C&C++ durch Beispiele

„Man kann C und C++ nicht in einer Woche vor der Klausur lernen“ - H.D.

Jan Maximilian Caspar

Zusammenfassung

Eine Einführung in C und C++ anhand von Beispielen. Ein Revue passieren lassen eines Semesters SWO3, C, C++ und Spaß rundherum. Beispiele mit ein wenig Theorie oben drüber.

Inhaltsverzeichnis

[1] einzelne Beispiele von https://www.cprogramming.com/tutorial/c-vs-c++.html
[2] Quelle des Beispiels http://cs.fit.edu/~mmahoney/cse1502/introcpp.html

[3] http://www.bogotobogo.com/cplusplus/cplusplus_keywords.php

Teil I. C: ein Schnelldurchlauf

1 Anatomie eines C Programms

Listing 1: **Beispiel: einfaches C-Programm**

```c
#include <stdio.h>

int main(int argc , char * argv[] ){
    if(argc != 2){
        printf("Wrong number of arumgents!\n");
        printf("Usage: %s <args> \n", argv[0]);
        return -1;
    }else{
        printf("args = \"%s\"\n",argv[1]);
    }
    return 0;
}
```


Args

Kommandozeilen Argumente wandern an die Main-Funktion über die Variablen argc, die Anzahl der übergebenen Argumente, und argv die übergebenen Argumente, wobei argv[0] immer den Pfad zur Applikation beinhaltet.

Argumente in Integer konvertieren

Listing 2: **Beispiel: Argument in Integer konvertieren und aufsummieren**

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    int sum = 0;
    for(int i = 1; i < argc; i++){
        int num;
        sscanf(argv[i], "%d", &num);
        sum += num;
    }
    printf("sum = %d\n",sum);
    return EXIT_SUCCESS;
}
```

Speicheranordnung

Code / Text **ReadOnly**	f() | f2() | ... | main() | strcopy()
Data	readonly constants | R/W data global vars | BSS uninit data
Heap	Heap (wächst nach unten)
shared libs	Shared Library Code
Stack	[f()] [f2()][main][ret][p] [ln] (Funktion + lokale Variablen + Parameter)

Die Größe der Speicherbereich lässt sich mit size ermitteln z.B.

```
[jan@starshine 30.09]$ gcc -o segment-sizes segment-sizes.c
[jan@starshine 30.09]$ size segment-sizes
   text    data     bss     dec     hex filename
   1499     592       8    2099     833 segment-sizes
```

2 Ein- und Ausgabe

printf

Listing 3: **Beispiel: Hello World mit printf**

```c
#include <stdio.h>

int main(int argc, char * argv[]){
    printf("Hello World");
    return 0;
}
```

Darstellungen

Zeichen	Format
%d	int
%u	unsigned int
%h	short int
%o / %x	octal / hex
%p	pointer
%f	float
%e	Exponent Darstellung
%s	String
%c	char

Modifikatoren

%6d auf 6 Stellen auffüllen

%-6s Pad-Right 6 Stellen

%ld longint

%*d variale Länge

%62f 6 Stellen insgesamt, 2 Nachkomma daher 3 Vorkommastellen

%% wird zu %

Ein Beispiel

Listing 4: **Beispiel: printf mit Formatierung**

```c
#include <stdio.h>

int main(){
    int i = 24;
    float x = 2.781;
    char *s = "addition";
    printf("%-12s: %03d + %f = %10.2f\n",s,i,x,i+x);
    printf("%0*.*f\n", 8,3,12.34);
    return 0;
}
```

Ausgabe

addition : 024 + 2.781000 = 26.78
0012.340

scanf

scanf ist das Eingabe-Äquivalent zu printf

Listing 5: **Beispiel: scanf**

```c
#include <stdio.h>

int main(){
    char name[20];
    printf("Name?");
    scanf("%s", name);
    printf("Hello %s\n", name);
    return 0;
}
```

3 Strings

Strings sind in C mit besonderer Vorsicht zu behandeln, sie sind als char-Arrays umgesetzt und werden **NULL-terminiert**. Der letzte Character in dem char-Array muss immer der NULL-Terminator **\0** sein, da sonst das Programm nicht aufhört Daten zu lesen bis es den ersten NULL-Terminator findet. Deshalb haben diese Arrays auch immer die Länge $Zeichenkette + 1$, da \0 auch einen „Slot“ im Array braucht.

Standarlibrary Funktionen

strlen(s) **gibt die Länge des Strings zurück**

strcpy(dest,src) **kopiert einen String - Achtung füllt ganze Größe auf**

strcat(dest,src) **konkateniert einen String**

strcmp(a,b) **vergleicht String a mit String b**

string.h Funktionen

strncpy(dest,src,n) **kopieren mit Zeichenanzahl vorgegeben**

4 Arrays

Arrays sind in C als Pointer realisiert, also ist jedes int[] essentiell nichts anderes wie eine Sammlung von Pointern auf Integer. Aus diesem Grund ist ein Array niemals Pass-by-value! Die Syntax um ein Array anzulegen ist `type id[<int>];` als für ein 100 Felder int-Array `int values[100];`. Mehrdimensionale Arrays in C sind so genannte jagged Arrays, da Arrays nur aus Pointern bestehen sind mehrdimensionale Arrays Pointer mit Pointer im jeweiligen Index. Der Speicherbedarf für `type a[SIZE1][SIZE2][SIZEU];` ist stets sizeof(a) * SIZE1 * SIZE2 * SIZEU

Listing 6: **Beispiel: Arrays**

```c
#include <stdio.h>

int main(){
    // declare array with 10 integer values
    int values[10];
    // declare and assign a array with 3 integer values
    int othervalues[3] = { 1, 2, 3 };
    // compiler will generate a array of 4
    int anothervalues[] = { 1, 2, 3, 4 };
     // 2x2 array
    int twodimensional[2][2];
    // gets 1 from other values array, arrays start with 0
    int one = othervalues[0];
    //same as above
    int alsoOne = *(othervalues + 0);
    // &othervalues[0] = othervalues + 0 = address of integer on
       position 0
    int sizeOfArray = sizeof(othervalues) / sizeof(othervalues[0])
       ;
    return 0;
}
```

5 Typecast & Integer Promotion

Über den Cast-Operator lassen sich Variablentypen „umcasten“, die Syntax dafür ist `(type) expression`, wenn man also nun einen Integer explizit auf einen double casten möchte: `double a = (double)b;` unter der Annahme das b ein Integer ist. Der Compiler selbst kann implizit das Casting vornehmen, dies ist über die Integer Promotion festgelegt.

Listing 7: **Beispiel: expliziter Typecast**

```
#include <stdio.h>

main() {
    int  i = 17;
    char c = 'c'; /* ascii code value = 99 */
    int sum;
    sum = i + c;
    printf("Sum is %d\n", sum );
}
```

Integer Promotion

Die oben erwähnte implizite Umwandlung lässt sich anhand dieses Beispiels gut erkennen.

Listing 8: **Beispiel: Integer Promotion**

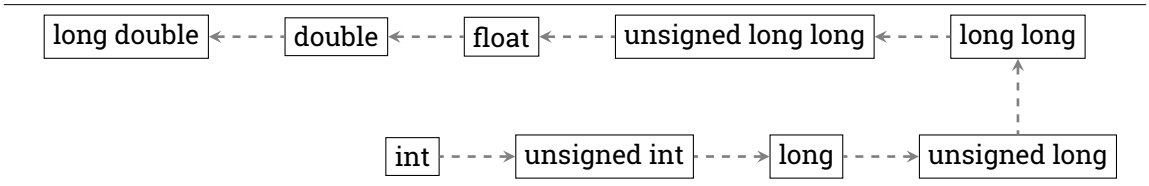
```
#include <stdio.h>

main() {
    int  i = 17;
    char c = 'c'; /* ascii code value = 99 */
    int sum;
    sum = i + c;
    printf("Sum is %d\n", sum );
}
```

Ausgabe

Die Ausgabe dieser Anwendung ist **Sum is 116**, da der Character automatisch durch die Integer-Promotion auf 99 umgewandelt wird.

Integer Promotion Regeln



6 Call by value vs Call by reference

Call by reference lässt sich nur über Pointer realisieren in C.

Call-By-Value (**FALSCH** in diesem Fall)

Listing 9: **Beispiel: Call-By-Value Swap**

```c
#include <stdio.h>

void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}

int main(){
    int x = 10;
    int y = 20;

    printf("x = %d, y = %d\n",x
    ,y);
    swap(x, y);
    printf("x = %d, y = %d\n",x
    ,y);

    return 0;
}
```

Ergebnis
x = 10, y = 20
x = 10, y = 20

Call-By-Reference

Listing 10: **Beispiel: Call-By-Reference Swap**

```c
#include <stdio.h>

void swap(int *x, int *y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int x = 10;
    int y = 20;

    printf("x = %d, y = %d\n",x
    ,y);
    swap(&x, &y);
    printf("x = %d, y = %d\n",x
    ,y);

    return 0;
}
```

Ergebnis
x = 10, y = 20
x = 20, y = 10

7 static, volatile & extern Variablen

static

erzeugt lokale Variable

Listing 11: Beispiel: static Variable

```
#include <stdio.h>

void f(){
    static int cnt = 0;
    cnt++;
    printf("f() was called %d times\n",cnt);
}

int main(){
    for(int i = 0;i < 10; i++){
        f();
    }

    return 0;
}
```

Ausgabe

```
f() was called 1 times
f() was called 2 times
f() was called 3 times
f() was called 4 times
f() was called 5 times
f() was called 6 times
f() was called 7 times
f() was called 8 times
f() was called 9 times
f() was called 10 times
```

extern

definiert in .h Files dass die Variable in der Implementierung existiert (nur Deklaration).

volatile

markiert die Variable, dass sie von „außen“ also außerhalb des Programms selbst manipuliert wird.

8 Pointer Arithmetik

++/-- springt immer die Länge des Datentypes

Listing 12: **Beispiel: Pointer Arithmetik**

```
#include <stdio.h>

int main(){
    char text[5] = { 'N', 'e', 'i', 'n', '\0'}
    do {
        //do something with text
    } while (*text++ != '\0');

    return 0;
}
```

Anmerkung

In dem obigen Beispiel wird das char-Array Text mit Pointer Arithmetik durchlaufen (*text++).

9 malloc, calloc & realloc

malloc Reserviert dynamischen Speichern, die genaue Signatur lautet `void *malloc(size_t size)`, wobei **size** die Größe in Byte ist.

calloc Reserviert dynamischen Speichern aber im Vergleich zu malloc setzt calloc den Speicherbereich auf 0, die genaue Signatur lautet `void *calloc(size_t nitems, size_t size)`, wobei **nitems** die Anzahl an Elemente ist und **size** die Größe der Elemente

realloc Ändert die Größe eines Speicherbereichs hinter einem Pointer, die genaue Signatur lautet `void *realloc(void *ptr, size_t size)`, wobei **ptr** der Pointer auf den Speicherbereich ist und **size** die neue Größe in Byte ist.

free gibt den Speicher wieder frei, die genaue Signatur lautet `void free(void *ptr)` wobei **ptr** ein Zeiger auf den Speicherbereich ist.

Achtung, malloc, calloc, realloc geben **NULL** zurück wenn kein Speicher mehr verfügbar ist

Listing 13: Beispiel: malloc

```c
#include <stdio.h>
int main(){
    int n = 2; //length of the string
    char *s = (char*)malloc(sizeof(char)*(n+1)); // +1 --> \0
    s[2] = 'x';
    *(s+2) = 'x'; //same as s[2], pointer arith.
    free(s);
    return 0;
}
```

Listing 14: Beispiel: malloc, realloc, free

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(){
    char *str;
    str = (char*) malloc(sizeof(char)*4); //reserve the space
    if(str == NUL) printf("OUT OF MEMORY"); // out of memory
    strcpy(str, "jan");
    printf("Hello %s\n", str); // Hello jan

    str = (char*)realloc(str, sizeof(char)*15); //resize space
    if(str == NULL) printf("OUT OF MEMORY"); // out of memory
    strcat(str, " maximilian");
    printf("Hello %s\n", str); // Hello jan maximilian
    free(str); //release memory
    return 0;
}
```

10 StandardLib Funktionen

Häufige genutzte Funktionen aus der C StandardLib:

assert Assertion, um Bedingungen zu prüfen

stdarg.h

div Ganzzahldivision

random/srandom Random und seeded Random / Zufallszahlengenerator

qsort,bsearch Quicksort, Binäre Suche

fopen,fwrite,fclose,fseek,ftell Dateioperationen

11 Datentypen

Skalare [4]

Ganzzahling

Typ	Größe	Bereich
char	1 byte	-128 bis 127 oder 0 bis 255
unsigned char	1 byte	0 bis 255
signed char	1 byte	-128 bis 127
int	2 or 4 bytes	-32 768 to 32 767 oder -2 147 483 648 to 2 147 483 647
unsigned int	2 or 4 bytes	0 bis 65 535 oder 0 bis 4 294 967 295
short	2 bytes	-32 768 bis 32 767
unsigned short	2 bytes	0 bis 65 535
long	4 bytes	-2 147 483 648 bis 2 147 483 647
unsigned long	4 bytes	0 bis 4 294 967 295

Fließkomma

Typ	Größe	Bereich	Genauigkeit
float	4 byte	1.2E-38 bis 3.4E+38	6 Dezimalstellen
double	8 byte	2.3E-308 bis 1.7E+308	15 Dezimalstellen
long double	10 byte	3.4E-4932 bis 1.1E+4932	19 Dezimalstellen

Void

Funktionsrückgabe void

Wenn eine Funktion nichts zurück gibt, so ist ihr Rückgabetyp void.

Funktionsargument void

Wenn eine Funktion keine Argumente nimmt so kann ihr Argumenttyp als void definiert werden z.B. int DoSomething(void);

Void-Pointer

Ein Pointer auf vom Typ void repräsentiert die Adresse eines beliebigen Objektes, aber nicht seinen Typ. malloc z.B. gibt einen void Pointer zurück, welcher dann auf einen beliebigen Typen gecastet werden kann.

[4] Quelle: https://www.tutorialspoint.com/cprogramming/c_data_types.htm

Structs

Structs sind Verbunddatentypen, die Sichtbarkeit von allen „Members“ ist public. Structs sind ebenfalls vom „Call-By-Value“ Problem betroffen.

Syntax von Structs

```
struct tag_name
{
    type name_1;
    type name_2;
    type name_n;
};
```

Listing 15: **Beispiel: struct ohne Pointer**

```
#include <stdio.h>
struct person
{
    int id;
    char name[20];
};
int main(){
    struct person someone = {1, "Someone"};
    //Person (1) Someone
    printf("Person: (%d) %s \n", someone.id, someone.name);
    return 0;
}
```

Listing 16: **Beispiel: struct mit Pointer**

```
#include <stdio.h>
struct person
{
    int id;
    char name[20];
};
int main(){
    struct person *someone, s;
    s = {1, "Someone"};
    someone = &s;
    //Person (1) Someone
    printf("Person: (%d) %s \n", someone->id, someone->name);
    return 0;
}
```

12 Typedef

Typedef erlaubt „Labels“ für Typen zu vergeben um diese anschließend als Typenamen verwenden zu können.

Listing 17: **Beispiel: typedef**

```c
#include <stdio.h>
//typedef + enum
typedef enum personStatus { Active, Inactive } PersonStatus;

//typedef + struct
typedef struct person
{
    int id;
    char name[20];
    PersonStatus status;
} Person; //typedef with tag Person

int main(){
    //no need for struct keyword
    Person someone = {1, "Someone", Active};
    //Person (1) Someone
    printf("Person: (%d) %s \n", someone.id, someone.name);
    return 0;
}
```

13 Makefiles

Makefiles werden als Build-Systeme genutzt, ihr Aufbau ist stets:
target : depdencies
    (tab) command args

simples makefile

main.c benötigt geo.c und weight.c, weight.c benötigt geo.c

Listing 18: **Ein einfaches makefile**

```
CC = gcc # compiler
CFLAGS = -pedantic -Wall -Wextra -std=c11 -ggdb # compiler flags

LD = gcc # linker
LDFLAGS = -lm # not really needed (links math lib m)

.c.o: # sufix  rule (*.c -> *.o)
    $(CC) $(CFLAGS) -c $<
# $< name of first dependecy / prerequisite

#main application
geo: main.o geo.o weight.o
    $(LD) -o $@ main.o geo.o weight.o $(LDFLAGS)
# @a targetname

# dependecies
geo.o: geo.c geo.h # first dep
weight.o: weight.c weight.h geo.h
main.o: main.c geo.h weight.h
```

Teil II. C++ - wo warst du als der Spaß aufhörte?

Die C-Standardbibliothek ist vollständig in der C++-Standard-(Klassen-)Bibliothek enthalten

14 Von C auf C++[5]

Der C++ Compiler kann auch C Programme kompilieren (nachdem C ja ein Subset von C++ ist) allerdings sind die „Constrains" viel strenger, hier sind häufige „Pitfalls" die dem übergang von C auf C++ passieren.

Impliziertes casten von void*

In C ist impliziertes casten von void* Stern möglich und auch „guter Stil". `int *x = malloc(sizeof(int)* 10);` ist in C vollkommen in Ordnung, aber wird mit einem C++ Kompiler nicht kompilieren. Der Grund hierfür ist das malloc **nicht type-safe** ist und **void*** alles sein kann und zum Zeitpunkt des Kompilierens ist es nicht möglich festzustellen was sich hinter dem void* Pointer befindet. In C++ ist der preferable Weg für die Allocation mit dem **new Schlüsselwort**, dieses hat zwei Vorteile gegenüber malloc, es ist **type-safe** und es wird sicher gestellt das der **Konstruktor aufgerufen wird**.

Arrays allokieren & löschen (new[] and delete[])

In C werden sowohl einzelne Typen als auch Arrays gleich angelegt beide werden mit malloc reserviert und mit free gelöscht, zum Beispiel

Listing 19: Beispiel: C vs CPP: malloc free

```
int *x = malloc(sizeof(int));
int *x_array = malloc(sizeof(int) * 10);
free(x);
free(x_array);
```

In C++ hingegen ist Arrays allokieren und löschen anders, hierfür gibt es die operationen new[] und delete[] welche explizit für Arrays geschaffen wurden. Dies hat den Grund, dass der delete[] Operator den Konstruktor jedes Elements in dem Array aufruft.

Listing 20: delete[]]Beispiel: C vs CPP: new[] delete[]

```
int *x = new int;
int *x_array = new int[10];

delete x;
delete[] x_array;
```

C in C++

C lässt sich in C++ ausführen, aber nicht umgekehrt, die Standard C-Libraries sind in C++ mit c prefixed, zum Beispiel `#include <cstdio>` für `std::printf("%d\n", "Hello World!");`

[5] einzelne Beispiele von https://www.cprogramming.com/tutorial/c-vs-c++.html

In C++ müssen Funktionen vor ihrem Aufruf deklariert werden

In C kann die Funktion unterhalb des Aufrufes deklariert werden, dies führt in C++ allerdings zu einem Compiler-Fehler

Listing 21: Beispiel: C vs CPP: Funktionsreihenfolge

```
#include <stdio.h>
int main()
{
    foo(); //will compile in c, not in c++
    return 0;
}

int foo()
{
    printf( "Hello world" );
}
```

const

Ander als in C ist const in C++ ist tatsächlich konstant, wobei kostant vermutlich die falsche Antwort wäre, viel mehr markiert das „const“ Schlüsselwort Elemente als „inmutable“. Es ein „compile-time“ Konstrukt das genutzt wird um die „Korrektheit“ des Codes zu waren, durch das deklarieren einer Variable oder Methode als „const“ markieren wir das die Daten (auch Kinder oder „nested Elements“) nicht verändert werden. Es ist nicht notwendig, aber guter Stil und dokumentiert gleichzeitig die Design-Entscheidung, dass es hier zu keiner Änderung kommen soll.

Enumerationen

Im Vergleich zu C lassen sich Enumerationen untereinander nicht mehr einfach Vergleichen, sie sind „type-safe“.

15 Deklarationen [7]

Objekte, Typen und Funktionen können global, lokal oder im Block-Scope deklariert werden und sind case-sensistive. Alle leeren Initialisierungen von Standardtypen sind stets „undefiniert“, es ist daher empfehlenswert gleich einen Standardwert zu setzen. Ich geh jetzt mal davon aus das der Leser mit „Block-Scoping“ vertraut ist und erklär das jetzt nicht extra...

Listing 22: **Beispiel: mögliche Deklarationen**

```
// Declaring objects of existing types (global or local)
int i, j=3;               // Integers, i's initial value is undefined
double x=3.5, y=j;        // Real numbers, y is 3.0
bool t=false, u=true;     // Logical e.g. if (t && u) {...}
string s, hi="hello";     // s is initially an empty string, "" in <string>
char c='a', c2=hi[1];     // Character, c2 is 'e'
vector<int> a(5);         // Array a[0] through a[4] of int in <vector>
map<string, int> m;       // Associative array, e.g. m["hi"]=5; in <map>
ifstream in("file.txt"); // Input file e.g. while (getline(in, s)) {...} in <
    fstream>
ofstream out(s.c_str()); // Output file e.g. out << "hello\n"; in <fstream>
const double PI=3.14159; // Not a variable, e.g. PI=4; is an error

// Functions (must be global)
int sum(int a, int b) {return a+b;}  // e.g. i=sum(j, 2); sets i=5
void swap(int& a, int& b) {          // Pass by reference, no return value
  int tmp=a; a=b; b=tmp;             // e.g. swap(i, j); must be variables
}                                    // Scope of a, b, and tmp ends here, implied
     return;
void print(const string& s) {cout << s;}  // Pass large objects by const
    reference

// Declaring new types (global or local)
typedef double Real;      // Real z; declares object z of type double
struct Complex {          // Complex a; declares a.re, a.im type double
  Real re, im;            // Data members
  Complex(double r=0, double i=0) {re=r; im=i;} // Initialization code
};
```

[7] Quelle des Beispiels http://cs.fit.edu/~mmahoney/cse1502/introcpp.html

16 Scope (Gültigkeitsbereich)

C++ ist genau wie C eine block-orentierte Sprache und in diesen Sprachen wird der Gültigkeitsbereich von Namen über die Block-Struktur bestimmt. Das Scope selbst lässt sich in einzelne Teilbereiche gliedern.[9]

Block Scope

Die Gültigkeit des klassichen „Block-Scopes“ (von daher auch Block-Orentiert) beginnt mit der Deklaration in einem Block und endet mit dem Ende des Blocks. Das Scope (dt. Gültigkeitsbereich) bleibt in den Kinderblöcken prinzipiell erhalten, sollte es aber eine neue Deklaration mit gleichen Namen in einem Unterblock geben, so wird diese die Übergeordnete überschatten.

Listing 23: **Beispiel: Block Scope erklärt**

```
int main()
{
    int a = 0; // scope of the first 'a' begins
    ++a; // the name 'a' is in scope and refers to the first 'a'
    {
        int a = 1; // scope of the second 'a' begins
                   // scope of the first 'a' is interrupted
        a = 42;    // 'a' is in scope and refers to the second 'a'
    } // block ends, scope of the second 'a' ends
      //              scope of the first 'a' resumes
} // block ends, scope of the first 'a' ends
int b = a; // Error: name 'a' is not in scope
```

Das potenzielle Scope von einem Funktionsparameter beginnt wieder an der Stelle der Deklaration und endet entweder mit dem letzten Exception-Handler eines eines try-catch Blocks oder am Ende der Funktion.

Listing 24: **Beispiel: Block Scope Funktionsparameter**

```
int f(int n = 2)  // scope of 'n' begins
try // function try block
{         // the body of the function begins
   ++n;   // 'n' is in scope and refers to the function parameter
   {
      int n = 2; // scope of the local variable 'n' begins
                 // scope of function parameter 'n' interrupted
      ++n; // 'n' refers to the local variable in this block
    }            // scope of the local variable 'n' ends
                 // scope of function parameter 'n' resumes
} catch(...) {
   ++n; // n is in scope and refers to the function parameter
   throw;
} // last exception handler ends, scope of function parameter 'n' ends
int a = n; // Error: name 'n' is not in scope
```

[9] Beispiele aus C++11 standard (ISO/IEC 14882:2011): 3.3 Scope [basic.scope]

Exception Handler spannen ein eigenes Block Scope auf

Listing 25: **Beispiel: Block Scope Exceptionhandler**

```
try {
    f();
} catch(const std::runtime_error& re) { // scope of re begins
    int n = 1; // scope of n begins
    std::cout << re.what(); // re is in scope
} // scope of re ends, scope of n ends
 catch(std::exception& e) {
    std::cout << re.what(); // error: re is not in scope
    ++n; // error: n is not in scope
}
```

Die „Range-Declaration“ in for-Schleifen sind nur im for-Loop selbst gültig.

Listing 26: **Beispiel: Block Scope for-loop**

```
Base* bp = new Derived;
if(Derived* dp = dynamic_cast<Derived*>(bp))
{
    dp->f(); // dp is in scope
} // scope of dp ends

for(int n = 0; // scope of n begins
    n < 10;    // n is in scope
    ++n)       // n is in scope
{
    std::cout << n << ' '; // n is in scope
} // scope of n ends
```

Namespace Scope

Vorweg für alle die jetzt keinen Nutzen für Namespaces haben: „Namespaces bieten ein Verfahren zur Verhinderung von Namenskonflikte in großen Projekten." Namespace Gültigkeiten beginnen stehen mit der Deklaration des Namespace Blocks und aggregieren alle „genesteden", (auch mit using), Namepsaces.

Listing 27: **Beispiel: Namespace Scope**

```
namespace N { // scope of N begins (as a member of global namespace)
    int i; // scope of i begins
    int g(int a) { return a; } // scope of g begins
    int j(); // scope of j begins
    void q(); // scope of q begins
    namespace {
        int x; // scope of x begins
    } // scope of x does not end
    inline namespace inl { // scope of inl begins
      int y; // scope of y begins
    } // scope of y does not end
} // scope of i,g,j,q,inl,x,y interrupted

namespace {
    int l=1; // scope of l begins
} // scope of l does not end (it's a member of unnamed namespace)

namespace N { // scope of i,g,j,q,inl,x,y continues
    int g(char a) {  // overloads N::g(int)
        return l+a;  // l from unnamed namespace is in scope
    }
    int i; // error: duplicate definition (i is already in scope)
    int j(); // OK: repeat function declaration is allowed
    int j() { // OK: definition of the earlier-declared N::j()
        return g(i); // calls N::g(int)
    }
    int q(); // error: q is already in scope with different return type
} // scope of i,g,j,q,inl,x,y interrupted

int main() {
    using namespace N; // scope of i,g,j,q,inl,x,y resumes
    i = 1; // N::i is in scope
    x = 1; // N::(anonymous)::x is in scope
    y = 1; // N::inl::y is in scope
    inl::y = 2; // N::inl is also in scope
} // scope of i,g,j,q,inl,x,y interrupted
```

Class Scope

Das Klassen-Scope beginnt mit der Deklaration der Klasse und spannt über den ganzen „Body“ der Klasse. Es umfasst auch alle Unterklassen, Funktionsrümpfe und Exceptionspezifikationen.

Listing 28: **Beispiel: Class Scope**

```
class X {
    int f(int a = n) { // X::n is in scope inside default parameter
         return a*n;   // X::n is in scope inside function body
    }
    int g();
    int i = n*2;   // X::n is in scope inside initializer

//  int x[n];      // Error: n is not in scope in class body
    static const int n = 1;
    int x[n];      // OK: n is now in scope in class body
};
int X::g() { return n; } // X::n is in scope in out-of-class member function body
```

Sollte etwas vor der Klassendeklaration bereits den selben Namen haben so ist das Programm „ill-formed“, hier ensteht laut C++ Spezifikation „Undefined behavior“ (wieder amoi - so großartig für Spezifikationen); aber zum Glück erkennt der Compiler das in diesem Fall.

Listing 29: **Beispiel: ill-formed Class Scope**

```
typedef int c; // ::c
enum { i = 1 }; // ::i
class X {
    char v[i]; // Error: at this point, i refers to ::i
               // but there is also X::i
    int f() {
         return sizeof(c); // OK: X::c, not ::c is in scope inside a member
             function
    }
    char c; // X::c
    enum { i = 2 }; // X::i
};

typedef char* T;
struct Y {
    T a; // error: at this point, T refers to ::T
         // but there is also Y::T
    typedef long T;
    T b;
};
```

Namen von jeden „Class Member“ können nur in folgenden Kontext genutzt werden

- In seinem eigenen Scope oder dem einer abgeleiteten Klasse
- Nach dem . Operator auf einen Ausdurck der Klasse oder einer Ableitung
- Nach dem -> Operator auf den Pointer dieser Klasse oder einer abgeleiteten Version
- Nach dem :: Operator auf die Klasse oder einer Ableitung

Enumeration Scope

Enumerationen gibt es „scoped“ und „unscoped“, am besten wird das einfach Anhand dieses Beispiels illustriert.

Listing 30: **Beispiel: Enumeration Scope**

```
enum e1_t { // unscoped enumeration
  A,
  B = A*2
}; // scope of A and B does not end

enum class e2_t { // scoped enumeration
    SA,
    SB = SA*2 // SA is in scope
}; // scope of SA and SB ends

e1_t e1 = B; // OK, B is in scope
// e2_t e2 = SB; // Error: SB is not in scope
e2_t e2 = e2_t::SB; // OK
```

Point of declaration

Scopes beginnen immer mit dem Point of Declaration, also mit der Deklaration. Für Variablen und einfache Deklaration ist der Punkt genau nach der Namensdeklaration und vor der Initialisierung

Listing 31: **Beispiel: Point of Declaration**

```
unsigned char x = 32; // scope of the first 'x' begins
{
    unsigned char x = x; // scope of the second 'x' begins before the initializer (= x)
                         // this does not initialize the second 'x' with the value 32,
                         // this initializes the second 'x' with its own, indeterminate, value
}
std::function<int(int)> f = [&](int n){return n>1 ? n*f(n-1) : n;};
           // the name of the function 'f' is in scope within the lambda, and can
           // be correctly captured by reference, giving a recursive function
```

Listing 32: **Beispiel: Point of Declaration 2**

```
const int x = 2; // scope of the first 'x' begins
{
    int x[x] = {}; // scope of the second x begins before the initializer (= {})
                   // but after the declarator (x[x]). Within the declarator, the outer
                   // 'x' is still in scope. This declares an array of 2 int.
}
```

17 String in C++

Strings sind in C++ nicht wie in C als char-Arrays umgesetzt sondern als so genannte „Streams“, an die mit dem << Operator angehängt werden kann. Die Umsetzung per-se ist zwar wesentlich komplizierter allerdings auch sicherer als die der C-Strings. Die Standardlibrary enthält einige nützliche Funktionen zum Thema strings, welche in den nachfolgenden Beispielen[10] illustriert werden.

Listing 33: **Beispiel: Strings einfache Beispiele**

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

using namespace std;

int main () {

   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int  len ;

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

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

   // total length of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```

Ausgabe

str3 : Hello
str1 + str2 : HelloWorld
str3.size() : 10

[10] Beispiele von https://www.tutorialspoint.com/cplusplus/cpp_strings.htm und http://anaturb.net/C/string_exapm.htm

Listing 34: **Beispiel: Strings Position**

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string s = "Nobody is perfect";

    // Returns s[pos]
    for ( int pos = 0; pos < s.length(); ++pos )
        cout << s.at(pos) << " ";
    cout << endl;

    return 0;
}
```

Ausgabe

```
N o b o d y i s p e r f e c t
```

Listing 35: **Beispiel: In c-String konvertieren**

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string str = "Anatoliy";
    char *ary = new char[str.length()+1];

    // strcpy ( ary, str ); that is wrong way
    strcpy ( ary, str.c_str() ); // that is correct

    cout << ary << endl;

    return 0;
}
```

Ausgabe

```
Anatoliy
```

Listing 36: **Beispiel: Strings compare**

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string str1 = "string";
    string str2 = "String";
    string str3 = "second string";
    char ch[]   = "first string";

    cout << "string str1 is : " << str1 << endl;
    cout << "string str2 is : " << str2 << endl;
    cout << "char ary ch is : " << ch   << endl;
    cout << "string str3 is : " << str3 << endl;
    cout << endl;

    // compare str1 and str2
    cout << "1." << endl;
    size_t comp = str1.compare(str2);
    cout << "String str1 is ";
    ( comp == 0 ) ? cout << "equal" : cout
         << "not equal";
    cout << " to string str2" << endl;

    // compare str1 and literal string "string"
    cout << "2." << endl;
    comp = str1.compare("string");
    cout << "String str1 is ";
    ( comp == 0 ) ? cout << "equal" : cout
        << "not equal";
    cout << " to array of char \"string\"" << endl;

    // compare str3 start from pos 7 to 5
    // with str1
    cout << "3." << endl;
    comp = str3.compare(str1,7,5);
    cout << "Part of string str3 is ";
    ( comp == 0 ) ? cout << "equal" : cout
        << "not equal";
    cout << " to str1" << endl;

    // compare str3 start from pos 7
    // with literal string "string"
    cout << "4." << endl;
    comp = str3.compare("string",7);
    cout << "Part of string str3 is ";
    ( comp == 0 ) ? cout << "equal" : cout
        << "not equal";
    cout << " to C string \"string\"" << endl;
   return 0;
}
```

Ausgabe

1. String str1 is not equal to string str2
2. String str1 is equal to array of char "string"
3. Part of string str3 is equal to str1
4. Part of string str3 is equal to C string "string"

Listing 37: **Beispiel: Strings copy**

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string str = "First Name: Robert";
    char fname[255];
    cout << "str is: " << str << endl;

    int n = str.find(':');

    str.copy(fname, // copy to array
             n+1,   // how many char
             0);    // start position from str

    // must terminate fname with '\0';
    fname[n+1] = 0;

    cout << "fname is: " << fname << endl;

    return 0;
}
```

Ausgabe

str is: First Name: Robert
fname is: First Name:

Listing 38: **Beispiel: Strings find**

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

int main ()
{
     string  str("C++ is best language");
     int pos1, pos2; // size_t or size_type
                     // work not correct
     // search for first string "best" inside of str
     // default position is 0
     pos1 = str.find ("best");
     cout << "Word best is found on position " << pos1+1
          << endl;

     // if pattern is not found - return -1
     pos2 = str.find ("best",pos1+1);
     cout << "Word best is found on position " << pos2+1
          << endl;

     // search for first occurrence of character
     pos1 = str.find('g');
     cout << "First character 'g' found on position "
          << pos1
          << endl;

     // search for first occurrence of string
     string s = "is";
     pos1 = str.find (s);
     cout << "Word 'is' is found on position " << pos1+1
          << endl;

     return 0;
}
```

Ausgabe

Word best is found on position 8
Word best is found on position 0
First character 'g' found on position 15
Word 'is' is found on position 5

18 Klassen

Struct vs Class[11]

stuct

- Daten und Operationen werden zu einer Einheit zusammengefasst
- Daten und Operationen sind öffentlich zugänglich (sichtbar)
- Enge Kopplung der Daten mit den Operationen (gleicher Gültigkeitsbereich, Daten "automatisch" in Operationen bekannt)

class

- Daten und Operationen werden zu einer Einheit zusammengefasst
- Sichtbarkeit von Daten und Methoden beliebig steuerbar

Hinweis zur Äquivalenz von struct und class:

`struct X {...};` entspricht `class X { public: ...};`
`class X {...};` entspricht `struct X { private: ...};`

Listing 39: **Beispiel: Person Implmentierung als Struct**

```
1  // This program prints my name and age.  Then it asks for your
2  // name and age and prints it, for example:
3  //
4  //   Matt is 52 years old.
5  //   What is your name? Sam
6  //   What is your age? 20
7  //   Sam is 20 years old.
8
9  #include <iostream>
10 #include <string>
11 using namespace std;
12
13 // A Person has a name and an age
14 struct Person
15 {
16   string name;
17   int age;
18 };
19
20 // Print a Person's name and age
21 void print(Person p)
22 {
23   cout << p.name << " is " << p.age << " years old.\n";
24 }
25
26 int main()
27 {
28
29   // Create some people
30   Person me, you;
31
32   // Print information about me
33   me.name = "Matt";
34   me.age = 52;
35   print(me);
36
```

[11] Beispiele von https://cs.fit.edu/~mmahoney/cse1502/tutorial/

```
37   // Get information about you and print it
38   cout << "What is your name? ";
39   cin >> you.name;
40   cout << "What is your age? ";
41   cin >> you.age;
42   print(you);
43
44   return 0;
45 }
46
47 /* Notes:
48
49 - 'struct Person' defines Person as a new type.
50 - 'me' and 'you' are objects of type Person.
51 - 'name' and 'age' are members.
52 - The members of an object are accessed using the . operator
53   using the form 'object.member'.
54 - The . operator has higher precedence than all other operators.
55   Thus 'me.age + you.age' means '(me.age) + (you.age)'.
56 - Objects can be assigned, e.g.
57
58     me = you;
59
60   is equivalent to
61
62     me.name = you.name;
63     me.age = you.age;
64
65 - Objects cannot be compared, e.g. if (me==you) is an error.
66 - A struct declaration can be global or local.  Usual scope
67   rules apply.  This example is global because it is not declared
68   inside a function.
69 - The general form of a struct declaration creating new type T is:
70
71     struct T
72     {
73       declarations...;
74     };
75
76   The declaration cannot contain any statements, expressions,
77   or initializatons, e.g.
78
79     struct MyType
80     {
81       int a, b;   // OK
82       int x = 0;  // error, can't initialize
83       cout << x;  // error, only declarations are allowed
84     };
85
86 - Don't forget ; after a struct declaration.
87 - By convention, user defined type names are Capitalized.
88 */
```

Listing 40: **Beispiel: Person Implmentierung als Class**

```
1 // This program prints my name and age.  Then it asks for your
2 // name and age and prints it, for example:
3 //
```

```
//   Matt is 52 years old.
//   What is your name? Sam
//   What is your age? 20
//   Sam is 20 years old.

#include <iostream>
#include <string>
using namespace std;

// A Person can be created with a name and age that can
// be read from the user or printed.
class Person
{
private:
  string name;
  int age;
public:
  Person(string n, int a);  // Constructor
  Person();      // Another constructor (overloaded)
  void read();  // Member function to get name and age from user
  void print()  // Member function (inlined) to print name and age
  {
    cout << name << " is " << age << " years old.\n";
  }
};

// Code for first constructor
Person::Person(string n, int a)
{
  name = n;
  age = a;
}

// Code for second constructor
Person::Person()
{
  name = "";
  age = 0;
}

// Code for read
void Person::read()
{
  cout << "What is your name? ";
  cin >> name;
  cout << "What is your age? ";
  cin >> age;
}

// End of definition of class Person

// Test code for class Person
int main()
{
  Person me("Matt", 52), you;  // Create 2 objects
  me.print();
  you.read();
  you.print();
```

```
62    return 0;
63  }
64
65  /* Notes:
66
67  - A class is a type like a struct, except that we hide the
68    implementation (member data) from the rest of the program,
69    and access the data indirectly through a public interface
70    using a set of member functions.
71
72    For example, string is a class.  We do not know the details
73    of its implementation (an array of char).  Instead we only know
74    how to use strings, e.g. input, output, .size(), .substr(), etc.
75
76    In large programs with many variables, grouping the variables
77    into classes helps organize the code, making it easier to maintain.
78    The parts of the program that need to access this data are made
79    into member functions.  The rest of the program is not allowed
80    to access this data.
81
82  - A class is like a struct but it can also contain member functions
83    (or function prototypes).
84
85  - Member functions have parameters and return types just like
86    ordinary functions, except that they are "attached" to objects.
87    For example:
88
89      me.print()
90
91    calls the member function print() in the object 'me', which is
92    of type 'Person'.  When print() executes the code 'cout << name'
93    it prints 'me.name'.
94
95  - A class is normally divided into a public and private section.
96    Only the member functions can access the private section, e.g
97
98      me.print();       // OK, print() is public
99      cout << me.name;  // error, name is private
100     you.age = 10;     // error, age is private
101
102   The program would still work if everything were public.  The
103   reason for using private data is so the compiler will enforce
104   the rule that data is only visible to the parts of the program
105   that have the need to know.  This reduces accidental programming
106   errors.
107
108 - A member function can either be written entirely in the class
109   definition (inlined), or it can be prototyped in the class and
110   the code moved outside.  It is recommended that only very small
111   (one line) functions be inlined and others be prototyped.
112
113 - When a member function is prototyped, the name of the function
114   has the form class::function, e.g. Person::read.
115
116 - The definition must agree with the prototype, just like ordinary
117   functions.
118
119 - A constructor is a special member function that is called
```

```
120   automatically when an object is created.  It is identified
121   by having the same name as the class (Person) and no return type.
122
123 - A constructor, like any function, can be overloaded.  Overloading
124   means that you can have more than one function with the same name
125   as long as you can distinguish which one you are calling by the
126   argument list.  For example:
127
128     double max(double a, double b);  // 1
129     int max(int a, int b);           // 2
130     int max(int a, int b, int c);    // 3
131
132   then max(1.5, 2.0) calls the first version, and max(3, 4, 5)
133   calls the third version.  Likewise
134
135     Person x;  // calls Person::Person()
136     Person y("Joe", 25);  // calls Person::Person(string, int)
137
138 - It is an error to call a constructor directly, e.g. me.Person()
139
140 */
```

Sichtbarkeiten[12]

Für alle Bezeichner (egal ob für Konstanten, Datentypen, Datenkomponenten oder Methoden), die in Verbunden (struct), Vereinigungen (union) oder Klassen (class) deklariert sind, kann selektiv die Sichtbarkeit geregelt werden

private nur in Methoden derselben Klasse (und in Freunden, s. u.) kann auf private Bezeichner (auch anderer Objekte dieser Klasse) zugegriffen werden

protected auch in Methoden abgeleiteter Klassen kann auf geschützte Bezeichner einer Basisklasse zugegriffen werden

public an allen Stellen (in beliebigen Methoden und Funktionen) kann auf öffentliche Bezeichner zugegriffen werden

Standardmäßig (ohne explizite andere Angabe) gilt:

- Alle Bezeichner in einem Verbund (struct) oder einer Vereinigung (union) sind öffentlich (public)
- Alle Bezeichner in einer Klasse (class) sind privat (private)

Ausnahmen für die "Privatisierung" können über Freunddeklarationen (mittels friend) für Funktionen, Methoden und Klassen erreicht werden.

[12] Quelle: Folien H.Dobler

Konstruktoren und Destruktoren[13]

- Objekte müssen vor ihrer Verwendung in einen definierten Zustand gebracht
- Konstruktoren werden automatisch aufgerufen, wenn ein Objekt einer Klasse erzeugt wird
- Destruktoren werden automatisch aufgerufen, wenn ein Objekt einer Klasse freigegeben wird
- Konstruktoren und Destruktoren dürfen keine Rückgabewerte haben
- Konstruktoren können beliebige Parameter haben und können damit überladen werden
- Destruktoren müssen immer parameterlos sein, daher keine Überladung möglich

statische Objekte Konstruktor wird bei Definition automatisch aufgerufen, Destruktor wird bei Verlassen des Blocks automatisch aufgerufen

dynamische Objekte Konstruktor wird bei Anlegen eines Objekts aufgerufen (von Operator new nach Speicherreservierung), Destruktor wird bei Freigeben eines Objekts aufgerufen (von Operator delete vor Speicherfreigabe)

Achtung: Wird Destruktor explizit aufgerufen, wird nur Destruktor ausgeführt, aber Speicher d. Objekts nicht freigegeben, also besser nicht

[13] Quelle: Folien H.Dobler

Vererbung

Prinzipiell wie Vererbung funktioniert sollte klar sein, C++ hat da allerdings zwei „Gusto-Stücke“ parat. Einmal unterstützt C++ mehrfach Vererbung (Diamant Problem!), was einfach vermieden werden sollte; es geht, es ist aber so gut wie nie eine gute Idee, anderseits Sichtbarkeit der Vererbung.[14]

Listing 41: **Beispiel: Vererbungs-Syntax**

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

Listing 42: **Beispiel: Simple Vererbung**

```
using namespace std;

//Base class
class Parent
{
    public:
      int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
      int id_c;
};

//main function
int main()
{

        Child obj1;

        // An object of class child has all data members
        // and member functions of class parent
        obj1.id_c = 7;
        obj1.id_p = 91;
        cout << "Child id is " <<  obj1.id_c << endl;
        cout << "Parent id is " <<  obj1.id_p << endl;

        return 0;
}
```

Ausgabe

Child id is 7
Parent id is 91

[14] Beispiele zu dem Thema sind von Geeks4Geeks: Harsh Agarwal

Arten der Vererbung

Hier sind wir jetzt bei einer C++ Besonderheit, es gibt 3 Arten der Vererbung. (Siehe access_mode in der Syntax)

public Wenn die abgeleitete Klasse public von der Basisklasse ableitet, dann werden alle „public member" der Basisklasse auch public in der abgeleiteten Klasse, alle „protected member" der Basisklasse werden protected in der abgeleiteten Klasse und private wird nicht vererbt.

protected Wenn wir die abgeleitete Klasse protected von der Basisklasse ableiten, dann werden sowohl public als auch protected Member der abgeleiteten Klasse protected. Private wird nicht vererbt (da-doy! :p)

private Wenn wir abgeleitete Klasse private von der Basisklasse ableiten, dann werden sowohl public als auch protected in der Abgeleiteten private. Private von oben erbt nicht (Überraschung)

	Abgeleitet		
Basisklasse	**Public Mode**	**Private Mode**	**Protected Mode**
Private	-	-	-
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Überschreiben von Methoden (overriding) [15]

Dynamisch gebundene Methode einer Basisklasse kann in abgeleiteter Klasse überschrieben werden

Invarianz Überschreibende Methode muss identische Parameterliste haben (kann aber zu einer konstanten Methode werden)

Kovarianz Nur Ergebnisdatenyp in überschreibender Methode kann auch von einer abgeleiteten Klasse sein

Überladen von Methoden (Operator Overloading)

Listing 43: Beispiel: PI Operator Overloading

```
1  // This program prints pi to 1000 decimal places using the Spigot
2  // algorithm on a new class of high precision fixed point numbers.
3
4  #include <iostream>
5  #include <string>
6  using namespace std;
7
8  // A fixed point number represents number that have the same range
9  // as an integer (-2147483648 to 2147483647) but with n decimal
10 // places of precision, where the user specifies n.  n defaults
11 // to 1000.  Supported operations are +=, *=, /= by integers
12 // and <<  (output).  There are implicit conversions from int to
13 // 1000 decimal place numbers and explicit conversions to any range
14 // of the form Fixed(x, n), where x is the  value and n is the number
15 // of decimal places.
16
17 class Fixed
18 {
19 private:
20   int i;     // integer part, e.g. 3 in 3.141592
21   string f;  // fractional part as a string of digits, e.g. "141592"
22 public:
23   Fixed(int x=0, int n=1000); // convert to x with n decimal zeros
24   void operator += (int a) {i+=a;}  // add a
25   void operator *= (int a);   // multiply by a
```

[15] Quelle: Folien H.Dobler

```
26   void operator /= (int a);   // divide by a
27   void print(ostream& out);   // print to out as a decimal number
28 };
29
30 Fixed::Fixed(int x, int n)
31 {
32   i = x;
33   f = string(n, '0');
34 }
35
36 void Fixed::operator *= (int a)
37 {
38   int carry = 0;
39   for (int j=int(f.size())-1; j>=0; --j)
40   {
41     int d = (f[j]-'0')*a + carry;
42     f[j] = d%10 + '0';
43     carry = d/10;
44   }
45   i = i*a + carry;
46 }
47
48 void Fixed::operator /= (int a)
49 {
50   int carry = i%a;
51   i /= a;
52   for (int j=0; j<int(f.size()); ++j)
53   {
54     int d = carry*10 + f[j]-'0';
55     carry = d%a;
56     f[j] = d/a + '0';
57   }
58 }
59
60 void Fixed::print(ostream& out)
61 {
62   out << i << "." << f;
63 }
64
65 ostream& operator << (ostream& out, Fixed& f)
66 {
67   f.print(out);
68   return out;
69 }
70
71 // Compute pi using Spigot algorithm
72 int main()
73 {
74   Fixed pi = 4;
75   for (int i = 3322; i > 0; --i)
76   {
77     pi *= i;
78     pi /= i * 2 + 1;
79     pi += 2;
80   }
81   cout << pi << endl;
82   return 0;
83 }
```

```
84
85 /* Notes:
86
87 - Overloading operators for numeric types allows a more natural notation.
88   For example, you could replace 'Fixed pi = 4' with 'double pi = 4'
89   and the program would still work (but only print 6 digits).
90
91 - An operator is really just a function using a different notation.  Thus,
92
93     a += b
94
95   could be rewritten either as a member function call,
96
97     a.operator += (b)
98
99   or as a global (nonmember) function,
100
101    operator += (a, b)
102
103  If += does not already have a meaning, then you can give it one by
104  writing a function called 'operator+=' using one of these two forms
105  (but not both).
106
107  Generally this means you can define an operator as long as either
108  a or b is a user defined class type, for example:
109
110    void Fixed::operator += (int b);  // as a member
111    void operator += (Fixed a, int b);  // as a global function
112
113  Because only member functions can access the private data in Fixed,
114  we use this method.
115
116 - You cannot overload operators that already have a meaning, for example,
117
118    void operator += (int a, int b);  // error, a+=b already defined for int
119
120 - Defining +=, -=, *=, /= does not automatically define +, -, *, /.
121   You need to overload those too.  Thus if you wanted to write
122
123    pi = pi + 2;
124
125  You would need to overload + taking Fixed as the left operand and int
126  as the right operand.  This could be done as a global function without
127  accessing the private data in Fixed:
128
129    Fixed operator + (Fixed a, int b)
130    {
131      a += b;
132      return a;
133    }
134
135 - Operators should be overloaded to produce expected behavior.  Users
136   expect addition to be commutative, but:
137
138    pi = 2 + pi;  // error, no operator + (int, Fixed)
139
140  So you would also need:
141
```

```
142     Fixed operator + (int a, Fixed b)
143     {
144       b += a;
145       return b;
146     }
147
148 - To reduce the number of overloaded operators, you can use implicit
149   conversion and overload only the highest type.  For example, given
150   the conversions int -> double -> Fixed, you need only define
151
152     Fixed::Fixed(double);  // constructor for double -> Fixed
153     Fixed operator + (Fixed, Fixed);  // accepts int or double
154
155   which will handle any mixed mode arithmetic containing at least
156   one Fixed, e.g.
157
158     pi = 2 + pi;  // implicit pi = Fixed(double(2)) + pi;
159
160 - Output can be handled by overloading operator <<.  The left
161   operand (cout) has type ostream.  It must be overloaded as a
162   global function because class ostream is part of the standard
163   library and we can't add a member function.  cout should be
164   passed by reference and returned by reference.  The general form
165   to print x, which has type T is:
166
167     ostream& operator << (ostream& out, T x)
168     {
169       // print x to out (not to cout)
170       return out;
171     }
172
173   out has to be returned because
174
175     cout << pi << endl;
176
177   really means
178
179     (cout << pi) << endl;  // What does endl print to?
180
181   The return by reference means that (cout << pi) is the original
182   cout and not a temporary copy with the same value.  You can't copy
183   cout, so it can only be passed or returned by reference.
184
185 - Fixed::print() is needed because operator << is not a member of
186   Fixed and needs to print its private data.
187
188 */
```

19 Smart Pointer

Der Smartpointer verhält sich ähnlich wie der normale Pointer, allerdings löscht er automatisch seinen Speicherbeich wenn er „out of scope“ geht. [16]

Listing 44: Beispiel: Smartpointer Implementierung

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

class SmartPtr
{
   int *ptr;  // Actual pointer
public:
   explicit SmartPtr(int *p = NULL) { ptr = p; }
   ~SmartPtr() { delete(ptr); }
   // Overloading dereferencing operator
   int &operator *() {  return *ptr; }
};

int main()
{
    SmartPtr ptr(new int());
    *ptr = 20;
    cout << *ptr;

    // We don't need to call delete ptr: when the object
    // ptr goes out of scope, destructor for it is automatically
    // called and destructor does delete ptr.

    return 0;
}
```

In der C++ STL gibt es fertige Implmentierung

auto_ptr

Lieftert ein Template für Smartpointer ähnlich dem obigen Beispiel, Teil der STL. Wurde von unique_pointer in <memory> abgelöst. Hat aber folgende Einschränkungen: Bei Zuweisung eines auto_ptr-Objekts dest = src; geht Eigentümerschaft an dest über, src ist dann "leer" (nullptr). Nicht für Felder von statischen Objekten möglich, denn delete wird im Destruktor von auto_ptr immer ohne [] aufgerufen.

unique_pointer<T> (statt auto_ptr)

für Objekte, die nur über einen einzigen Zeiger referenziert werden, mit autom. Speicherfreigabe bei Blockende

Listing 45: Beispiel: unique pointer

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

using namespace std;
class X{
    public:
    void m();
};
```

[16] https://www.geeksforgeeks.org/smart-pointers-cpp/

```
9
10 int main()
11 {
12     unique_ptr<X> up1(new X); // X has meth. m
13     up1->m();
14     unique_ptr<X> up2;
15     up2 = move(up1); // up2 gains ownership ...
16     up2->m(); // ... and up1 is "empty"
17     return 0;
18 }
```

shared_ptr<T>

für Objekte, die von mehreren Zeigern (shared_ptr) referenz. werden können, mit Referenzenzähler (RZ) u. autom. Speicherfreigabe wenn RZ = 0 wird

Listing 46: **Beispiel: shared ptr**

```
1  #include<iostream>
2  #include<memory>
3
4  using namespace std;
5  class X{
6      public:
7      void m();
8  };
9
10 int main()
11 {
12     shared_ptr<X> sp1(new X);
13     sp1->m(); // call X::m()
14     shared_ptr<X> sp2;
15     sp2 = sp1; // now sp1 and sp2 refer. ...
16     sp2->m(); // ... to the same object
17     sp1 = nullptr; // same as sp1.reset()
18     sp2 = nullptr; // deletes the X object
19 }
```

Listing 47: **Beispiel: Listenknoten freigeben mit shared pointer**

```
1  #include<iostream>
2  #include<memory>
3
4  using namespace std;
5  class Node { // Node for singly-linked lists: unique_ptr could be used
6      public: // but shared_ptr allows component sharing (see ex.)
7          string val;
8          shared_ptr<Node> next;
9          Node(string val, shared_ptr<Node> next = nullptr) {
10             this->val = val;
11             this->next = next; // better: init. list
12         } // Node
13         virtual ~Node() {
14             cout << "destructing Node with val " << val << endl;
15         } // ~Node
16 }; // Node
17
```

```
int main(int argc, char *argv[]) {
    cout << "BEGIN" << endl;
    shared_ptr<Node> l1(new Node("3"));
    l1.reset(new Node("2", l1));
    shared_ptr<Node> l2(new Node("1.2", l1));
    l1.reset(new Node("1.1", l1));
    cout << "delete l1:" << endl;
    l1 = nullptr; // deletes node with "1.1"
    cout << "END" << endl;
    return 0; // l2 with all remainig Nodes ...
} // main // ... deleted automatically
```

weak_ptr<T>

Listing 48: **Beispiel: weak ptr**

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

using namespace std;
class X{
    public:
    void m();
};

int main()
{
    shared_ptr<X> sp(new X);
    sp->m();
    weak_ptr<X> wp;
    wp = sp; // now wp & sp refer to same object
    wp.lock()->m(); // works
    sp = nullptr; // deletes the X object
    wp.lock()->m(); // error: nullptr deref.
}
```

20 RAII

Ressourcenbelegung ist Initialisierung, meist abgekürzt durch RAII, für englisch resource acquisition is initialization, bezeichnet eine verbreitete Programmiertechnik zur Verwaltung von Betriebsmitteln (auch Ressourcen genannt). Dabei wird die Belegung von Betriebsmitteln an den Konstruktoraufruf einer Variablen eines benutzerdefinierten Typs und die Freigabe der Betriebsmittel an dessen Destruktoraufruf gebunden. Die automatische Freigabe wird beispielsweise durch das Verlassen des Gültigkeitsbereichs ausgelöst (am Blockende, bei Ausnahmeauslösung, durch Rückgabe an den Aufrufer usw.), der implizite Destruktoraufruf der Variablen sorgt dann für die Wiederfreigabe der Ressource

21 RTTI

RTTI **Run-time Type Identification**

Erlaub das ermitteln eines Types zur Laufzeit. RTTI stellt zwei Operatoren zur Verfügung und mithilfe des dynamic_cast Operators kann sich in der Vererbeungshirache nach oben gearbeitet werden. Der typeid Operator erlaub es den typen eines Objektes festzustellen.

Listing 49: **Beispiel: RTTI**

```
#include <iostream>
#include <typeinfo> // Header for typeid operator
using namespace std;

// Base class
class MyBase {
  public:
    virtual void Print() {
        cout << "Base class" << endl;
    };
};

// Derived class
class MyDerived : public MyBase {
  public:
    void Print() {
        cout << "Derived class" << endl;
    };
};

int main()
{
    // Using typeid on built-in types types for RTTI
    cout << typeid(100).name() << endl;
    cout << typeid(100.1).name() << endl;

    // Using typeid on custom types for RTTI
    MyBase* b1 = new MyBase();
    MyBase* d1 = new MyDerived();

    MyBase* ptr1;
    ptr1 = d1;

    cout << typeid(*b1).name() << endl;
    cout << typeid(*d1).name() << endl;
    cout << typeid(*ptr1).name() << endl;

    if ( typeid(*ptr1) == typeid(MyDerived) ) {
    cout << "Ptr has MyDerived object" << endl;
    }

    // Using dynamic_cast for RTTI
    MyDerived* ptr2 = dynamic_cast<MyDerived*> ( d1 );
    if ( ptr2 ) {
    cout << "Ptr has MyDerived object" << endl;
    }
}
```

Ausgabe

```
i
d
6MyBase
9MyDerived
9MyDerived
Ptr has MyDerived object
Ptr has MyDerived object
```

22 Rule of Three / Rule of Five

Die Regel der Drei

...wenn **Copy Constructor**, **Destructor** oder **Copy Assignment Operator** als explicit deklariert wird, muss man mit hoher Sicherheit alle Drei als explicit deklarieren.

Original: The Rule of Three claims that if one of these had to be defined by the programmer, it means that the compiler-generated version does not fit the needs of the class in one case and it will probably not fit in the other cases either.

Die Regel der Fünf

Durch die C++ 11 Erweiterung in Bezug auf die „move-semantic“ wird die Regel der Drei erweitert auf:

- destructor
- copy constructor
- move constructor
- copy assignment operator
- move assignment operator

Einführung / Problemstellung[17]

```
1  class Person
2  {
3      private:
4          string name;
5          int age;
6      public:
7          person(const string& name, int age) : name(name), age(age)
8          {
9          }
10 };
11
12 int main()
13 {
14     Person a("Bjarne Stroustrup", 60);
15     Person b(a);    // What happens here?
16     b = a;          // And here?
17 }
```

Die Grundfrage

die sich stellt ist, was heißt es eine “Kopie” eines Objektes anzulegen, die vom Compiler generierte „Kopier-Funktionalität“ kopiert einen Pointer / eine Referenz in dem sie einfach die Speicheradresse kopiert, dies führt dazu das die Kopie nur eine Referenz auf das selbe Objekt ist - was wir aber in diesem Fall wollen ist eine autarke Kopie des Objektes - genauer der Datenkomponenten, das oben angeführte Beispiel zeigt 2 “Kopier-Szenarien” einerseits, den Kopier-Konstruktor **Person b(a)**, dieser hat die Aufgabe eine neues Objekt aus den Daten des bestehenden Objektes zu konstruieren andererseits den Kopier-Zuweisungsoperator **b = a**, dieser ist schwieriger zu realisieren da sich das Objekt auf der linken Seite bereits in einem gültigen Zustand befindet.

[17] Geklaut aus der eigenen Aufarbeitung :p

Nachdem

wir weder Kopier-Konstruktor noch den Assignment Operator definiert haben werden diese implizit vom Compiler gestellt:

Auszug aus dem C++ Standard (Section 12 §1)

The [...] copy constructor and copy assignment operator, [...] and destructor are special member functions. [Note: **The implementation will implicitly declare these member functions for some class types when the program does not explicitly declare them**. The implementation will implicitly define them if they are used ...

Implizite Definition

```
1  // 1. copy ctor
2  Person(const person& that) : name(that.name), age(that.age)
3  {
4  }
5
6  // 2. copy assignment operator
7  Person& operator=(const person& that)
8  {
9      name = that.name;
10     age = that.age;
11     return *this;
12 }
13
14 // 3. destructor
15 ~Person()
16 {
17 }
```

Die „Drei“ implizit definiert.

Die Kopie klont unsere Datenkomponenten und legt uns eine neue unabhängige Person an, der Destruktor bleibt leer. Das ist das „Geschenk des Compilers“ - das passiert wenn man diese „Special Member Functions“ nicht selbst angibt. Alles „Fun and Games“ bis zu dem Moment wo eine Datenkomponente nicht ein einfaches „Primitive“ bzw. Skalar ist. Und wie wir inzwischen wissen: Regel der Drei/Fünf, sobald wir einen anlegen müssen - müssen (oder zumindest sollten :P) wir die anderen auch anlegen.

Auszug aus dem C++ Standard (Section 12.8)

§16 The implicitly-defined copy constructor for a non-union class X performs a memberwise copy of its subobjects.

§30 The implicitly-defined copy assignment operator for a non-union class X performs memberwise copy assignment of its subobjects

Dynamische Datenobjekte

Hier fängt jetzt die Krux an, nehmen wir das selbe Beispiel - nur anstatt eines Strings nehmen wir jetzt, der „Einfachheit“ halber, ein char-Array.

```
class Person
{
    char* name;
    int age;
    public:
    // the constructor acquires a resource:
    // in this case, dynamic memory obtained via new[]
    Person(const char* the_name, int the_age)
    {
        name = new char[strlen(the_name) + 1];
        strcpy(name, the_name);
        age = the_age;
    }

    // the destructor must release this resource via delete[]
    ~Person()
    {
        delete[] name;
    }
};
```

Problem

Hier wird wieder „member-wise“ kopiert, das heißt aber bei dem Char-Array, dass der Skalar-Wert kopiert wird und dieser beinhaltet nicht die Daten auf der Adresse sondern die Adresse selbst. Nun teilt sich die Kopie die entsteht einen Speicherbereich mit dem Original und wir bekommen unerwünschtes Verhalten.

1. Alle Änderungen des Namens von Person a wird ist automatisch auch in dem kopierten Objekt Person b.
2. Sobald b zerstört wird, ist der Pointer in a ein ungültiger Pointer
3. Einen ungültigen Pointer erneut löschen erzeugt undefinierbares Verhalten.
4. Memory Leaks über Zeit

Explizite Definition

Nachdem wir nun das Problem der implicit Kopie („member-wise cloning“) kennen und eben dieses Verhalten nicht wollen müssen wir einen Kopier-Konstruktor und einen „Copy-Assignment-Operator“ bzw Kopier-Zuweisungsoperator selbst anlegen um dieses Problem zu beheben.

```
1  // 1. copy constructor
2  Person(const person& that)
3  {
4      name = new char[strlen(that.name) + 1];
5      strcpy(name, that.name);
6      age = that.age;
7  }
8
9  // 2. copy assignment operator
10 Person& operator=(const person& that)
11 {
12     if (this != &that)
13     {
14         delete[] name;
15         // This is a dangerous point in the flow of execution!
16         // We have temporarily invalidated the class invariants,
17         // and the next statement might throw an exception,
18         // leaving the object in an invalid state
19         name = new char[strlen(that.name) + 1];
20         strcpy(name, that.name);
21         age = that.age;
22     }
23     return *this;
24 }
```

Achtung

Hier wird nun auch der oben erwähnte Komplexitätsunterschied zwischen Kopier-Konstruktor und Kopier-Zuweisungsoperator sichtbar. Wir müssen im Kopier-Zuweisungsoperator prüfen ob es sich nicht um das selbe Objekt handelt, da wir sonst das Quell-Array zerstören würden und nicht wieder herstellen könnten. Außerdem müssen wir das Char-Array löschen um keine Memory Leaks zu produzieren, da dieses, anders als beim Kopier-Konstrukt bereits existiert. Was an dem oberen Beispiel immer noch ein Problem darstellen kann, ist wenn der Speicher zum erneuten Reservieren des Arrays nicht mehr ausreicht, hier kann es durch das fangen der geworfenen Exception passieren, dass das Objekt in einem ungültigen Zustand hinterlassen wird.

Kopierschutz

Wenn man nun nicht möchte, dass das Objekt kopiert werden kann so setzt man den Kopier-Zuweisungsoperator und den Kopierkonstruktor einfach private ohne Implmentierung:

```
1 private:
2     Person(const person& that);
3     Person& operator=(const person& that);
```

C++ 11 aufwärts

unterstütz eine eigene Semantik um diesen Kopierschutz umzusetzen:

```
1 public:
2     Person(const person& that) = delete
3     Person& operator=(const person& that) = delete
```

Erweiterung „move-semantics“

Ab C++11 sind die so genannten „move-semantics“ dazu gekommen, diese ermöglichen es einem anderen Objekt einem anderen Daten zu „stehlen“. Zusammen mit den Drei vorher genannten „Special Member Functions“ bilden die zwei Weiteren, Move-Konstruktor und Move-Zuweisungsoperator, die **Regel der Fünf**.

```
// 1. Move Ctor
Person&(Person&& that)
{
    //if its unsure that that.name is initialized
    //also add a check for nullptr
    name = that.name;
    age = that.age;
    //'stolen' data
    that.name = nullptr;
    //'stolen' scalar value gets its 'default'
    that.age = 0;
}

// 2. Move-Assignment Operator for Person
Person& operator=(Person&& that)
{
    if (this != &that)
    {
        delete[] name;
        name = that.name;
        age = that.age;
        // we have 'stolen' the data
        that.name = nullptr;
        that.age = 0;
    }
    return *this;
}
```

23 Schlüsselwörter Index

Für eine Schnellübersicht siehe III auf Seite 56

Übernommen von C

auto const double float int short struct unsigned break continue else for long signed switch void case default enum goto register sizeof typedef volatile char do extern if return static union while

Neu in C++

asm dynamic_cast namespace reinterpret_cast try bool explicit new static_cast typeid catch false operator template typename class friend private this using const_cast inline public throw virtual delete mutable protected true wchar_t

Ab C++11

and bitand compl not_eq or_eq xor_eq and_eq bitor not or xor

Vordefinierte Namen

cin endl INT_MIN iomanip main npos std cout include INT_MAX iostream MAX_RAND NULL string

24 Implmentierungen

Quickselect (C)

Listing 50: Beispiel: Quickselect C Implmentierung

```c
/* * * * * * * * * * *
 2017, Caspar Jan

* * * * * * * * * * */

/**
 * @brief helper function to swap elements in array
 *
 * @param l left part for swap
 * @param r right part for swap
 */
void swap(int *l, int *r){
    int temp = *l;
    *l = *r;
    *r = temp;
}

/**
 * @brief quicksort rec
 *
 * @param a array
 * @param left left-side
 * @param right right-side
 * @param i i-th element to be found
 * @return int i-th largest element
 */
int quicks(int a[], int left, int right, int i){
    int px = a[right];
    int p = left;

    for(int x = left; x < right; x++){
        if(a[x] > px){
            swap(&a[p],&a[x]);
            p++;
        }
    }
    swap(&a[p], &a[right]);

    if(p == (i-1)){
        return a[p];
    }else if((i-1) < p){ // has to be left
        return quicks(a,left,p-1,i);
    }else{ //has to be right
        return quicks(a,p+1,right,i);
    }
}

```

```
/**
 * @brief retuns the ith largest object from a array
 * based on a quickselect
 * @param a array
 * @param n size of array
 * @param i i-th element to be returned
 * @return int i-th element from array
 */
int quickselect(int a[], int n, int i){
    int l = quicks(a,0,n-1,i);
    return l;
}
```

Binary Search

Listing 51: **Beispiel: Binary Search**

```
int binary_search(int arr[], int start, int end, const int key)
{
    // Termination condition: start index greater than end index
    if(start > end)
    {
        return -1;
    }

    // Find the middle element of the array and use that for splitting
    // the array into two pieces.
    const int middle = start + ((end - start) / 2);

    if(arr[middle] == key)
    {
        return middle;
    }
    else if(arr[middle] > key)
    {
        return binary_search(arr, start, middle - 1, key);
    }

    return binary_search(arr, middle + 1, end, key);
}
```

Knapsack[18]

Listing 52: **Beispiel: Knappsack**

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
```

[18] https://www.geeksforgeeks.org/knapsack-problem/

```
int knapSack(int W, int wt[], int val[], int n)
{
    // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then
    // this item cannot be included in the optimal solution
    if (wt[n-1] > W)
        return knapSack(W, wt, val, n-1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                     knapSack(W, wt, val, n-1)
                   );
}

// Driver program to test above function
int main()
{
     int val[] = {60, 100, 120};
     int wt[] = {10, 20, 30};
     int  W = 50;
     int n = sizeof(val)/sizeof(val[0]);
     printf("%d", knapSack(W, wt, val, n));
     return 0;
}
```

Matrix transponieren[19]

Listing 53: Beispiel: Matrix transponieren

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

int main()
{
    int a[10][10], trans[10][10], r, c, i, j;

    cout << "Enter rows and columns of matrix: ";
    cin >> r >> c;

    // Storing element of matrix entered by user in array a[][].
    cout << endl << "Enter elements of matrix: " << endl;
    for(i = 0; i < r; ++i)
    for(j = 0; j < c; ++j)
    {
        cout << "Enter elements a" << i + 1 << j + 1 << ": ";
        cin >> a[i][j];
    }

    // Displaying the matrix a[][]
```

[19] https://www.programiz.com/cpp-programming/examples/matrix-transpose

```
21      cout << endl << "Entered Matrix: " << endl;
22      for(i = 0; i < r; ++i)
23          for(j = 0; j < c; ++j)
24          {
25              cout << " " << a[i][j];
26              if(j == c - 1)
27                  cout << endl << endl;
28          }
29
30      // Finding transpose of matrix a[][] and storing it in array trans[][].
31      for(i = 0; i < r; ++i)
32          for(j = 0; j < c; ++j)
33          {
34              trans[j][i]=a[i][j];
35          }
36
37      // Displaying the transpose,i.e, Displaying array trans[][].
38      cout << endl << "Transpose of Matrix: " << endl;
39      for(i = 0; i < c; ++i)
40          for(j = 0; j < r; ++j)
41          {
42              cout << " " << trans[i][j];
43              if(j == r - 1)
44                  cout << endl << endl;
45          }
46
47      return 0;
48  }
```

Teil III. Schnellerklärung aller C++ Schlüsselwörter[20]

and alternative to && operator

and_eq alternative to &= operator

asm insert an assembly instruction

auto declare a local variable, or we can let the compiler to deduce the type of the variable from the initialization.

bitand alternative to bitwise & operator

bitor alternative to | operator

bool declare a boolean variable

break break out of a loop

case a block of code in a switch statement

catch handles exceptions from throw

char declare a character variable

class declare a class

compl alternative to ~ operator

const declare immutable data or functions that do not change data

const_cast cast from const variables

continue bypass iterations of a loop

default default handler in a case statement

#define All header files should have #define guards to prevent multiple inclusion.

delete make dynamic memory available

do looping construct

double declare a double precision floating-point variable

dynamic_cast perform runtime casts

else alternate case for an if statement

enum create enumeration types

exit() ending a process

explicit only use constructors when they exactly match

export allows template definitions to be separated from their declarations

extern declares a variable or function and specifies that it has external linkage

extern "C" enables C function call from C++ by forcing C-linkage

false a constant representing the boolean false value

float declare a floating-point variable

for looping construct

friend grant non-member function access to private data

goto jump to a different part of the program

if execute code based on the result of a test

inline optimize calls to short functions

int declare an integer variable

long declare a long integer variable

mutable override a const variable

namespace partition the global namespace by defining a scope

new allocate dynamic memory for a new variable

not alternative to ! operator

not_eq alternative to != operator

operator create overloaded operator functions

or alternative to || operator

or_eq alternative to |= operator

private declare private members of a class

protected declare protected members of a class

public declare public members of a class

register request that a variable be optimized for speed

reinterpret_cast change the type of a variable

short declare a short integer variable

signed modify variable type declarations

sizeof return the size of a variable or type

static create permanent storage for a variable

static_cast perform a nonpolymorphic cast

struct define a new structure

switch execute code based on different possible values for a variable

template create generic functions

this a pointer to the current object

throw throws an exception

true a constant representing the boolean true value

try execute code that can throw an exception

typedef create a new type name from an existing type

typeid describes an object

typename declare a class or undefined type

union a structure that assigns multiple variables to the same memory location

unsigned declare an unsigned integer variable

using import complete or partial namespaces into the current scope

virtual create a function that can be overridden by a derived class

void declare functions or data with no associated data type

volatile warn the compiler about variables that can be modified unexpectedly

void declare functions or data with no associated data type

wchar_t declare a wide-character variable

while looping construct

xor alternative to ^ operator

xor_eq alternative to ^= operator

[20] http://www.bogotobogo.com/cplusplus/cplusplus_keywords.php

Programm-Listings