee, in dem die neuen Standards erarbeitet werden. Auf internationalen Konferenzen präsentiert er, wie sich Code besser schreiben lässt. Er publiziert Fachartikel z.B. für das iX Magazin und hat mehrere Fachbücher zu C++ veröffentlicht.
Mit C++ Insights (https://cppinsights.io) hat Andreas ein international anerkanntes Werkzeug geschaffen, das Nutzenden ermöglicht, hinter die Kulissen von C++ zu schauen und dadurch Konstrukte noch besser zu verstehen.
Vor seiner Tätigkeit als Trainer und Berater arbeitete er zehn Jahre für die Philips Medizin Systeme GmbH als C++ Softwareentwickler und Architekt mit Schwerpunkt auf eingebetteten Systemen.
Andreas ist online unter andreasfertig.com erreichbar.