1.2. Datentypen (std::string)

AUSWEG: Datentyp `std::string` (<string>)

- eine Standardklasse zur Verarbeitung von Strings
- etwa auf dem Niveau von java.lang.String mit der
- Möglichkeit der Initialisierung aus C-Strings
- optimierte Implementation (z.B. kurze Strings im Objekt)

```
std::string vorname = "bjarne";
```

- und einer Vielzahl von Operationen (a la Java):

```
std::string nachname = "Stroustrup";
```

1.2. Datentypen (std::string)

```
std::string name; // noch leer !

vorname[0]='B'; // unchecked !
vorname.at(0) = 'B'; // checked
name = vorname + " " + nachname;
if (name != "")
  std::cout << name << std::endl;
int l = name.length(); // ohne 0-Byte !

"hallo" + ", World\n"; // ERROR
string("hallo") + ", World\n"; // OK

const char* cstring = name.c_str();
```

... Vergleich, Suche, I/O ... ↳ http://de.cppreference.com

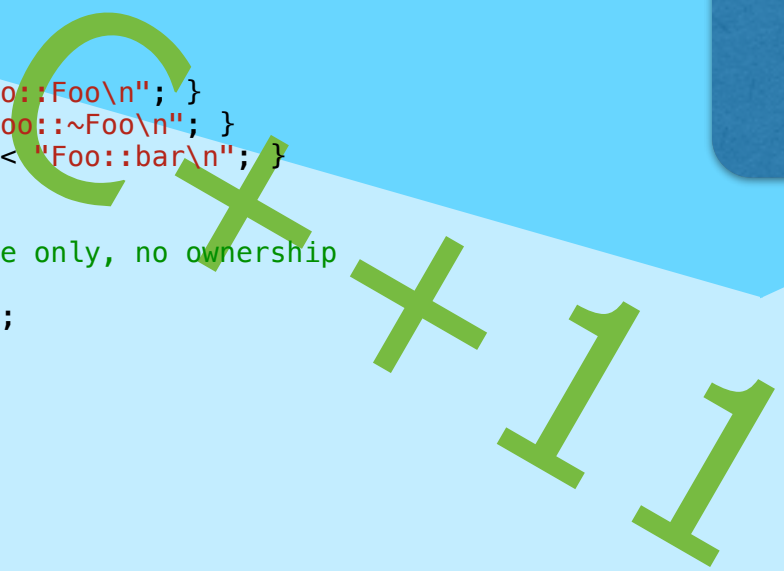
1.2. bessere Zeiger

- C- Zeigertypkonzept (raw pointers) ist fehleranfällig
- es verknüpft zwei sematische Aspekte:
  - Referenzierung und Besitz (ownership)
- einem Zeiger sieht man nicht an ob er nur den Zugang zu einem Objekt herstellt, oder auch für dieses verantwortlich ist
- Abhilfe durch die C++-Standardbibliothek
  - `#include <memory>`
  - 2 Arten von owning pointers in Form von Klassen:
    - `std::unique_ptr<T>` - single ownership
    - `std::shared_ptr<T>` - reference counted multiple ownership
- nutzen Sie in neuem Programmcode bevorzugt diese Zeiger, sie sind bequemer, sicherer und nicht / nur wenig teurer als raw pointers

1.2. ‚schlechte' Zeiger

```
#include <iostream>

struct Foo {
    Foo() { std::cout << "Foo::Foo\n"; }
    ~Foo() { std::cout << "Foo::~Foo\n"; }
    void bar() { std::cout << "Foo::bar\n"; }
};

void f(Foo* foo) // reference only, no ownership
{
    std::cout << "f(Foo*)\n";
}

int main() {
    Foo* p1(new Foo);
    if (p1) p1->bar();

    {
        Foo* p2(p1);
        f(p2);
        delete p2;
    }
    std::cout << "before calling bar again\n";

    // undefined:
    if (p1) p1->bar(); // call on a zombie

}
```

```
Foo::Foo
Foo::bar
f(Foo*)
Foo::~Foo
before calling bar again
Foo::bar
```



1.2. bessere Zeiger unique_ptr

```
#include <iostream>
#include <memory>

struct Foo {
    Foo() { std::cout << "Foo::Foo\n"; }
    ~Foo() { std::cout << "Foo::~Foo\n"; }
    void bar() { std::cout << "Foo::bar\n"; }
};

void f(Foo* foo) // reference only, no ownership
{ std::cout << "f(Foo*)\n"; }

int main() {
    std::unique_ptr<Foo> p1(new Foo);  // p1 owns Foo
    if (p1) p1->bar();

    {   std::unique_ptr<Foo> p2(std::move(p1));  // now p2 owns Foo
        f(p2.get());

        p1 = std::move(p2);  // ownership returns to p1
        std::cout << "destroying p2...\n";
    }
    std::cout << "before calling bar again\n";

    if (p1) p1->bar();

    // Foo instance is destroyed 'automagically' when p1 goes out of scope
}
```

```
Foo::Foo
Foo::bar
f(Foo*)
destroying p2...
before calling bar again
Foo::bar
Foo::~Foo
```

C++11

not copyable !

1.2. bessere Zeiger unique_ptr

```
#include <iostream>
#include <memory>

struct Foo {
    Foo() { std::cout << "Foo::Foo\n"; }
    ~Foo() { std::cout << "Foo::~Foo\n"; }
    void bar() { std::cout << "Foo::bar\n"; }
};

void f(Foo* foo) // reference only, no ownership
{ std::cout << "f(Foo*)\n"; }

int main() {
    std::unique_ptr<Foo> p1(new Foo);  // p1 owns Foo
    if (p1) p1->bar();

    {   std::unique_ptr<Foo> p2(std::move(p1));  // now p2 owns Foo
        f(p2.get());

        // p1 = std::move(p2);
        std::cout << "destroying p2...\n";
        // Foo instance is destroyed 'automagically' when p2 goes out of scope
   }
    std::cout << "before calling bar again\n";

    if (p1) p1->bar();

}
```

```
Foo::Foo
Foo::bar
f(Foo*)
destroying p2...
Foo::~Foo
before calling bar again
```

C++11

not copyable !

1.2. bessere Zeiger shared_ptr

```
#include <iostream>
#include <memory>

struct Foo {
    Foo() { std::cout << "Foo::Foo\n"; }
    ~Foo() { std::cout << "Foo::~Foo\n"; }
    void bar() { std::cout << "Foo::bar\n"; }
};

void f(Foo* foo) // reference only, no ownership
{ std::cout << "f(Foo*)\n"; }

int main() {
    std::shared_ptr<Foo> p1(std::make_shared<Foo>());  // p1 owns Foo
    if (p1) p1->bar();

    {   std::shared_ptr<Foo> p2(p1);  // now p2 owns Foo too !!!!!!
        f(p2.get());

        p1 = p2;  // p1 already owns it
        std::cout << "destroying p2...\n";
    }
    std::cout << "before calling bar again\n";

    if (p1) p1->bar();

    // Foo instance is destroyed 'automagically' when p1 (last owner) goes out of scope
}
```

```
Foo::Foo
Foo::bar
f(Foo*)
destroying p2...
before calling bar again
Foo::bar
Foo::~Foo
```

C++11

copyable !

1.2. bessere Zeiger shared_ptr

```
#include <iostream>
#include <memory>

struct Foo {
    Foo() { std::cout << "Foo::Foo\n"; }
    ~Foo() { std::cout << "Foo::~Foo\n"; }
    void bar() { std::cout << "Foo::bar\n"; }
};

void f(Foo* foo) // reference only, no ownership
{ std::cout << "f(Foo*)\n"; }

int main() {
    std::shared_ptr<Foo> p1(std::make_shared<Foo>());  // p1 owns Foo
    if (p1) p1->bar();

    {   std::shared_ptr<Foo> p2(p1);  // now p2 owns Foo too !!!!!!
        f(p2.get());

        // p1 = p2;
        std::cout << "destroying p2...\n";
    }
    std::cout << "before calling bar again\n";

    if (p1) p1->bar();

    // Foo instance is destroyed 'automagically' when p1 (last owner) goes out of scope
}
```

```
Foo::Foo
Foo::bar
f(Foo*)
destroying p2...
before calling bar again
Foo::bar
Foo::~Foo
```

C++11

copyable !

1.2. Datentypen

Referenztypen:

Eine Neuerung gegenüber C, Aliasnamen für Objekte mit Referenzsemantik ähnlich zur primären Objektsemantik von Java, aber

– für alle Typen (incl. build-in Typen)

– es gibt KEINE ‚Nullreferenz'

– Referenz ohne ownership !

in Anlehnung an die Syntax von Zeigervereinbarungen

```
int i = 42;
// int& ri;
// ERROR: Referenzen MÜSSEN initialisiert werden
int& ri = i; // i alias ri
class X{…} x;  X& y=x; // zwei Namen für ein Objekt
```

1.2. Datentypen

Konstantentypen:

Ein Typ `T` wird durch den Zusatz `const` zu einem Konstantentyp, Objekte solcher Typen sind unveränderlich (per statischer Kontrolle durch den Compiler)

für Argumente von Funktionen bedeutet dies, dass die Funktion

1. die (nachprüfbare) Zusicherung gibt, dieses Argument NICHT zu verändern
2. beim Aufruf für das Argument auch konstante Objekte benutzt werden dürfen (was für non-const nicht erlaubt ist, weil ja die Funktion keine Zusicherung gegeben hat und daher ...)

```
double const pi=3.1415926; double someMathFkt(double);
double const x = someMathFkt(pi);  // call by value !
```

1.2. Datentypen (Konstantentypen)

Konstante Objekte müssen initialisiert werden (weil eine spätere Zuweisung nicht erlaubt ist)

konstante Objekte können auch über Zeiger nicht verändert werden, weil die Adresse einer `T const` Variablen vom Typ `T const *` ist

```
double* dp = &pi; // ERROR
*dp = 33.3;

double const * cdp = &pi;
*cdp = 33.3; // ERROR
```

1.2. Datentypen (Konstantentypen)

bei Zeigern ist wohl zu unterscheiden zwischen der constness des Zeigers selbst

```
int * const constant_pointer = &someint;
```

und der constness des referenzierten Objektes (Feldes)

```
const int * pointer_to_constant;
```

```
const int * const constant_pointer_to_constant = …;
```

1.2. Datentypen (Konstantentypen)

const T vs. T const ?

```
const int c1 = 1;    // not allowed: c1 = 2;

int const c2 = 2;    // not allowed: c2 = 3;

// 'const T' and 'T const' seems to be the same?
// BUT
int i = 0, j = 0;

const int* p1 = &i;
// not allowed:
// *p1 = 3;
// even if ok:
i = 3;

int* const p2 = &i;
// this is allowed:
*p2 = 3;

// p1 and p2 MUST be different!
// indeed: the star in 'const int* p1' binds to p1, thus
// p1 is a pointer to const int
p1=&j; // ok
```

1.2. Datentypen (Konstantentypen)

const T vs. T const ?

```
// p2 is a const pointer to (a modifyable) int
// p2 = &j;  not ok

// RECOMMENDATION: put const AFTER type:

int const *p3 = &i;

// read backwards: p3 is pointer to const int

int* const p4 = &j;

// read backwards: p4 is const pointer to int

int const * const p5 = &j;

// read backwards: p5 is const pointer to const int

// neither: p5 = &i;
// nor:     *p5 = 7;
```