9.3.13 unique_copy

```
template<class InputIterator, class OutputIterator>
OutputIterator
unique_copy(InputIterator first, InputIterator last, OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator
unique_copy(InputIterator first, InputIterator last, OutputIterator result,
      BinaryPredicate pred);
```

Für einen Bereich [first, last), der nicht leer ist, kopiert unique_copy aus Gruppen aufeinander folgender äquivalenter Elemente nur das jeweils erste Element in den bei result beginnenden Bereich. Für die nicht zu kopierenden Elemente gilt *(i - 1) == *i beziehungsweise pred(*(i - 1), *i) == true, wobei i ein Iterator aus dem Bereich [first + 1, last) ist. Um portable Programme zu erhalten, sollten für unique_copy nur Prädikate eingesetzt werden, die prüfen, ob zwei Elemente äquivalent sind (siehe auch unique). Es werden last - first - 1 Vergleiche beziehungsweise Aufrufe von pred vorgenommen. Für *Forward-Iteratoren* wird der Funktionsrumpf für unique_copy ohne Prädikat typischerweise wie folgt definiert:

```
if (first == last)
      return result;
*result = *first;
while (++first != last)
      if (*result != *first)
            *++result = *first;
return ++result;
```

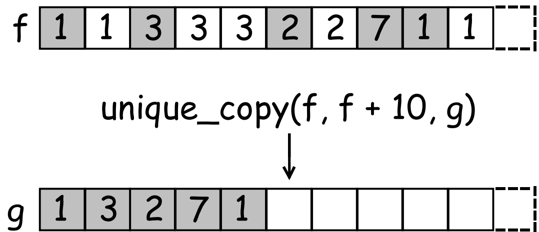
Diese Implementierung vergleicht das aktuelle Element *first mit dem zuletzt kopierten Element *result. Für *Output-Iteratoren* gestaltet sich die Implementierung etwas umständlicher, weil bei diesen *result nicht gelesen werden kann. Der Rückgabewert ist ein Iterator auf das Ende des resultierenden Bereichs. Damit Elemente nicht fälschlicherweise überschrieben werden, sollte result nicht im Bereich [first, last) liegen.

algorithmen/unique_copy.cpp

```
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;

int main() {
   int f[] = { 1, 1, 3, 3, 3, 2, 2, 7, 1, 1 }, g[10];
   int* z = unique_copy(f, f + 10, g);
   unique_copy(g, z, ostream_iterator<int>(cout, "  "), equal_to<int>());
}
```

Wie die folgende Abbildung zeigt, werden durch den ersten Aufruf von unique_copy nur fünf Elemente von f nach g kopiert.



Durch den zweiten Aufruf von unique_copy werden alle Elemente von g ausgegeben:

```
1 3 2 7 1
```

9.3.14 unique

```
template<class ForwardIterator>
ForwardIterator
unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator
unique(ForwardIterator first, ForwardIterator last, BinaryPredicate pred);
```

Für einen Bereich, der nicht leer ist, eliminiert unique aus Gruppen aufeinander folgender äquivalenter Elemente alle Elemente bis auf das jeweils erste. Für die zu eliminierenden Elemente gilt *(i - 1) == *i beziehungsweise pred(*(i - 1), *i) == true, wobei i ein Iterator aus dem Bereich [first + 1, last) ist. unique entfernt also aus Folgen gleicher Elemente alle bis auf das erste. Dazu werden last - first - 1 Vergleiche beziehungsweise Aufrufe von pred benötigt, sofern der Bereich nicht leer ist. Ein typischer Funktionsrumpf für unique ohne Prädikat basiert auf unique_copy und lautet:

```
first = adjacent_find(first, last);
return unique_copy(first, last, first);
```

Es wird ein Iterator auf das Ende des resultierenden Bereichs zurückgegeben. Bei Einsatz des Prädikats equal_to liefern beide Versionen das gleiche Ergebnis.

algorithmen/unique.cpp

```
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;

int main() {
   int f[] = { 1, 1, 3, 3, 3, 7, 1, 1, 2, 2 };
   int* z1 = unique(f, f + 10);
```

```
    cout << "[f, f + 10) = ";
    copy(f, f + 10, ostream_iterator<int>(cout, "  "));
    cout << "\n[f, z1) = ";
    copy(f, z1, ostream_iterator<int>(cout, "  "));
    int* z2 = unique(f, z1, greater<int>());  // siehe Text
    cout << "\n[f, z2) = ";
    copy(f, z2, ostream_iterator<int>(cout, "  "));
}
```

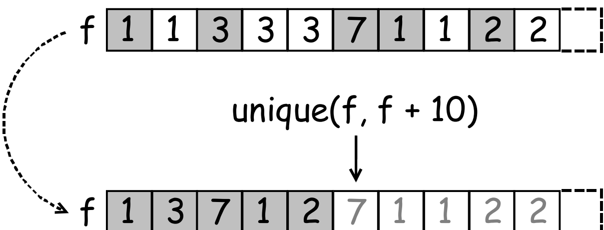
Die Ausgabe des Programms ist:

```
[f, f + 10) = 1  3  7  1  2  7  1  1  2  2
[f, z1) = 1  3  7  1  2
[f, z2) = 1  3  7
```

Die zweite Zeile verdeutlicht, dass Elemente auch nach dem Aufruf von unique noch mehrfach enthalten sein können. Wenn dies unerwünscht ist, muss der Bereich vorher sortiert werden.

Für die letzte Zeile werden derzeit auch andere Ergebnisse (speziell 1 3 7 2) diskutiert, da die Funktionsspezifikation im Standard diesen Fall bisher nicht ausdrücklich berücksichtigt. Um portable Programme zu erhalten, sollten für unique deshalb nur Prädikate eingesetzt werden, die prüfen, ob zwei Elemente äquivalent sind.

Wie die folgende Abbildung zeigt, werden Elemente nicht tatsächlich entfernt, sondern lediglich mit nachfolgenden überschrieben. Für vector, deque und string kann es daher sinnvoll sein, nach unique die Elementfunktion erase aufzurufen (vgl. Seite 185).



Für die Klasse list gibt es gleichnamige Elementfunktionen, die Elemente tatsächlich löschen (siehe Seite 88).