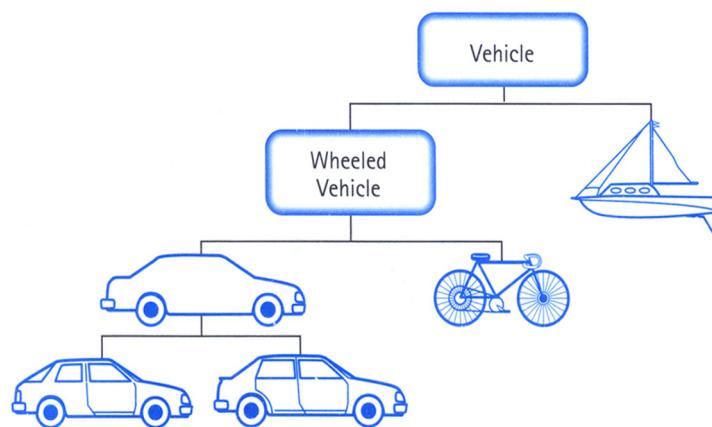


Objekt-Orientierte Programmierung

VORLESUNG 2





Modularisierung



Module

- man kann einfach den Quellcode in mehreren Dateien aufteilen
- logische Groupierung zusammengehörender Funktionen, Klassen
- jede Datei bildet dadurch ein Modul
- Ein **Modul** ist eine Sammlung von Funktionen und Variablen, die eine klar definierte Funktionalität erfüllen.
- Ziele:
 - Trennung der Schnittstelle von der Implementierung
 - Verstecken die Implementierungsdetails



Deklaration und Definition

- Eine **Deklaration** ist eine Anweisung, die einen Namen in einen Gültigkeitsbereich (scope) einführt und
 - einen **Typ** für das benannte Objekt angibt
 - eine Funktion ist vom Typ ihrer Rückgabe
 - optional einen **Initialisierer** angibt
- D.h.: eine Deklaration ist die **Schnittstelle** zur Verwendung des deklarierten
 - meist in Headerdateien zu finden
 - Deklarationen können sich beliebig oft wiederholen
- Eine Deklaration, die das deklarierte Element auch **vollständig spezifiziert**, nennt man **Definition**.
 - die Definition reserviert im Unterschied zur Deklaration Speicher
 - die Definition ist immer auch eine Deklaration
- D.h.: die Definition ist die **Implementierung**, durch welche das benannte Element das macht, wofür es gedacht ist.
- **Jeder Name muss nur einmal definiert sein! (one definition rule)**



Deklaration und Definition

- durch die Unterscheidung Deklaration-Definition lässt sich das Programm auch in **mehrere Teile** (Dateien) zerlegen, die voneinander **getrennt kompiliert** werden
- jeder Teil muss (nur) alle Deklarationen kennen
 - die konsistent sein müssen
- der Linker verknüpft den Objektcode
- das Schlüsselwort **extern**
 - gibt an, dass die nachfolgende Deklaration keine Definition ist

Deklarationen:

```
double sqrt(double d);
```

```
double sqrt(double d);
```

```
extern int x;
```

```
extern int x;
```

Definitionen:

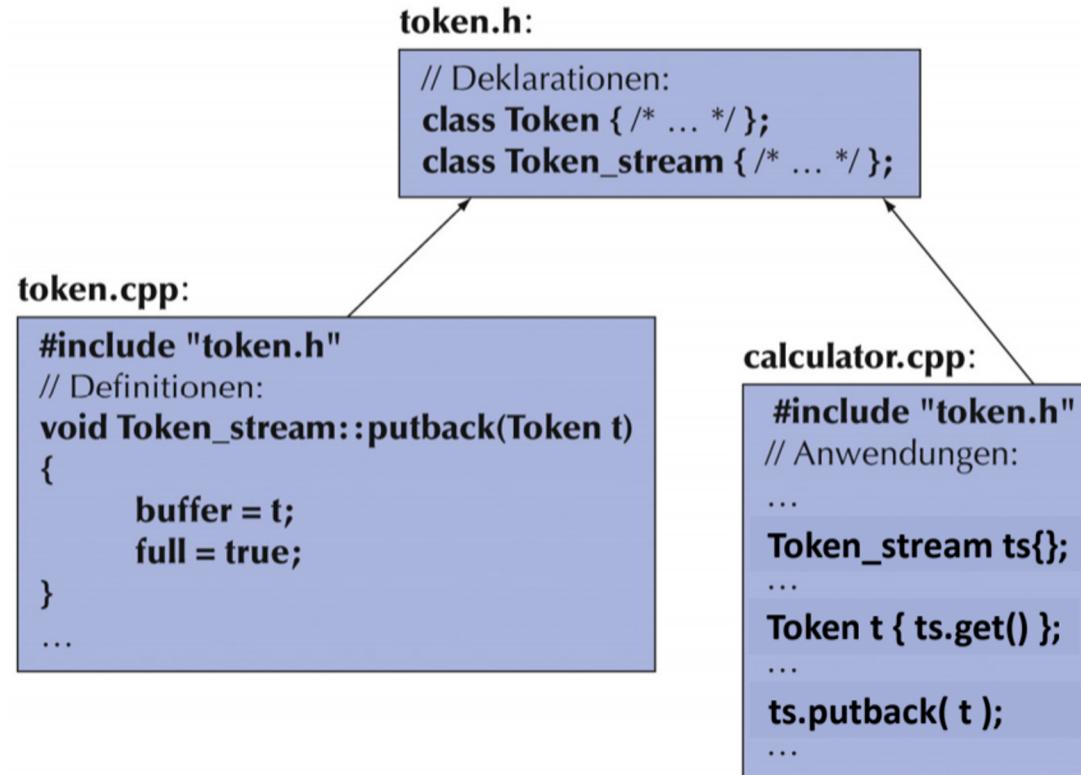
```
double sqrt(double d)
{
    // berechnet die
    // Quadratwurzel
    // von d
}
```

```
int x {7};
```



Deklaration und Definition, Headerdateien

Header werden mit der `#include` Direktive vom Präprozessor in eine Quellcode-Datei einkopiert.





Headerdateien. Libraries

- Funktionsprototypen (Funktionsdeklarationen) werden in einer separaten Datei gruppiert (Headerdatei)
- eine **Bibliothek** besteht aus einer Reihe von Funktionen, die für die Wiederverwendung von anderer Programme implementiert wurde
- Bibliotheken werden im Allgemeinen so verteilt:
 - eine Headerdatei (.h) mit den Funktionsprototypen und
 - eine Binärdatei (.dll oder .lib), die die kompilierten Implementierungen enthält
- der Quellcode (cpp) muss nicht unbedingt verteilt werden



Headerdateien. Libraries

- die Bibliotheksbenutzer benötigen nur die Funktionsprototypen (aus der Headerdatei), nicht die Implementierung
- **die Funktionsspezifikation ist von der Implementierung getrennt**
- der Linker stellt alles zusammen
- **die statische Verknüpfung (linking)** erfolgt zur Kompilierungszeit und die **.lib** ist vollständig in der ausführbaren Datei enthalten
 - verursacht eine Vergrösserung der resultierenden ausführbaren Datei
- **die dynamische Verknüpfung** (DLL-Dateien) enthält nur die Informationen, die zur Laufzeit zum Suchen und Laden der DLL benötigt werden, die ein Datenelement oder eine Funktion enthält.



Präprozessordirektiven

- zeilen im Code, denen ein Hash-Zeichen (#) vorangestellt ist, werden vor der Kompilierung vom Präprozessor ausgeführt

- `#include headerdatei`
 - wenn die Headerdatei in `<>` eingeschlossen ist, wird die Datei in den Systemverzeichnissen durchsucht
 - wenn der Header in `""` eingeschlossen ist, wird die Datei zuerst im aktuellen Verzeichnis und dann in den Systemverzeichnissen durchsucht

- `#define identifier replacement`
 - jedes Auftreten des `identifier` im Code wird durch `replacement` ersetzt
 - zB ein Mechanismus für Konstante (nicht eine gute Idee in C++)



Präprozessordirektiven

- `#ifdef Macro, ..., #endif`
 - der Codeabschnitt zwischen diese beiden Anweisungen wird nur kompiliert, wenn das angegebene Macro definiert wurde
- `#ifndef Macro, ..., #endif`
 - der Codeabschnitt zwischen Diese beiden Anweisungen wird nur kompiliert, wenn das angegebene Macro **nicht** definiert wurde
- `#ifndef #define` und `#endif` können als **Include-Guards** verwendet werden
- Include-Guards werden verwendet, um **mehrfach includes** zu vermeiden, wenn wir die `# include`-Direktive verwenden
 - **Mehrfach includes** führen zu Kompilationsfehler (One Definition Rule)
 - `#pragma once`



Modulare Programme

- der Code eines C ++-Programms ist in mehrere Quelldateien aufgeteilt:
 - h-Dateien - enthalten die Funktionsdeklarationen (die Schnittstellen)
 - cpp-Dateien - enthalten die Funktionsimplementierungen
- Vorteil: die .cpp-Dateien können separat kompiliert werden
 - gilt auch für Testing
- immer wenn eine Headerdatei geändert wird, müssen alle Dateien auf sie verweisen (direkt oder indirekt) neu kompiliert werden
- Die Headerdatei ist ein Vertrag zwischen dem Entwickler und dem Anwender einer Bibliothek, die die Datenstrukturen beschreibt und gibt die Argumente und Rückgabewerte für Funktionsaufrufe an
- Der Compiler setzt den Vertrag durch, indem er die Deklarationen für alle Strukturen und Funktionen benötigt, bevor sie verwendet werden
 - Aus diesem Grund muss die Headerdatei eingebunden sein.



Testen



Test functions

- `#include <cassert>`
`void assert (int expr);`
- wenn der Ausdruck auf 0 ausgewertet wird, wird eine Nachricht auf das Standardfehler geschrieben und **die Ausführung wird gestoppt**
- die Nachricht enthält: den Ausdruck, dessen Zusicherung fehlgeschlagen ist, den Namen der Quelldatei und die Zeilennummer, in der sie aufgetreten ist.

```
• int main() {  
    assert(2+2==5);  
}
```

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)  
► clang++-7 -pthread -o main main.cpp  
► ./main  
main: main.cpp:44: int main(): Assertion `2+2==5' failed.  
exited, aborted
```



variablen und Konstanten



Globale Variablen

- Variablen, die außerhalb einer Funktion definiert sind, sind globale Variablen
- kann von jeder Funktion aufgerufen werden
- der Scope ist die gesamte Anwendung
- globale Variable sind **fast immer** eine schlechte Idee



Konstanten

- Reserviertes Wort `const` ändert Deklaration
- `const`-Variablen sind nur lesbar
- Initialisierung erfolgt bei Deklaration
- Vorzuziehen gegenüber `#define`, weil vom Compiler verwaltet
 - Definition von lokalen Konstanten um **type-safety** und **scope** sicherzustellen
 - Zeiger auf Konstanten möglich

```
const int k = 42;

char* const s1 = "Test1";
const char* s2 = "Test2";
const char* const s3 = "Test3";

k = 4; // Fehler: k ist const
s1 = "New test"; // Fehler: Zeiger ist const
*s1 = 'P'; // okay, Zeichen von s1 sind nicht const
s2 = "New test"; // okay, Zeiger selbst ist nicht const
*s2 = 'P'; // Fehler: Zeichen von s2 sind const
```



Funktionen



Funktionen

- eine Funktion ist eine Gruppe von Anweisungen, die zusammen eine bestimmte Aufgabe erfüllen
- Deklaration (Function prototype)

```
<result type> name (<parameter list>);
```

```
/*
```

Computes the greatest common divisor of two positive integers.

Input: a, b integers , a, b > 0

Output: returns the the greatest common divisor of a and b.

```
*/
```

```
int gcd ( int a , int b ) ;
```



Überladen von Funktionen

- gleicher Funktionsname für unterschiedliche Implementierungen
- überladene Funktionen werden unterschieden durch:
 - Anzahl der Parameter
 - Typ der Parameter
 - Reihenfolge der Parametertypen
- Rückgabewert kann ignoriert werden

```
void print();  
  
void print(int, char*);  
  
int print(float);  
  
int print(); //error
```



Default-Parameter

- Funktionsparameter können einen Defaultwert besitzen
- Wird verwendet, wenn der Parameter im Aufruf fehlt
- Nur am Ende der Parameterliste erlaubt

```
void print(char* string, int n = 1);  
  
print("Test", 0);  
  
print("Test"); // äquivalent zu print("Test", 1)
```



Achtung!

Überladen und Default-Parameter können Mehrdeutigkeit verursachen

```
void print(char* string) {  
    std::cout << string;  
}  
  
void print(char* string, int n = 1)  
{  
    std::cout << string << std::endl;  
    std::cout << n;  
}  
  
print("Test");
```





Parameter - Pass by value

- `void byValue (int a)`
- der Standard-Parameterübergabemechanismus in C++
- beim Funktionsaufruf erstellt C ++ eine Kopie des Parameter
 - auf dem **Stack**
- Veränderungen an der Variable gehen mit dem Funktionsende verloren



Parameter - Pass by reference

- `void byRef (int& a)`
- die Speicheradresse des Parameters wird an die Funktion übergeben
- die vorgenommenen Änderungen an der Variable passieren auf dem “Original” des Aufrufers
- Nützlich für Arrays und größeren Datenstrukturen, sie "als Referenz" zu übergeben (vermeidet Kopie: Zeit und Speicherplatz)
- der Adress-Operator &



Beispiel

```
struct Vector
{
    int arr[5];
    int length;
};

void byValue(int a) {
    a = a + 1;
    std::cout << "Address of a in function: " <<
    &a << std::endl;
}

void byRef(int& a) {
    a = a + 1;
}

void passArray(int x[]) {
    x[0] = 1000;
    * (x + 3) = 4000; // <=> x[3]
}

void passStruct1(Vector v) {
    v.arr[0] = 1000;
}

void passStruct2(Vector* v) {
    v->arr[0] = 1000;
}

void passStruct3(Vector& v) {
    v.arr[0] = 1000;
}
```



Beispiel

```

int main() {
    int a = 10;

    std::cout << "Address of a: " << &a << std::endl;
    byValue(a);
    std::cout << "Value remains unchanged: a = " << a
    << std::endl;

    byRef(a);
    std::cout << "Value has changed: a = " << a <<
    std::endl;

    int arr[5] = { 1, 2, 3, 4, 5 }; // arr is a pointer
    to the first element of the array arr
    passArray(arr);

    std::cout << arr[0] << " " << arr[3] << std::endl;
}

Vector v;
v.arr[0] = 1;
v.arr[1] = 2;
v.length = 2;
passStruct1(v);
std::cout << "after passStruct1 v.arr[0] = "
<< v.arr[0] << std::endl;

passStruct2(&v);
std::cout << "after passStruct2 v.arr[0] = "
<< v.arr[0] << std::endl;

passStruct3(v);
std::cout << "after passStruct3 v.arr[0] = "
<< v.arr[0] << std::endl;
}

```



Referenzvariablen

- Adress-Operator & in Variablen-deklaration

```
type &reference_variable = variable_of_type;
```

- kein eigenständiger Wert
- **Alias/Verweis** für eine andere Variable
- muss bei Deklaration initialisiert werden
- Operationen auf Referenzvariablen verändern die referenzierte Variable

```
int x = 5; // Variable  
int &rx = x; // Referenz auf x  
x = 6; // x==6 und rx==6
```



Referenzvariablen

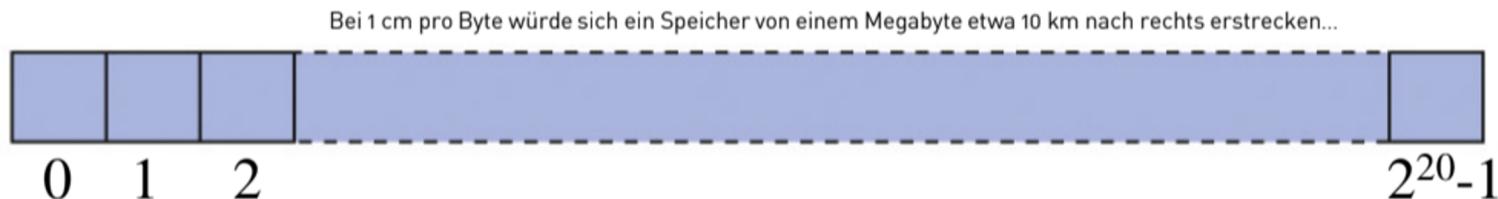
- ähnlich zu Zeigern mit impliziter Dereferenzierung
- Rückgabe von Referenzen ebenfalls möglich
- Funktion liefert eine Speicheradresse (**Ivalue**), nicht einen Wert

```
int global = 0; // globale Variable
int& func() {
    return global; // Rückgabe: Referenz auf global
}
int main() {
    int x; x = func() + 1; // x = global + 1;
    func() = x; // global = x;
}
```



Speicher, Adressen und Zeiger

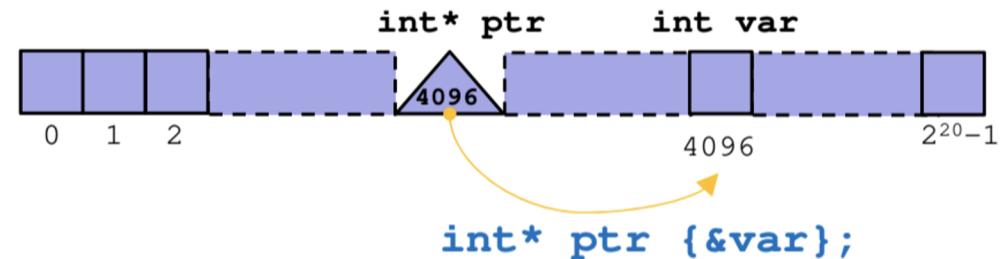
- der Arbeitsspeicher des Computers besteht aus **einer Folge von Bytes**.
 - die Bytes sind durchnummeriert, beginnend bei 0.
 - eine Nummer (Position) eines Bytes im Speicher nennen wir **Adresse**.
- Alles, was wir im Speicher ablegen, hat eine Adresse
- `int var {17};`
- diese Anweisung reserviert im Arbeitsspeicher einen "int-großen" Speicherblock für den Namen `var` und schreibt dort ein Bitmuster, welches den Wert 17 darstellt.





Zeiger

- Zeiger sind Variablen, die Speicheradressen speichern
- Sie ermöglichen es uns, Daten flexibler zu bearbeiten
- Dereferencing bedeutet, auf den Wert zuzugreifen, auf den ein Zeiger zeigt
- Dereferenzierungsoperator: `*`
- Adressoperator: `&`
- **Null Pointer** ist ein Zeiger
 - der auf 0 gesetzt ist
 - ungültiger Zeiger
 - `nullptr` in C++
- Zeiger werden auf 0 (oder NULL) gesetzt, um anzugeben, dass sie derzeit nicht gültig sind
- Man muss prüfen, ob ein Zeiger null ist, bevor wir den Wert nehmen!





Beispiele

Zeigerwerte sind zwar ganze Zahlen, aber nicht vom int Typ

```
double d {0.5}; double* pd {&d}; cout << pd << ' ' << *pd;  
int i {42}; int* pi {&i}; cout << pi << ' ' << *pi;  
  
*pi = 78; cout << pi << ' ' << *pi;  
*pd = 2.718; cout << pd << ' ' << *pd;  
*pd = *pi; cout << pd << ' ' << *pd;
```



Zeiger

C++

```
int* p;
```

```
int x;
p = &x;
```

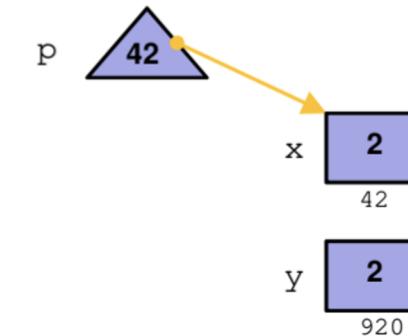
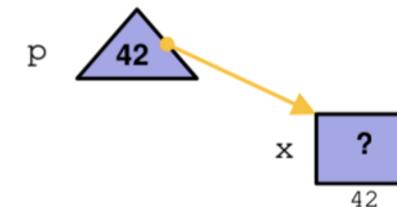
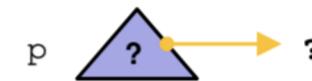
```
x = 2;
int y {*p};
```

Erklärung

Die definierte Variable namens `p` ist ein Zeiger auf `int`. Da nicht initialisiert wurde, ist ihr Wert (d.h. die Adresse) unbestimmt.

Die Adresse der definierten `int`-Variablen namens `x` wird in `p` gespeichert ("`p` zeigt auf `x`").

`p` zeigt noch auf `x`. Die `int`-Variable `y` wird definiert und mit dem Wert 2 (dem Inhalt der Adresse, auf die `p` zeigt) initialisiert.





Zeiger

C++

```
int x {2};
int* p {&x};
*p = -5;
```

```
int x {2};
int* p {&x};
*p += 1;
```

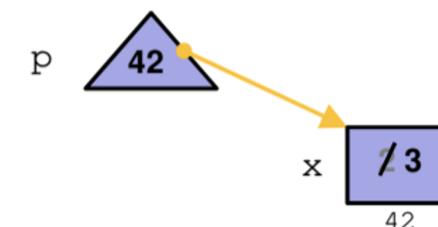
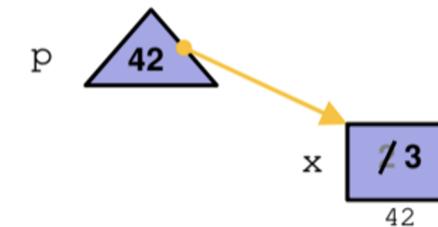
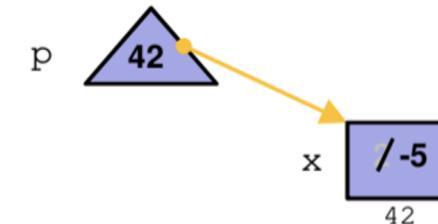
```
int x {2};
int* p {&x};
(*p)++;
```

Erklärung

Zuerst wird die Speicherstelle, auf die `p` zeigt, ermittelt (Adresse von `x`), dann der Wert an dieser Speicherstelle auf `-5` gesetzt.

Inkrementiert den Wert der Variablen `x`.

Inkrementiert den Wert der Variablen `x`. Die Klammern sind wegen des Vorrangs der Operatoren erforderlich.



Zeiger

C++

```
int* pi {};
double* pd {};
```

```
pi = new int[2];
pi++;
```

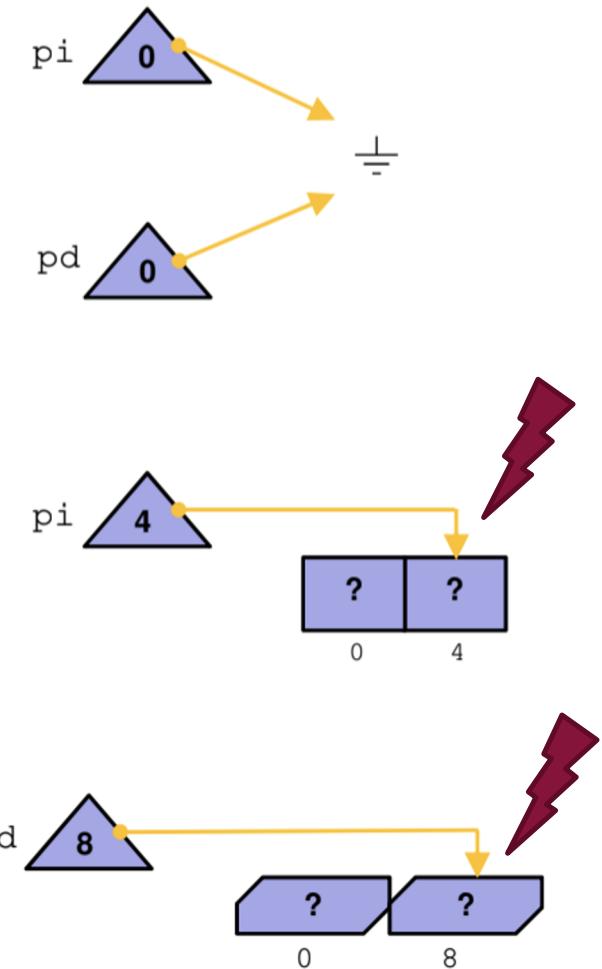
```
pd = new double[2];
pd++;
```

Erklärung

pi ist der Nullzeiger auf int und pd ist der Nullzeiger auf double.

pi zeigt "ein int weiter" in den Speicher (Zeigerarithmetik).

pd zeigt "ein double weiter" in den Speicher (Zeigerarithmetik).





sizeof()

- der `sizeof()` Operator kann auf Typnamen und Ausdrücke angewandt werden
- der `sizeof()` Operator gibt die Anzahl Bytes zurück
 - eine positive ganze Zahl in Vielfachen von `sizeof(char)`
 - `sizeof(char)` ist in C++ per Definition gleich 1.
 - Ein C++ Compiler reserviert für `char` genau ein Byte
- es gibt keine Garantie dafür, dass ein Typ unter jeder Implementierung von C++ dieselbe Größe hat



Type	Size in bits	Format	Value range	
			Approximate	Exact
character	8	signed (one's complement) 🔗		-127 to 127
		signed (two's complement) 🔗		-128 to 127
		unsigned		0 to 255
integral	16	signed (one's complement)	$\pm 3.27 \cdot 10^4$	-32767 to 32767
		signed (two's complement)		-32768 to 32767
		unsigned	$0 \text{ to } 6.55 \cdot 10^4$	0 to 65535
	32	signed (one's complement)	$\pm 2.14 \cdot 10^9$	-2,147,483,647 to 2,147,483,647
		signed (two's complement)		-2,147,483,648 to 2,147,483,647
		unsigned	$0 \text{ to } 4.29 \cdot 10^9$	0 to 4,294,967,295
	64	signed (one's complement)	$\pm 9.22 \cdot 10^{18}$	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807
		signed (two's complement)		-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
		unsigned	$0 \text{ to } 1.84 \cdot 10^{19}$	0 to 18,446,744,073,709,551,615
floating point	32	IEEE-754 🔗	$\pm 3.4 \cdot 10^{\pm 38}$ (~7 digits)	<ul style="list-style-type: none"> min subnormal: $\pm 1.401,298,4 \cdot 10^{-47}$ min normal: $\pm 1.175,494,3 \cdot 10^{-38}$ max: $\pm 3.402,823,4 \cdot 10^{38}$
	64	IEEE-754	$\pm 1.7 \cdot 10^{\pm 308}$ (~15 digits)	<ul style="list-style-type: none"> min subnormal: $\pm 4.940,656,458,412 \cdot 10^{-324}$ min normal: $\pm 2.225,073,858,507,201,4 \cdot 10^{-308}$ max: $\pm 1.797,693,134,862,315,7 \cdot 10^{308}$



Fragen und Antworten
