1 Aufbau eines Programmes

```
#include <iostream> // Standart In-/
    Output stream
#include <vector>   // Vector library
#include <cmath>    // Für math.
    funktionen
#include <time.h>   // Zeitmessung
#include "headerfile.h" // Einbiden
    Headerfile
#define N 10  // defines jeglicher art

//structs, functions, enums

int main(void)
{
//programm code
return 0;
}
```

2 Variablen

Group	Type names	Notes on size / precisio
Character types	char	One byte 8 bits.
	char16_t	At least 16 bits
	char32_t	At least 32 bits.
	wchar_t	Largest character set
Integer types (signed)	signed char	Min 8 bits.
	signed short int	Min 16 bits.
	signed int	Min 16 bits.
	signed long int	Min 32 bits.
	signed long long int	Min 64 bits.
Integer types (unsigned)	unsigned char	Min 8 bits.
	ßhort int	Min 16 bits.
	ïnt	Min 16 bits.
	"long int	Min 32 bits.
	"long long int	Min 64 bits.

Mögliche Initialsisations von vaiablen

```
int x;
int x = 1;
int x (1);
int x {1};
```

2.1 Pointer und Referenzen als Rückgabewert und Parameterübergabe

Bei Variablenübergabe (call by value) werden Kopien übergeben, welche nicht verändert werden können.
Bei Referenzübergabe (call by reference) kann die Subroutine die Werte bleibend verändern.

Objekte einer Klasse und Strukturvariablen sollen immer by reference übergeben werden!

2.2 call by reference

```
void swap(int& a, int& b)
{
int tmp = a;
a = b;
b = tmp;
}
int main()
{
int x = 4;
int y = 3;
swap(x, y); // OK!
return 0;
}
```

```
void swap(int* a, int* b)
{
int tmp = *a;
*a = *b;
*b = tmp;
}
int main()
{
int x = 4;
int y = 3;
swap(&x, &y); // OK!
return 0;
}
```

2.3 call by value

```
void swap(int a, int b)
{
int tmp = a;
a = b;
b = tmp;
}
int main()
{
int x = 4;
int y = 3;
swap(x, y); // keine Ausw.
return 0;
}
```

3 Funktionen

Funktionen sind Unterprogramme, die häufig verwendeten Code enthalten.
Ein Beispiel:

```
int add (int a, int b);   //Prototyp

//PRE: a, b > 0
//POST: true, wenn eine das doppelte
  der anderen ist
bool timestwo (int a, int b){
  bool c=false;
  return a==add(b, b) || b==add(a,a);
}
```

Rückgabewert ist immer genau **ein** Variabeltyp (Workaround: Structs). Ohne Rückgabewert schreibt man void.

3.1 Aufbau

```
rückgabewert funktionsname (argument){
  funktionskörper
  return ;
}
```

3.2 Pre- und postconditions

Preconditions beschreiben den Input der Funktion, Postcondition den Output und die Wirkung der Funktion. Preconditions prüft man mit
`assert (a>0 && b>0)`

3.3 Prototyp und Gültigkeitsbereiche

Falls eine Funktion g, die Funktion f benötigt muss diese vorab definiert sein, da sich der Gültigkeitsbereich einer Funktion nur unterhalb seiner Defintion befindet. Die formalen Argumente verhalten sich wie Variabeln und haben nur einen Lokalen Gültigkeitsbereich im Funktionsblock.

3.4 Rekursion

Wenn eine Funktion sich selber wieder auruft, nennt man das Rekursion. Dabei muss es eine Abbruchbedingung geben, die auch erreicht wird. Dann wird von innen aufgelöst.

```
int fak (int n){
  if(n==1) return 1;
  return n* fak(n-1);
}
```

4 Pointer und Referenzen

Bei Variablenübergabe (call by value) werden Kopien übergeben, welche nicht verändert werden können.
Bei Referenzübergabe (call by reference) kann die Subroutine die Werte bleibend verändern.
Objekte einer Klasse und Strukturvariablen sollen immer by reference übergeben werden!

4.1 call by reference

statisch:

```
void swap(int&
a, int& b){
  int tmp = a;
  a = b;
  b = tmp;
}
int main(){
  int x = 4;
  int y = 3;
  swap(x, y);//
 OK!
  return 0;
}
```

dynamisch:

```
void swap(int*
a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
int main(){
  int x = 4;
  int y = 3;
  swap(&x, &y);
// OK!
  return 0;
}
```

4.2 call by value

```
void swap(int a, int b){
  int tmp = a;
  a = b;
  b = tmp;
}
int main(){
  int x = 4;
  int y = 3;
  swap(x, y); // keine Auswirkung
  return 0;
}
```

4.3 return by reference

```
int& inc(int& i){
  return ++i;
}
```

Der Funktionsaufruf ist nun selbst ein L-Wert, was nun Ausdrücke wie `inc(inc(x))` oder `++inc(x)` erlaubt. **Achtung** Gültigkeitsbereiche: Return by reference auf lokale Variable ist undefined behavior. //**Edit sobald Pointer in Vorlesung**

5 Vektoren

Vektoren dienen zum Speichern gleichartiger Daten.

5.1 Initialisierung

```
std::vector<int> vec(3);
//{0, 0, 0}
std::vector<int> vec(4, 2);
//{2, 2, 2, 2}
std::vector<int> vec{4,3,2,1};
//{4, 3, 2, 1}
std::vector<int> vec;
//leerer Vektor
```

5.2 Zugriff

Das erste Element eines Vekotrs hat index 0. Ein Zugriff auf Elemente ausserhalb der gültigen Grenze führt zu undefinierten Verhalten. C++ bietet eine optionale überprüfung.

```
std::vector<int> vec(3);
vec.at(3) = 1;  //Error compiler
vec[3] = 1; //undefined behaviour
```

5.3 Befehle

```
std::vector<int> vec{0,1};
vec.size(); //Länge des Vektors: 2
vec.push_back(3) //hängt wert an:
  {0,1,3}
vec.clear(); //löscht Inhalt : {0,0,0}
vec.resize(2); //ändert Grösse: {0,0}
```