



# Programmieren 3

## C++

### Vorlesung 04: Konstruktoren, Destruktoren

Prof. Dr. Dirk Kutscher  
Dr. Olaf Bergmann

# Wiederholung

# Zeichenketten in C

- In C: ein-dimensionale Arrays aus Buchstaben mit einem Null-Byte am Ende.

```
#include <stdio.h>

int main () {
    const char greeting[]="Moin!";
    printf("Greeting message: %s\n", greeting);
    return 0;
}
```

# Zeichenketten in C

- In C: ein-dimensionale Arrays aus Buchstaben mit einem Null-Byte am Ende.

```
#include <stdio.h>

int main () {
    const char greeting[]="Moin!";
    printf("Greeting message: %s\n", greeting);
    return 0;
}
```


Index	0	1	2	3	4	5
Variable	M	o	i	n	!	\0
Adresse	0x23450	0x23451	0x23452	0x23453	0x23454	0x23455

# Zeichenketten in C

- In C: ein-dimensionale Arrays aus Buchstaben mit einem Null-Byte am Ende.

```
#include <stdio.h>

int main () {
    const char greeting[]="Moin!";
    printf("Greeting message: %s\n", greeting);
    return 0;
}
```



Index	0	1	2	3	4	5
Variable	M	o	i	n	!	\0
Adresse	0x23450	0x23451	0x23452	0x23453	0x23454	0x23455

# Zeichenketten in C

```
#include <stdio.h>

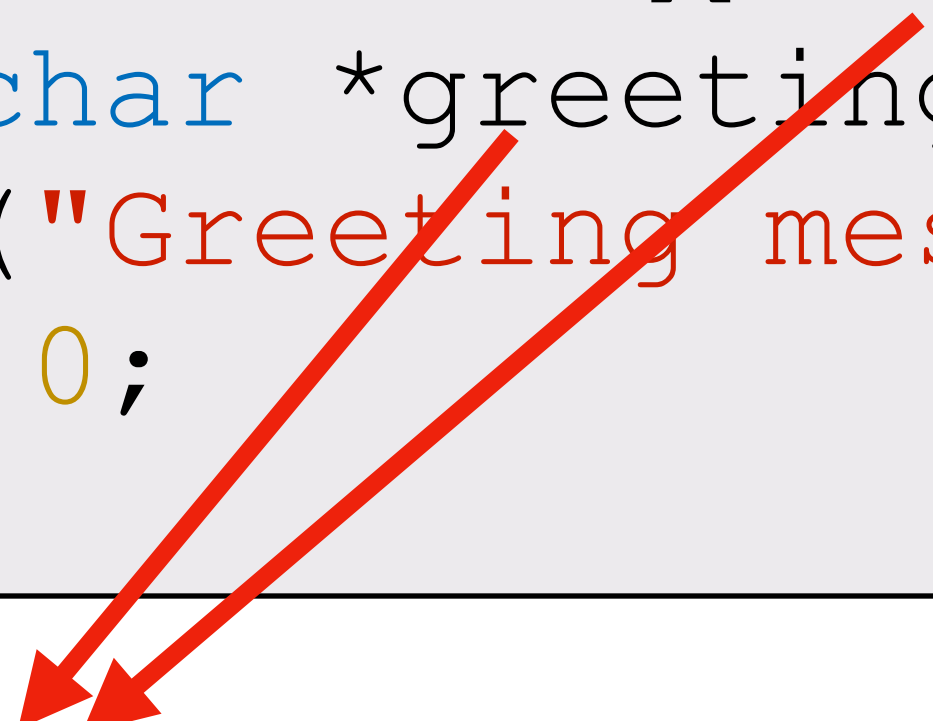
int main () {
    const char moin[]="Moin!";
    const char *greeting=moin;
    printf("Greeting message: %s\n", greeting);
    return 0;
}
```

Index	0	1	2	3	4	5
Variable	M	o	i	n	!	\0
Adresse	0x23450	0x23451	0x23452	0x23453	0x23454	0x23455

# Zeichenketten in C

```
#include <stdio.h>

int main () {
    const char moin[]="Moin!";
    const char *greeting=moin;
    printf("Greeting message: %s\n", greeting);
    return 0;
}
```



Index	0	1	2	3	4	5
Variable	M	o	i	n	!	\0
Adresse	0x23450	0x23451	0x23452	0x23453	0x23454	0x23455



# Zeichenketten in C

```
#include <stdio.h>
#include <string.h>

int main () {

    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    size_t len;

    /* copy str1 into str3. Always use strncpy */
    strncpy(str3, str1, sizeof(str3));
    printf("strncpy(str3, str1, ...): %s\n", str3);

    len = strlen(str1);
    /* concatenates str1 and str2 */
    strncat(str1, str2, sizeof(str1) - len - 1);
    printf("strncat(str1, str2, ...): %s\n", str1);

    /* total length of str1 after concatenation */
    len = strlen(str1);
    printf("strlen(str1): %zu\n", len);

    return 0;
}
```



# Strings in C++

- In C++: Strings werden auch in Arrays gespeichert
- Aber Speichermanagement passiert automatisch — und dynamisch
- Zuweisungsoperator (=) bewirkt Kopieren der Zeichenkette

```
#include <iostream>
#include <string>

using namespace std;

int main () {
    string str1 = "Hello";
    string str2 = "World";
    string str3;

    // copy str1 into str3
    str3 = str1;
    cout << "str3: " << str3 << endl;

    // concatenates str1 and str2
    str3 = str1 + str2;
    cout << "str1 + str2: " << str3 << endl;

    return 0;
}
```

# Strings in C++

```
#include <string>

using namespace std;

int main() {
    // strings erzeugen
    string hallo;
    string moin("Moin");
    string greeting(moin);

    // strings kopieren
    hallo=moin;

    // string manipulieren
    moin.append(greeting);
    moin.insert(4, " ");
    hallo.clear(); // löschen
```

```
// strings konkatenieren
hallo = moin + greeting;

// Zugriff auf Elemente & und Abschnitte

string m1 = hallo.substr(0, 4);
char m = m1[0];

// Eigenschaften

int l = m1.size();
bool isEmpty = m1.empty();

}
```

# Strings in C++

## Element access

<code>at</code>
<code>operator[]</code>
<code>front</code> (C++11)
<code>back</code> (C++11)
<code>data</code>
<code>c_str</code>
<code>operator basic_string_view</code> (C++17)

## Iterators

<code>begin</code> <code>cbegin</code> (C++11)
<code>end</code> <code>cend</code> (C++11)
<code>rbegin</code> <code>crbegin</code> (C++11)
<code>rend</code> <code>crend</code> (C++11)

## Capacity

<code>empty</code>
<code>size</code> <code>length</code>
<code>max_size</code>
<code>reserve</code>
<code>capacity</code>
<code>shrink_to_fit</code> (C++11)

## Operations

<code>clear</code>
<code>insert</code>
<code>erase</code>
<code>push_back</code>
<code>pop_back</code> (C++11)
<code>append</code>
<code>operator+=</code>
<code>compare</code>
<code>starts_with</code> (C++20)
<code>ends_with</code> (C++20)
<code>replace</code>
<code>substr</code>
<code>copy</code>
<code>resize</code>
<code>swap</code>

## Search

<code>find</code>
<code>rfind</code>
<code>find_first_of</code>
<code>find_first_not_of</code>
<code>find_last_of</code>
<code>find_last_not_of</code>

## Non-member functions

<code>operator+</code>
<code>operator==</code> <code>operator!=</code> <code>operator&lt;</code> <code>operator&gt;</code> <code>operator&lt;=</code> <code>operator&gt;=</code>
<code>std::swap</code> (std::basic_string)
<code>erase</code> (std::basic_string) <code>erase_if</code> (std::basic_string) (C++20)
<b>Input/output</b>
<code>operator&lt;&lt;</code> <code>operator&gt;&gt;</code>
<code>getline</code>

# Pointer in C

1. Dynamische Speicherverwaltung
2. Effiziente Parameterübergabe (“Call by Reference”)
3. Übergabe von Ergebnisparametern (“Call by Reference”)

unübersichtlich, unsicher in der Benutzung

(z. B. dangling pointer, double free, memory leak)

# Referenzen in C++

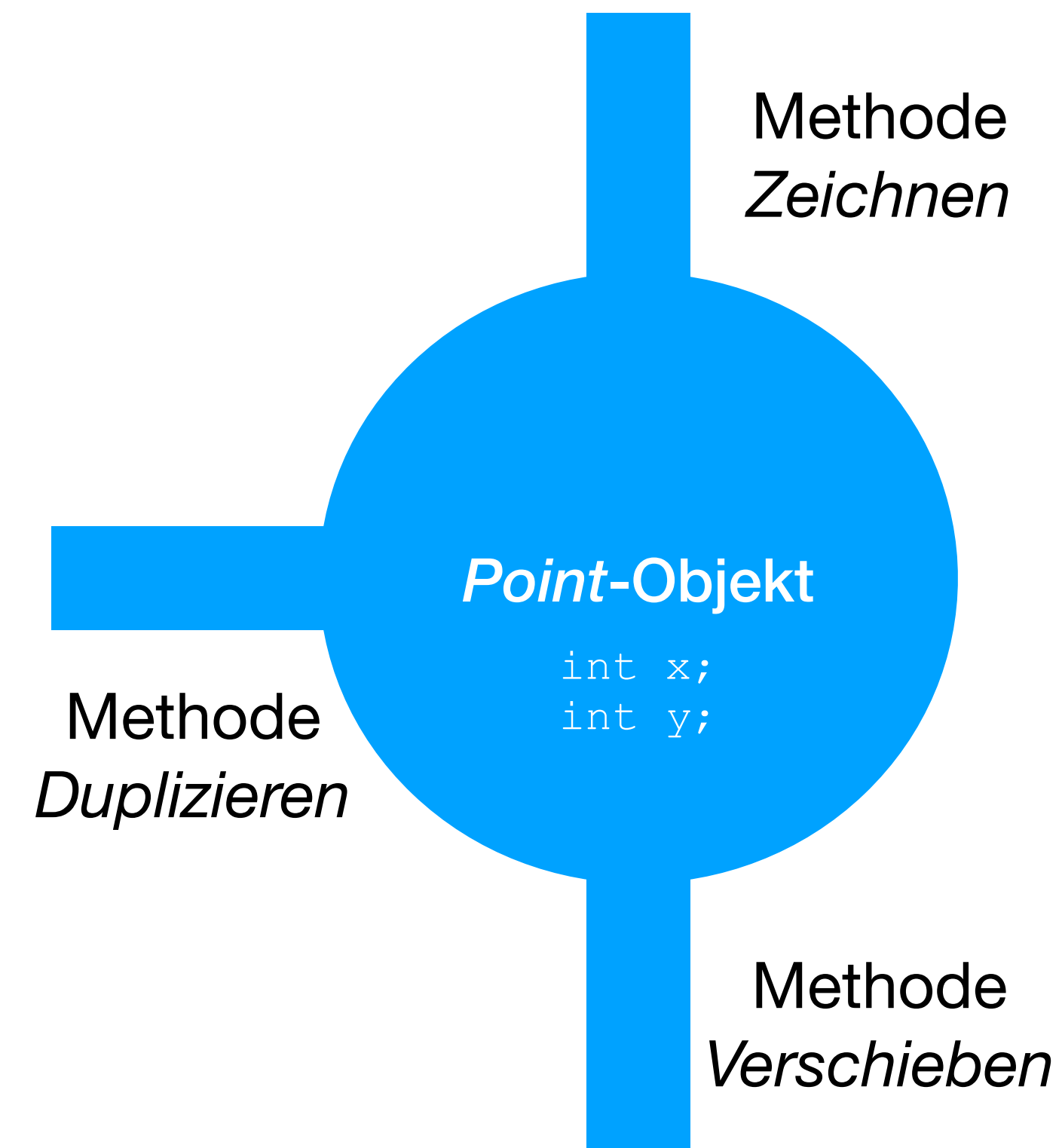
1. ~~Dynamische Speicherverwaltung~~
2. Effiziente Parameterübergabe (“Call by Reference”)
3. Übergabe von Ergebnisparametern (“Call by Reference”)

immer an existierendes Objekt gebunden

(keine dangling pointer möglich, automatisch entfernt)

# Klassen in C++

- Vollwertige Abstrakte Datentypen
- Die sich wie “eingebaute” Datentypen verhalten
- Interne Datenstruktur: ***Kapselung***
- Extern aufrufbare Funktionen (Methoden)



# Beispiel

Deklaration der Member-Funktion



```
public:  
    int size();  
};
```

Definition der Member-Funktion



```
int SimpleString::size() {  
    return strSize;  
}
```

Verwendung der Member-Funktion



```
std::cout << greeting.size();  
  
return 0;  
}
```

```
#include <iostream>
```

```
class SimpleString {
```

```
    char buffer[128];
```

```
    int strSize;
```

```
public:
```

```
    int size();
```

```
};
```

```
int SimpleString::size() {
```

```
    return strSize;
```

```
}
```

```
int main() {
```

```
    SimpleString greeting;
```

```
    std::cout << greeting.size();
```

```
    return 0;
```

```
}
```



# Konstrukturen

- Klassen benötigen in der Regel eine Form der Initialisierung
- Initialwert annehmen
- Member-Objekte initialisieren
- ggf. Speicher allozieren
- Dafür hat C++ das Sprachelement “Konstruktor”

```
#include <cstring>
#include <iostream>

class SimpleString {

    char buffer[128];
    int strSize;

public:
    void init (const char* initString);
    int size();
};

int SimpleString::size() {
    return strSize;
}

void SimpleString::init(const char* initString) {
    strncpy(buffer, initString, 128);
    strSize=strlen(buffer);
}

int main() {
    SimpleString greeting;

    greeting.init("Moin");

    std::cout << greeting.size();

    return 0;
}
```

# Konstrukturen

Deklaration des Konstruktors

```
#include <string.h>
#include <iostream>

class SimpleString {

    char buffer[128];
    int strSize;

    void init (const char* initString);
public:
    SimpleString(const char* initString);
    int size();
};
```

Definition des Konstruktors

```
int SimpleString::size() {
    return strSize;
}

void SimpleString::init(const char* initString) {
    strncpy(buffer, initString, 128);
    strSize=strlen(buffer);
}

SimpleString::SimpleString(const char* initString) {
    init(initString);
}
```

Verwendung des Konstruktors

```
int main() {
    SimpleString greeting("Moin");

    std::cout << greeting.size();

    return 0;
}
```

# Anforderungen

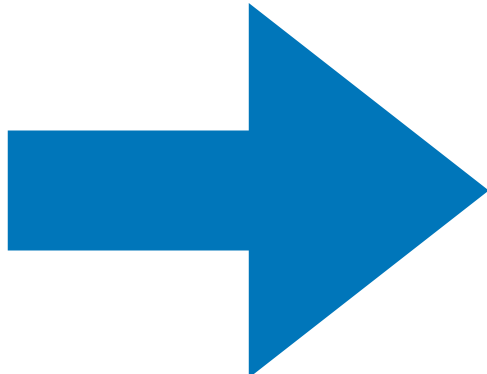
```
int main() {  
    SimpleString greeting1("Moin");  
    SimpleString greeting(greeting1);  
    SimpleString name(" C++");  
  
    greeting.add(name);  
  
    cout << greeting.str() << endl;  
  
    return 0;  
}
```

# Programmierschnittstelle

```
int main() {  
    SimpleString greeting1("Moin");  
    SimpleString greeting(greeting1);  
    SimpleString name(" C++");  
  
    greeting.add(name);  
  
    cout << greeting.str() << endl;  
  
    return 0;  
}
```

```
class SimpleString {  
  
public:  
    SimpleString();  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
};
```

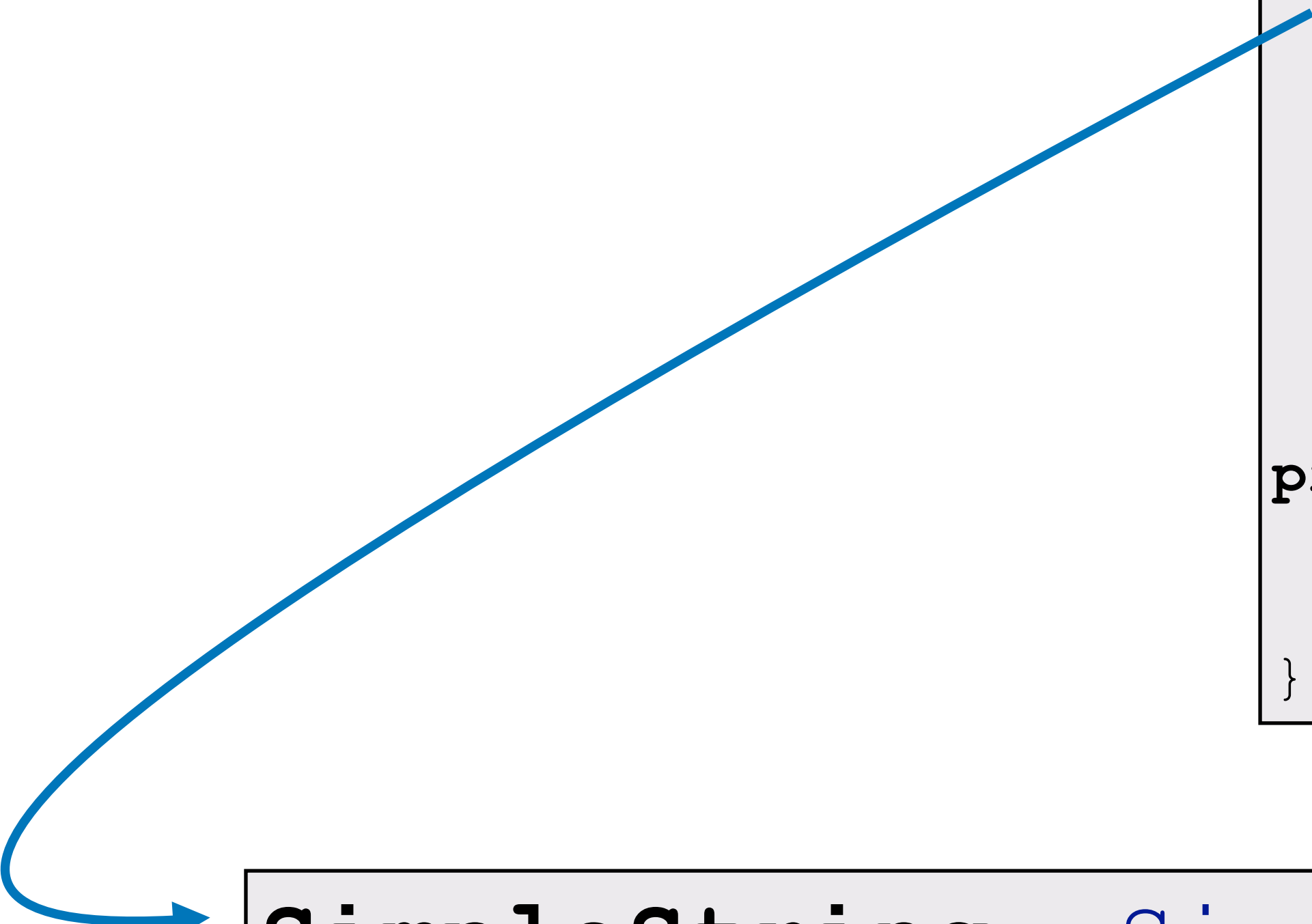
# String-Repräsentation



```
class SimpleString {  
  
public:  
    SimpleString();  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
private:  
    char *buffer;  
    size_t bufferSize;  
};
```

# String-Implementierung: Konstruktor

```
class SimpleString {  
  
public:  
    SimpleString(void);  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
private:  
    char *buffer;  
    size_t bufferSize;  
};
```



```
SimpleString::SimpleString(void)  
: buffer(nullptr), bufferSize(0) {  
}
```

# String-Implementierung: Konstruktor

```
class SimpleString {  
  
public:  
    SimpleString(void);  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
private:  
    char *buffer;  
    size_t bufferSize;  
  
    size_t increaseBuffer(size_t newSize);  
};
```

```
SimpleString::SimpleString(void)  
    : buffer(nullptr), bufferSize(0) {  
}  
  
SimpleString::SimpleString(const char *initString)  
    : SimpleString() {  
    size_t stringLength=strlen(initString);  
  
    size_t sz = increaseBuffer(stringLength);  
    strncpy(buffer, initString, sz);  
}
```



# String-Implementierung

```
SimpleString::SimpleString(void)
: buffer(nullptr), bufferSize(0) {
}

SimpleString::SimpleString(const char *initString)
: SimpleString() {
    size_t stringLength=strlen(initString);

    size_t sz = increaseBuffer(stringLength);
    strncpy(buffer, initString, sz);
}

size_t SimpleString::increaseBuffer(size_t newSize) {
    char *newBuffer = new char[newSize+1]; // may throw std::bad_alloc
    if(bufferSize > 0) { // we have existing string data
        strncpy(newBuffer, buffer, newSize+1);
        delete[] buffer; // we don't need the old buffer anymore
    }
    buffer = newBuffer;
    bufferSize = newSize+1;
    return bufferSize;
}
```

# Konstrukturen in C++

```
int main() {
    SimpleString greeting1("Moin");
    SimpleString greeting(greeting1);
    SimpleString name(" C++");

    greeting.add(name);

    cout << greeting.str() << endl;

    return 0;
}
```

```
class SimpleString {
public:
    SimpleString(void);
    SimpleString(const char *initString);
    SimpleString(const SimpleString &initString);

    const char *str(void) const;
    void add(const SimpleString &addedString);

private:
    char* buffer;
    int bufferSize;

    size_t increaseBuffer(size_t newSize);
};
```

- **Konstrukturen**

- Besondere Member-Funktionen
- Zum Initialisieren eines Objekts
- Heissen so wie die Klasse
- Kann man nicht direkt aufrufen (nur beim Initialisieren)
- Verschiedene Varianten

## Drei grundlegende Arten von Konstrukturen

- **Default-Konstruktor:** ohne Parameter
- **Copy-Konstruktor:** Parameter vom Typ der eigenen Klasse
- **Converting-Konstruktor:** sonstige Parameter

# Konstrukturen in C++

```
int main() {  
    SimpleString greeting1;  
    return 0;  
}
```

```
class SimpleString {  
public:  
    SimpleString(void);  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
private:  
    char *buffer;  
    size_t bufferSize;  
  
    size_t increaseBuffer(size_t newSize);  
};
```

- **Default-Konstruktor**

- Keine Argumente
- Immer vorhanden, auch wenn nicht explizit definiert (ruft dann implizit die Konstrukturen etwaiger Member auf)

- **Konstrukturen**

- Besondere Member-Funktionen
- Zum Initialisieren eines Objekts
- Heissen so wie die Klasse
- Kann man nicht direkt aufrufen (nur beim Initialisieren)
- Verschiedene Varianten

```
SimpleString::SimpleString(void)  
    : buffer(nullptr), bufferSize(0) {  
}
```

# Konstrukturen in C++

```
int main() {  
    SimpleString greeting1("Moin");  
    SimpleString greeting(greeting1);  
}
```

- **Copy-Konstruktor**

- const Reference auf ein bestehendes Objekt der Klasse als Parameter
- Initialisiert das neue Objekt mit dem (per Reference) übergebenen Objekt
- Wenn nicht explizit definiert: Kopiert die einzelnen Member

```
class SimpleString {  
  
public:  
    SimpleString(void);  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
private:  
    char *buffer;  
    size_t bufferSize;  
  
    size_t increaseBuffer(size_t newSize);  
};
```

- **Konstrukturen**

- Besondere Member-Funktionen
- Zum Initialisieren eines Objekts
- Heissen so wie die Klasse
- Kann man nicht direkt aufrufen (nur beim Initialisieren)
- Verschiedene Varianten

```
SimpleString::SimpleString  
    (const SimpleString &initStr) {  
    // Details gleich angucken  
}
```

# Konstrukturen in C++

```
int main() {  
    SimpleString greeting1("Moin");  
}
```

- **Converting Konstruktor**

- Parameter eines anderen Typs
- Neues Objekt mit dem übergebenen Objekt initialisieren
- Oder: Bestehendes Objekt in ein neues Objekt der Klasse konvertieren

```
class SimpleString {  
  
public:  
    SimpleString(void);  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
private:  
    char *buffer;  
    size_t bufferSize;  
  
    size_t increaseBuffer(size_t newSize);  
};
```

- **Konstrukturen**

- Besondere Member-Funktionen
- Zum Initialisieren eines Objekts
- Heissen so wie die Klasse
- Kann man nicht direkt aufrufen (nur beim Initialisieren)
- Verschiedene Varianten

```
SimpleString::SimpleString(const char *initString)  
: SimpleString() {  
    size_t stringLength = strlen(initString);  
  
    size_t sz = increaseBuffer(stringLength);  
    strncpy(buffer, initString, sz);  
}
```

# Copy-Konstruktor

```
int main() {  
    SimpleString greeting1("Moin");  
    SimpleString greeting(greeting1);  
    SimpleString name(" C++");  
  
    greeting.add(name);  
  
    cout << greeting.str() << endl;  
  
    return 0;  
}
```



```
class SimpleString {  
  
public:  
    SimpleString(void);  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
private:  
    char *buffer;  
    size_t bufferSize;  
  
    size_t increaseBuffer(size_t newSize);  
};
```



# Copy-Konstruktor

```
int main() {
    SimpleString greeting1("Moin");
    SimpleString greeting(greeting1);
    SimpleString name(" C++");

    greeting.add(name);

    cout << greeting.str() << endl;

    return 0;
}
```



```
class SimpleString {
public:
    SimpleString(void);
    SimpleString(const char *initString);
    SimpleString(const SimpleString &initString);

    const char *str(void) const;
    void add(const SimpleString &addedString);

private:
    char *buffer;
    size_t bufferSize;

    size_t increaseBuffer(size_t newSize);
};
```

```
SimpleString::SimpleString(const char *initString)
: SimpleString() {
    if (initString) {
        size_t stringLength = strlen(initString);

        size_t sz = increaseBuffer(stringLength);
        strncpy(buffer, initString, sz);
    }
}
```

```
SimpleString::SimpleString(const SimpleString &initString) {
    // so ähnlich wie oben – nur mit initString.buffer
}
```

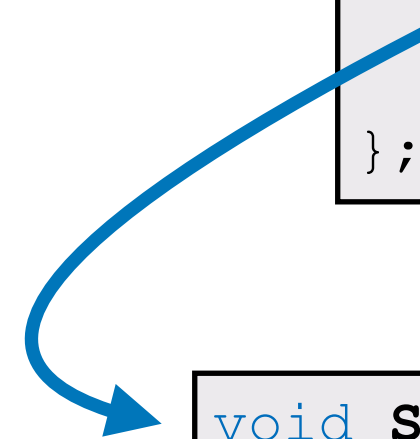


# Copy-Konstruktor

```
int main() {  
    SimpleString greeting1("Moin");  
    SimpleString greeting(greeting1);  
    SimpleString name(" C++");  
  
    greeting.add(name);  
  
    cout << greeting.str() << endl;  
  
    return 0;  
}
```



```
class SimpleString {  
  
public:  
    SimpleString(void);  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
private:  
    char *buffer;  
    size_t bufferSize;  
  
    void init(const char *initString);  
    size_t increaseBuffer(size_t newSize);  
};
```



```
SimpleString::SimpleString(void)  
    : buffer(nullptr), bufferSize(0) {  
}  
  
SimpleString::SimpleString(const char *initString)  
    : SimpleString() {  
    init(initString);  
}  
  
SimpleString::SimpleString(const SimpleString &initString) {  
    init(initString.buffer);  
}
```

```
void SimpleString::init(const char *initString) {  
    if (initString) {  
        size_t length = strlen(initString);  
  
        buffer = nullptr;  
        bufferSize = 0;  
        size_t sz = increaseBuffer(length);  
        strncpy(buffer, initString, sz);  
    }  
}
```

Weitere Optimierungen später...

# Ausblick: Details zu Konstruktoren

- Initialisierungsliste als Parameter für Konstruktoren
  - *(initializer list), Sequenzkonstruktor*
- Default-Konstruktoren explizit hinzufügen (`default`)
- Konstruktoren explizit verbieten (`delete`)

# add

```
int main() {  
    SimpleString greeting1("Moin");  
    SimpleString greeting(greeting1);  
    SimpleString name(" C++");  
  
    greeting.add(name);  
  
    cout << greeting.str() << endl;  
  
    return 0;  
}
```

```
class SimpleString {  
  
public:  
    SimpleString(void);  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
private:  
    char* buffer;  
    size_t bufferSize;  
  
    void init(const char *initString);  
    size_t increaseBuffer(size_t newSize);  
};
```

```
void SimpleString::add(const SimpleString &addedString) {  
    size_t newSize = size() + addedString.size();  
    size_t sz = newSize + 1;  
    if (bufferSize < newSize + 1) {  
        sz = increaseBuffer(newSize);  
    }  
    strncat(buffer, addedString.buffer, sz);  
}
```

# str

```
int main() {  
    SimpleString greeting1("Moin");  
    SimpleString greeting(greeting1);  
    SimpleString name(" C++");  
  
    greeting.add(name);  
  
    cout << greeting.str() << endl;  
  
    return 0;  
}
```

```
class SimpleString {  
  
public:  
    SimpleString(void);  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
  
    const char *str(void) const;  
    void add(const SimpleString &addedString);  
  
private:  
    char *buffer;  
    size_t bufferSize;  
  
    void init(const char *initString);  
    size_t increaseBuffer(size_t newSize);  
};
```

```
const char *SimpleString::str(void) const {  
    return buffer;  
}
```

# Was fehlt noch?

```
int main() {
    SimpleString greeting1("Moin");
    SimpleString greeting(greeting1);
    SimpleString name(" C++");

    greeting.add(name);

    cout << greeting.str() << endl;

    return 0;
}
```

```
class SimpleString {
public:
    SimpleString(void);
    SimpleString(const char *initString);
    SimpleString(const SimpleString &initString);

    const char *str(void) const;
    void add(const SimpleString &addedString);

private:
    char *buffer;
    size_t bufferSize;

    void init(const char *initString);
    size_t increaseBuffer(size_t newSize);
};
```

# Was fehlt noch?

```
int main() {
    SimpleString greeting1("Moin");
    SimpleString greeting(greeting1);
    SimpleString name(" C++");

    greeting.add(name);

    cout << greeting.str() << endl;

    return 0;
}
```

```
class SimpleString {
public:
    SimpleString(void);
    SimpleString(const char *initString);
    SimpleString(const SimpleString &initString);

    const char *str(void) const;
    void add(const SimpleString &addedString);

private:
    char *buffer;
    size_t bufferSize;

    void init(const char *initString);
    size_t increaseBuffer(size_t newSize);
};
```

- **Lebensdauer von lokalen Variablen**
  - Lokale Variablen werden in einem {}-Block instanziiert
  - Und beim Verlassen des Blocks automatisch gelöscht
    - Speicherplatz (für die Member-Variablen) freigeben
  - Bei Objekten von Klassen möchte man noch mehr Kontrolle haben
    - Dynamisch allozierten Speicher freigeben
    - Sonstige “Aufräumaktionen”
- ➤ **Destruktoren**

# Destruktor

```
int main() {  
    SimpleString greeting1("Moin");  
    SimpleString greeting(greeting1);  
    SimpleString name(" C++");  
  
    greeting.add(name);  
  
    cout << greeting.str() << endl;  
  
    return 0;  
}
```

```
class SimpleString {  
  
public:  
    SimpleString();  
    SimpleString(const char *initString);  
    SimpleString(const SimpleString &initString);  
    ~SimpleString(void);  
  
    const char *str(void);  
    void add(const SimpleString &addedString);  
  
private:  
    char *buffer;  
    size_t bufferSize;  
  
    void init(const char *initString);  
    size_t increaseBuffer(size_t newSize);  
};
```

- **Lebensdauer von lokalen Variablen**

- Lokale Variablen werden in einem {}-Block instanziiert
- Und beim Verlassen des Blocks automatisch gelöscht
  - Speicherplatz (für die Member-Variablen) freigeben
- Bei Objekten von Klassen möchte man noch mehr Kontrolle haben
  - Dynamisch allozierten Speicher freigeben
  - Sonstige “Aufräumaktionen”

- ➤ **Destruktoren**

```
SimpleString::~SimpleString(void) {  
    delete[] buffer;  
}
```



# Alles zusammen

```
#include <cstring>
#include <cstdlib>
#include <iostream>

// SimpleString class with dynamic memory management

using namespace std;

class SimpleString {
public:
    SimpleString(void);
    SimpleString(const char *initString);
    SimpleString(const SimpleString &initString);
    ~SimpleString(void);

    const char *str(void) const;
    void add(const SimpleString &addedString);

private:
    char *buffer;
    size_t bufferSize;

    void init(const char *initString);
    size_t increaseBuffer(size_t newSize);
};
```

```
int main() {
    SimpleString greeting1("Moin");
    SimpleString greeting(greeting1);
    SimpleString name(" C++");

    greeting.add(name);

    cout << greeting.str() << endl;

    return 0;
}
```

```
/ Standard constructor
SimpleString::SimpleString(void)
    : buffer(nullptr), bufferSize(0) {
}

// Converting constructor
SimpleString::SimpleString(const char *initString)
    : SimpleString() {
    init(initString);
}

// Copy constructor
SimpleString::SimpleString(const SimpleString &initString)
    : SimpleString() {
    init(initString.buffer);
}

// Destructor
SimpleString::~SimpleString(void) {
    delete[] buffer;
}

void SimpleString::init(const char *initString) {
    if (initString) {
        size_t length = increaseBuffer(strlen(initString));
        strncpy(buffer, initString, length);
    }
}

size_t SimpleString::increaseBuffer(size_t newSize) {
    if (newSize > 0) { // change only when necessary
        char *newBuffer = new char[newSize + 1]; // may throw std::bad_alloc
        if (bufferSize > 0) {
            strncpy(newBuffer, buffer, bufferSize);
            delete[] buffer;
        }

        buffer = newBuffer;
        bufferSize = newSize + 1;
    }
    return bufferSize;
}

void SimpleString::add(const SimpleString &addedString) {
    size_t newSize = size() + addedString.size();
    size_t sz = newSize + 1;
    if (bufferSize < newSize + 1) {
        sz = increaseBuffer(newSize);
    }
    strncat(buffer, addedString.buffer, sz);
}
```