Praxisorientierte Einführung in C++
Lektion: "Konstanten und Variablen"

Christof Elbrechter

Neuroinformatics Group, CITEC

April 24, 2014

Table of Contents

Vordefinierte Typen

- C++ stellt eine Anzahl von Typen zur Verfügung, die benutzt werden können, um
  - z.B. einfache Arithmetik zu betreiben
  - Ein- und Ausgabe von Text zu realisieren
  - Komplexere Typen zusammenzusetzen
    - `struct`, `class`, `union` und `enum`
- Ganzzahltypen
  - `char`, `wchar_t`, `short int`, `int`, `long int`
    - `signed` und `unsigned` Varianten
- Fliesskommazahltypen
  - `float`, `double`, `long double`
- Wahrheitswerte
  - `bool` mit Werten `true` und `false`

Vordefinierte Typen - Implementationsabhängig

- Vordefinierte Datentypen unterscheiden sich:
  - Anhand des physikalischen Speicherbedarfs
  - Anhand der Interpretation der Binärdaten
  - Vieles davon ist implementierungsabhängig (mit Einschränkungen)

Ganzzahltypen

- Ganzzahlwerte ergeben sich i.d.R indem man binär "hochzählt" (2er-Komplement[1]):
- 00000000, 00000001,00000010,...11111111
- "Most-significant-bit" entspricht dem Vorzeichen für vorzeichenbehaftete Datentypen
- Vorzeichen-Bit eines N-Bit Datentypen entspricht rechnerisch -2^{N-1}
- Also (mit Vorzeichen im höchsten Bit oder nicht):
  - $2^0 bit_0 + 2^1 bit_1 + 2^2 bit_2 + ... + 2^{N-1} bit_{N-1}$
  - $2^0 bit_0 + 2^1 bit_1 + 2^2 bit_2 + ... - 2^{N-1} bit_{N-1}$

[1] http://de.wikipedia.org/wiki/Zweierkomplement

Fließkommazahlen

- Ergeben sich aus einer Formel der Art: $Vorzeichen * Basis^{Exponent}$
- Wertebereiche und Bitanzahl für Basis und Exponent sind festgelegt
- Es können Rundungsfehler auftreten (... Numerik Vorlesung)
- Genaue Interpretation ist zu komplex, um sie hier zu besprechen
  - Für Hintergrundinformationen siehe IEEE 754 Standard[2]

[2] http://de.wikipedia.org/wiki/IEEE_754

Vorzeichen - Signed vs. unsigned

- Für jeden Ganzzahltypen `T` existiert eine `unsigned` und eine `signed` Variante
  - `T` alleine steht *meistens* für die `signed` Variante
  - Ausnahme hier: `char` - Implementierungsabhängig, ob `signed` oder `unsigned`

Beispiele für Ganzzahlttypen

```
int a;                  // == signed int
unsigned int b;
signed int c;
char d;                 // signed oder unsigned??
signed char e;
unsigned long int f;
```

- Fliesskommazahltypen sind dagegen *immer* vorzeichenbehaftet

Relative Größen

- C++-Standard definiert nur relative Größenangaben
- Fest steht:
- Ein `char` ist i.d.R ein Byte groß
  - Kann also *i.d.R* 256 verschiedene Werte darstellen
  - Es ist abhängig vom Compiler/OS, ob dieser Type `signed` oder `unsigned` interpretiert wird
  - Aber: `char`, `signed char` und `unsigned char` sind gleich groß
- `int` ist der effizienteste Ganzzahl-Datentyp
- Ansonsten gilt[3]:

```
sizeof(char) <= sizeof(short int) <= sizeof(int) <= sizeof(long int) <= sizeof(long long int)
```

und

```
sizeof(float) <= sizeof(double) <= sizeof(long double)
```

[3] `long long` ist kein standard Typ

Abkürzungen für Typenbezeichner

- Statt `short int` kann man `short` schreiben
- Statt `long int` kann man `long` schreiben
- Statt `unsigned int` kann man `unsigned` schreiben (und äquivalent für `signed int`)
- Statt `long long int` kan man `long long` schreiben

Alles äquivalent

```
long int i;
long i;
signed long i;
```

- Es gibt noch mehr Abkürzungen (auf die hier aber nicht eingegangen wird)

size_t für Größenangaben

- Für Größenangaben eingebauter Typen wird i.d.R. der Typ size_t verwendet
- Vorzeichenbehaftung: implementationsabhängig
- Bei Schleifen über die Länge von built-in Typen sollte immer ein size_t verwendet werden
  - Verwendung von int oder unsigned int ist nicht auf jedem System gültig
  - Führt zu Kompilerwarnung ("comparison between signed and unsigned ...")
  - Globales Deaktivieren von Warnungen sollte vermieden werden

Beispiel

```
std::string text = "Hello World";
for(int i=0;i<text.length();++i){    // Intuitiv, aber unguenstig
  ...
}
for(size_t i=0;i<text.length();++i){  // besser!
  ...
}
```

Datentypen mit absoluter Größe

- Für Datentypen mit absoluter Größe, benutze Header `stdint.h`
- z.B. `int8_t`, `int32_t` oder `uint16_t`

Größe eines Typs bestimmen - sizeof

- Mittels `sizeof` kann die Größe eines Datentyps in der Einheit `sizeof(char)` ermittelt werden
- D.h. wieviel Speicher belegt ein Element dieses Datentyps
- **Vorsicht** bei Zeigern und Arrays. Dazu später mehr

sizeof - Beispiel

```
#include <iostream>

int main(){
   std::cout << "int: " << sizeof(int) << std::endl;
   std::cout << "char: " << sizeof(char) << std::endl;
   std::cout << "float: " << sizeof(double) << std::endl;
   ...
   return 0;
}
```

Größen von Variablen - sizeof

- sizeof kann auch benutzt werden, um Speicherbedarf von Variable auszugeben

```
int i;
std::cout << sizeof(i) << std::endl;
```

Überblick für typisches 32 bit Linux System

Typ	Größe	Wertebereich (32Bit gcc)	Relative Größe
bool	1 Byte	true oder false bzw. 0 oder 1	≥ 1 und $\leq$ long
char	1 Byte	256 Zeichenwerte	$= 1$ und $\leq$ wchar_t
wchar_t	4 Byte	-2147483648 bis 2147483647	$\geq$ char und $\leq$ long
unsigned char	1 Byte	0 bis 255	$= 1$ und $\leq$ wchar_t
unsigned short int	2 Byte	1 bis 65535	$\geq$ char und $\leq$ long
short int	2 Byte	-32768 bis 32767	$\geq$ char und $\leq$ long
int	4 Byte	-2147483648 bis 2147483647	$\geq$ short int und $\leq$ long
unsigned int	4 Byte	0 bis 4294967295	$\geq$ short int und $\leq$ long
long	4 Byte	-2147483648 bis 2147483647	$\geq$ int und $\leq$ long long
unsinged long	4 Byte	0 bis 4294967295	$\geq$ int und $\leq$ long long
long long	8 Byte	-2^{63} bis $2^{63}-1$	$\geq$ long
unsinged long long	8 Byte	0 bis $2^{64}-1$	$\geq$ long
float	4 Byte	1.2e-38 bis 3.4e38	$\leq$ double
double	8 Byte	2.2e-308 bis 1.8e308	$\geq$ float und $\leq$ long double
long double	12 Byte	...	$\geq$ double
void	0 Byte	entspricht der leeren Menge	

siehe C++ In 21 Tagen u. C++-Standard)

Deklaration von Variablen

Syntax

```
Typ Bezeichner;
Typ Bezeichner1, Bezeichner2;
Typ Bezeichner = Wert;
Typ Bez1 = Wert1, Bez2 = Wert2; // VORSICHT (keine extra Zuweisung)
Typ Bezeichner(Wert);           // analog zu Type Bezeichner = Wert;
```

Beispiel

```
int i=5, j=3*4+8;
```

Initialisierung mit Werten

- Implizite Typumwandlung möglich
- Bsp.: `int i = 5.2;` (Informationsverlust!)
- Auf der rechten Seite darf ein beliebiger kompatibler Ausdruck stehen (dazu später mehr)

Abgeleitete Typen

- Zusätzlich zu den Standardtypen werden von C++ noch jeweils weitere 'Meta-Typen' bereitgestellt:
- ist `T` ein gültiger Typ, so:
  - Ist auch `T[]` ein gültiger Typ (Arrays)
    - und damit auch `T[][]` usw.
  - Ist auch `T*` ein gültiger Typ (Pointer/Zeiger)
    - und damit auch `T**` usw..
  - Ist auch `T&` ein gültiger Typ (Referenzen)
    - Ausnahme hier: Funktioniert nicht rekursiv
  - Seit C++11: zusätzlich `T&&` (sog. RValue Referenz)
    - wir später behandelt!
  - Später: const, volatile, ...

Arrays: Deklarationssyntax

- Ein Array ist ein zusammenhängender Speicherbereich, der eine feste Anzahl von Variablen eines Typs enthält
- Deklaration: `Typ Bezeichner[Anzahl];`
- Anmerkung: in ...

```
int array1[10];
int array2[11];
```

- ... haben `array1` und `array2` *unterschiedliche* Typen

Beispiele

Uninitialisiert:

```
int myArray[5];
```

Oder direkt initialisiert:

```
int myArray[3] = {1, 2, 3};
```

Mehrdimensionale Arrays

```
int myMatrix[4][4];
int myMatrix[3][2] = { {1, 2}, {3, 4}, {5, 6} };
```

Spezialfall

```
int myArray[3] = {0}; // Sonderfall
```

Mit Klassen

```
QWidget widgets[10];
```

Arrays - Eigenschaften

- Indizes beginnen bei 0
- Kein Index-Check!
  - Bei falschen Indizes: Undefiniertes Verhalten. Wenn `SEGFAULT` passiert hat man Glück
- Keine direkte Zuweisung möglich!

Beispiele

```
#include <iostream>

int main(){
  int a[4] = {1, 2, 3, 4};
  int b[4] = {0, 0, 0, 0};
  int c[2] = {7, 8};

  // b = a; -> ungueltig!

  for(int i=0;i<6;i++) {
    b[i] = a[i]; // hmm, irgendwas ist hier falsch
  }

  std::cout << c[0] << std::endl; // 1
  std::cout << c[1] << std::endl; // 2
}
```

Arrays - Initialisierungsbeispiele

Beispiele

```
// Ohne Laengenangabe
int a[] = {1, 2, 3}; // Typ ist int[3]

// im Initialisierungsausdruck koennen auch nur die ersten Elemente
// definiert werden, der Rest bleibt uninitialisiert
int b[10] = { 1, 2, 3, 4, 5 };

// mehrdimensionale Arrays (Laengenangaben von aussen nach innen)
int c[3][2]= {{1, 1}, {2, 2}, {3, 3}};  // 3 mal 2 elemente

// fuer die erste Klammer kann die Laengenangabe weggelassen werden
int d[][2]= {{1, 1}, {2, 2}, {3, 3}};

// hier werden immer nur die ersten 2 Elemente der Einzelarrays initialisiert
int e[][10]= {{1, 1}, {2, 2}, {3, 3}};
```

Datenlayout von mehrdimensionalen Arrays

- Die gesamten Daten liegen kontinuirlich im Speicher
- Auch die einzelnen *inneren* Arrays liegen kontinuirlich im Speicher

Mehrdimensionales Array - Speicherlayout

```
int a[3][2] = {{1,2},
               {3,4},
               {5,6}};

int *b = (int*)a;

for(int i = 0; i < 6; ++i){
  std::cout << b[i] << "  ";
}
std::cout << std::endl;
```

Ausgabe

```
1 2 3 4 5 6
```

Arrays und `sizeof`

- `sizeof` funktioniert auch mit Arrays ...
- ... allerdings **mit Eingeschränkungen**

Hier funktioniert `sizeof` wie erwartet ...

```
int i[5];
std::cout << sizeof(i) << std::endl;
```

- Liefert auf System mit 32-bit `int` 20 zurück

Größen von Variablen - sizeof

... aber hier nicht

```
#include <iostream>
void print_array(int a[]) {
  std::cout << "array size: " << sizeof(a) << std::endl;
  int arrayLen = sizeof(a) / sizeof(a[0]);
  std::cout << "array length: " << arrayLen << std::endl;
  for (unsigned int i = 0; i < arrayLen; ++i) {  std::cout << a[i] << " "; }
  std::cout << std::endl;
}

int main() {
  int i[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  print_array(i);
}
```

Ausgabe

```
array size:  4
array length:  1
1
```

Erklärung(sversuch)

- Größeninformationen werden nicht implizit mitgeschleppt
- In einem Funktionsinterface ist der Typ `int []` ein Alias für "Zeiger auf `int`" (also `int*`)
- und da (eindimensionale) Arrays (also `int i[] = {...};`) implizit in einen `int*` umgewandelt werden können ...
- .. beschwert sich der Compiler nicht!

Erklärung(sversuch)

- Größeninformationen werden nicht implizit mitgeschleppt
- In einem Funktionsinterface ist der Typ `int []` ein Alias für "Zeiger auf `int`" (also `int*`)
- und da (eindimensionale) Arrays (also `int i[] = {...};`) implizit in einen `int*` umgewandelt werden können ...
- .. beschwert sich der Compiler nicht!

- $\Rightarrow$ **Arrays sind böse!**

Auch nicht mit expliziter Größenangabe

```
#include <iostream>
#include <typeinfo>

void f(int a[5]) {
    std::cout << __FUNCTION__ << ":  "<< typeid(a).name() << std::endl;
    std::cout << __FUNCTION__ << ":  " << sizeof(a) << " " << sizeof(a[0]) << std::endl;
}
int main() {
    int a[5];
    std::cout << __FUNCTION__ << ":  "<< typeid(a).name() << std::endl;
    std::cout << __FUNCTION__ << ":  "<< sizeof(a) << " " << sizeof(a[0]) << std::endl;

    f(a);
}
```

Auch nicht mit expliziter Größenangabe

```
#include <iostream>
#include <typeinfo>

void f(int a[5]) {
    std::cout << __FUNCTION__ << ":  "<< typeid(a).name() << std::endl;
    std::cout << __FUNCTION__ << ":  " << sizeof(a) << " " << sizeof(a[0]) << std::endl;
}
int main() {
    int a[5];
    std::cout << __FUNCTION__ << ":  "<< typeid(a).name() << std::endl;
    std::cout << __FUNCTION__ << ":  "<< sizeof(a) << " " << sizeof(a[0]) << std::endl;

    f(a);
}
```

Ausgabe

```
main:  A5_i
main:  20 4
f:  Pi
f:  4 4
```

Arrays ..

- **Was soll das ganze dann eigentlich?**

Arrays ..

- **Was soll das ganze dann eigentlich?**
- Ein Arraytyp in einem Funktionsinterface ist sowas wie ein Tip für den Benutzer

```
/// Hint fuer den Benutzer, dass 3D 'Vektoren' erwartet werden
void cross_product(float v1[3], float v2[3], float result[3]){
  ...
}
/// syntaktisch genau das gleiche wie
void cross_product(float *v1, float *v2, float *result){
  ...
}
```

- Und nochmal: **Keine Index-Checks**!
- $\Rightarrow$ Lieber Container aus der STL benutzen (z.B. `std::vector<T>`)
- Es sei denn, man weiß, was man tut – oder es gibt gute Gründe dagegen
- Zur STL (viel) später (viel) mehr

Pointer sind Adressen mit Typ

- Ein Pointer ist eine Variable, die eine Speicheradresse enthält
- In C++ ist jeder Pointer i.d.R auf den Inhalt des Speichers, auf den er zeigt "getypt"
- Pointer können auch auf ungültige Speicheradressen zeigen
  - Definierte ungültige Speicheradresse ist 0 (`NULL`)
  - Diese wird niemals bei Speicherreservierung u. ähnlichem verwendet!

Deklaration

- Eine Pointervariable wird mit einem * markiert:
- Uninitialisiert: `Typ *Bezeichner;`
- Initialisiert: `Typ *Bezeichner=Adresse;`
- Auch Pointer auf Pointer (usw.) möglich: `int **p=0;`
  - Faustregel: Pointer *immer* initialisieren. Entweder mit 0 oder mit einer *gültigen* Speicheradresse

Beispiel

```
int i = 5;  // kein Pointer
int *a;     // uninitialisierter Pointer (dangling Pointer) -> boese!
int *b = 0; // initialisiert
```

Warum Pointer - Speicher direkt adressieren

- Pointer sind gut, um z.B. Speicher direkt zu adressieren
  - Nützlich bei Systemprogrammierung, da viele "Low-Level" Interaktionsmöglichkeiten durch spezielle Speicheradressen gegeben sind
    - Printerport ist gemapped auf Adresse 0x378
    - Interrupttabelle liegt an bestimmter Adresse im Speicher
    - DMA-Controller schieben Werte an bestimmte Stellen im Speicher
  - Betriebssystem verwaltet Speicherblöcke über deren Adressen
    - Auf modernen Systemen ist dies eine grobe Vereinfachung, wegen virtuellem Speicher, aber es stimmt aus Sicht der Applikation

Warum Pointer - Call by Reference in C

- In C wurden Pointer of eingesetzt, um "Call by Reference" zu implementieren.

```
struct VGAImage {
   unsigned char data[640*480];
};
// image (ca 1/3 MB) wird bei jedem Aufruf kopiert
void inspect_wrong(VGAImage image){ /* image wird nur gelesen! */ }

// nur der Zeiger ( 4 Byte) muss kopiert werden
void inspect_better(VGAImage *image) { /* image wird nur gelesen! */ }
```

- Auf diese Art und Weise wird nur Adresse des Bildes an `inspect_better(..)` übergeben, aber nicht das ganze Objekt kopiert
- In C++: I.d.R. mittels Referenzen (später)

Beispiel für z.B. Listen

- Verkettete Liste:

Vorausblick: Strukturen

```
struct Node{
   int elem;
   Node next;   // so geht es nicht, denn so laesst size sizeof(Node) nicht bestimmen
                // in Java wuerde das gehen, da alle Typen eigentlich Referenzen sind
};

struct Node{
   int elem;
   Node *next;  // so geht es: sizeof(Node) ist sizeof(int) + sizeof(Node*)
};
```

Stack vs. Heap

```
Node node1;               // node1 liegt auf den Stack
Node *node2 = new Node;   // (*node2) liegt auf dem Heap und der Pointer
                          // node2 zeigt auf die Adresse im Heap
```

Pointer

- Zunächst ergeben sich zwei Fragen:
  - Wie erhält man eine sinvolle Speicheradresse?
  - Wie kommt man an den Wert, auf den ein Pointer zeigt?

Wie kommt man an die Adressen von Speicher?

- Direkt: `char *i = (char*)0x17755U;`
- Die Speicheradresse einer Variable (allgemein: einer r-value oder l-value (später)) erhält man durch den sog. "Adress"-Operator &

Beispiel

```
int i = 5;      // kein Pointer
int *pi = &i;   // pi enthaelt die Adresse von i
```

- Aber auch Array-Variablen lassen sich implizit in einen Pointer auf das erste Element umwandeln:

Beispiel

```
int array[4]={1,2,3,4};
int *pa = array;
```

- Arrays sind nur "ganz dünne" Abstraktion um Pointer

Speicher auf dem Heap mittels `new` und `new []` (light)

- Dynamischer Speicher muss mit `new` auf den Heap alloziert werden
  - *dynamisch* heißt: Die Länge ist nicht zur Compile-Time bekannt
  - Bei Arrays wie `int myArray[11];` muss die Array-Länge immer konstant sein

Speicher auf dem Heap mittels `new` und `new []` (light)

- Dynamischer Speicher muss mit `new` auf den Heap alloziert werden
  - *dynamisch* heißt: Die Länge ist nicht zur Compile-Time bekannt
  - Bei Arrays wie `int myArray[11];` muss die Array-Länge immer konstant sein
- 2 Auspägungen von `new`
  - `new` alloziert ein Objekt eines Typs auf dem Heap
  - `new []` alloziert n Objekte eines Type auf dem Heap
- beide Versionen von `new` geben Pointer (Adresse) auf das erste Element zurück

Speicher auf dem Heap mittels `new` und `new []` (light)

- Dynamischer Speicher muss mit `new` auf den Heap alloziert werden
  - *dynamisch* heißt: Die Länge ist nicht zur Compile-Time bekannt
  - Bei Arrays wie `int myArray[11];` muss die Array-Länge immer konstant sein
- 2 Auspägungen von `new`
  - `new` alloziert ein Objekt eines Typs auf dem Heap
  - `new []` alloziert n Objekte eines Type auf dem Heap
- beide Versionen von `new` geben Pointer (Adresse) auf das erste Element zurück

Sehr wichtig

- Manuell auf dem Heap allozierter Speicher muss explizit freigegeben werden, sonst: Speicherleck!
- `delete` bzw. `delete []`

Auf was zeigt ein Pointer?

- An das "Ziel" eines Pointers kommt man mit dem sog. "Dereferenz"-Operator *
- Achtung: Nicht durcheinanderkommen mit dem * welches eine Pointer-Variable signalisiert!

Beispiel

```
int i = 5;      // kein Pointer
int *pi = &i;   // pi enthaelt die Adresse von i

...

int j = *pi;    // pi wird dereferenziert,
                // und Ergebnis wird j zugewiesen
```

Pointer und Strukturen/Klassen (Vorgriff)

- Der Operator . kann benutzt werden, um auf Elemente eines Objekts zuzugreifen
- Wenn man statt Objekt-Referenz Pointer auf Objekt hat, benutzt man den Operator ->

```
struct X {
   int value;
};

int main() {
   X x;
   x.value = 0;

   X *px = &x;
   px->value = 7;

   // alternative mittels dereferenz
   (*px).value = 22;
}
```

Zeigerarithmetik

- Da Pointer eigentlich nur Adressen (Zahlen) sind, ist natuerlich auch Arithmetik mit diesen möglich

```
int numbers[2] = {1, 2};

int *p1 = numbers;
int *p2 = p1+1;
```

- Welche Adresse enthält nun p2?
  - Bei Zeigerarithmetik mit einem X-Pointer ist die Inkrementeinheit immer `sizeof(X)`
  - also p2 zeigt auf die 2 im Array `numbers`

Zugriff auf die "hinteren" Elemente

- Array und Pointer Elemented können mit dem Index-Operator ([]) erreicht werden

Beispiele

```
// bei arrays
int a[3] = { 1, 2, 3};
a[0] = 3; a[1] = 4; a[2] = 5;

// bei pointern gehts genau so
int *p = new int[3];

// p zeigt auf mindestens genug Speicher fuer 3 ints
// der Speicher ist aber nicht initializiert (Speicherinhalt ist unbestimmt)
// Zugriff auf elemente wie bei arrays
p[0] = 3; p[1] = 4; p[2] = 5;

// oder mittels Zeigerarithmetik
(*p+1) = 4;

// oder ... (aber besser nicht)
(*p+2)[-1] = 4;

// Speicher freigeben nicht vergessen (delete [] immer ohne Laengenangabe)
delete [] p;
```

String sind eigentlich auch nur Pointer/Arrays

- Was genau ist eigentlich `"hello world"`?

String sind eigentlich auch nur Pointer/Arrays

- Was genau ist eigentlich `"hello world"`?

- **Antwort:** `"hello world"` ist ein String Literal
- Das lässt sich implizit in einen `const char*` umwandeln
- Ein C-Style-String ist also eine Sequenz von `char`s.
- Wichtig: das Ende des Strings wird immer durch einen unsichtbaren `'0'`-char signalisiert
  - Die Ausgabe gibt alle Zeichen bis zum ersten `'0'`-char aus
  - Um die Länge eines Strings zu ermitteln, muss man die `char`s bis zur ersten 0 Zählen

```
// s zeigt auf "hello world" im Speicher
const char *s = "hello world";

// array basiert geht auch (hier: nicht zwangsweise const, da die chars kopiert werden)
char a[] = "hello world";

// genau genommen ist das eine Kurzform von dem hier
char a[] = { 'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd' , '\0'};
```

C-Strings (ein paar Funktionen)

```
#include <iostream>

int len(const char *s){  // <cstring>: strlen
  int len = 0;
  while(s[len] != '\0') len++;
  return len;
}

void cpy(const char *s, char *d, int len){  // <cstring>: strncpy
  for(int i=0;i<len;++i){
    d[i] = s[i];
  }
  d[len] = '\0';
}

char *cat(const char *a, const char *b){ // <cstring>: strcat
  int la = len(a), lb = len(b);
  char *ab = new char[la+lb+1];
  cpy(a,ab,la);
  cpy(b,ab+la,lb);
  ab[la+lb] = '\0';
  return ab;
}

int main(){
  std::cout << cat(cat("hello"," "),"world") << std::endl;
}
```

Referenzen

- Eine Referenz entspricht zunächst einem alternativen Namen für eine Variable

Syntax

```
TypX &Bezeichner = WasAnderes;
```

- `WasAnderes` muss den Typ `TypX` haben
- Referenzen
  - Müssen immer direkt initialisiert werden
  - Können nicht *umgebogen* werden
  - Es ist nicht möglich, Arrays von Referenzen zu erstellen

Referenzen - Beispiel

Beispiel

```
int main(){
  int a=5;
  int &refOfa = a;
  int copyOfa = a;
  a = 7;

  std::cout << "a is:" << a << endl;
  std::cout << "ref(a) is:"<< refOfa << std::endl;
  std::cout << "copy(a) is:" << copyOfa << endl;

  return 0;
}
```

- Erstellen von alternativen Bezeichnern in einem Scope ist nicht immer vorteilhaft
- Teilweise kann damit aber viel Schreibarbeit gespart werden
  - Z.B. in verschachtelten schleifen
- Kann auch Geschwindigkeitsvorteile bringen

Referenzen - Hauptvorteil: Call by Reference

- Sehr nützlich für Parameterübergabe ohne Kopien zu erstellen (fast immer schneller als Kopien)
  - "Call by Reference"

Kopien vermeiden mit Referenzen

```
void inspectObject (const HugeClassType &object) {
  // inspiziere Objekt und mach was mit dem Resultat,
  // keine Kopie noetig
}

int main() {
  HugeClassType meinObjekt;
  inspectObject(meinObjekt);
}
```

- Sehr ähnlich zum Pointer-Konzept, jedoch sehr viel weniger fehlerträchtig in den meisten Fällen

Referenzen - Call by Reference cntd..

- Auch nützlich, um eine Funktion ein Objekt modifizieren zu lassen
- Gehen wir auch später nochmal genauer drauf ein (Funktionen)

Objekte im Caller-Scope modifizieren

```
void modifyOriginalObject (MyType &object) {
  // object "referenziert" jetzt meinObjekt in main()
}

int main() {
  MyType meinObjekt;
  modifyObject(meinObjekt);
  // meinObjekt ist nun veraendert worden
}
```

Referenzen - Konstanten

- Nicht `const`-Referenzen können nicht mit `const`-Werten initialisiert werden

```
int &i = 5;
```

- 5 ist eine Konstante. Zu Konstanten später mehr.
- Es ist aber möglich eine `const` Referenz mit einer Konstante zu initialisieren:

```
const int &i = 5;
```

- Auch dazu später mehr.

Referenzen - Termporäre Objekte

- Etwas ähnliches gilt auch fuer temporäre Objekte. Das hier geht nicht:

```
int foo() { return 5; }

int main(){
  int &a = foo();  // Fehler
}
```

- Das hier geht aber:

```
int foo() { return 5; }

int main(){
  const int &a = foo();  // Geht
}
```

Referenzen - Arrays

- Es ist nicht möglich Arrays von Referenzen zu erstellen

```
int ar[8];
int &aref[8] = ar; // waere nett, geht aber nicht
```

- Es ist aber möglich, *Elemente* eines Arrays zu referenzieren

```
int ar[3] = { 1, 2, 3 };
int &aref = ar[1]; // Geht

aref = 10;
// Array – Inhalt nun { 1, 10, 3 }
```

Referenzen - Pointer

- Es ist auch möglich das Ziel von Pointern zu referenzieren

Referenz auf Pointer-Ziel

```
int array[5] = { 1, 2, 3, 4, 5};

int *p = array+2;  // zeigt nun auf die '3'
int &r3 = *p;       // referenziert nun die '3'
r3 = 7;

int &r2 = array[2]; // referenziert die '2'
r2 = 33;

// array Inhalt nun: { 1,33,7,4,5 };
```