enum
```
enum wochentag {montag, dienstag};
```

Strukturen
```
struct person {
    string name;
    void ausgabe() {cout << „Name = „ << name;}}
```

Statisch
```
class kreis {
 public:
  double radius;
  static double PI; //Deklaration innerhalb der Klasse
  static const double T_DEF_AND_DEKL=1.0;
  static const double T_EXTRA_DEKL;
  static const char *desc; // static const string desc;
 ~~static void b() const {}~~
  static double bogenlaenge(kreis &k, double bogen
  { return this->bogen * ... }
  kreis() {}
  ~kreis(){}
};
double kreis::PI = 3.0; //Definition
const double kreis::T_EXTRA_DEKL = 1.0;
const char *kreis::desc = "KREIS";
//const std::string kreis::desc = "KREIS";
int main() { kreis k; }
```

Headerdatei
```
#ifndef header_h
#define header_h
//Class deklaration
#endif
```

Templates
```
template <typename T>
T max(T a, T b) {…}
```
Klassen Template Stack.h:
```
template<class T>
class Stack {...};
```
Definition Stack.h (nicht cpp):
```
template<class T>
T Stack<T>::pop() {}
```
Instanziierung (whatever. cpp):
```
Stack<int> i; main.cpp
```

Operatorüberladung (Zusammenfassung S 526)
```
std::ostream& operator<< ( ostream& s, Person& x) {
s << x.Name << endl; return s; }
```

Postfix/Präfix
```
@aa Präfixoperator: complex & complex::operator++()
{ return *this ; }
aa@ Postfixop.: complex complex::operator++(int)
{ return tmp; }
```

Verdeckung
```
obj.print() → Abgeleitete Klasse
obj.basis::print() → Basisklasse
```
using (Mehrfachvererbung)
```
class TelefonMitAnrufbeantworter : public Telefon, public
Anrufbeantworter
{public: using Phone::displayTime;};
```

Zufallszahl zwischen 1 und 0
```
#include <ctime>
srand( time(NULL) );//Initialisierung Zufallszahlengen.
zahl = static_cast<float>( rand() ) / RAND_MAX;
```

friend-Deklarationen
```
friend-Deklarationen werden nicht vererbt
class demo1 {
  friend class demo2; …
  friend int func(demo &objekt); //Globale Methode
};
```

Inline
```
Angabe bei Deklaration
```

Funktionsdeklarationen
```
int f( int i );
//void f( int i );
//char f( int i ) ;
int f( int & i );
int f( int i, int k );
int f( char c, int k );
//int f( int i, int k = 0 );
int f( double d, int i, char c = 'a' );
int f( double d, double e = 1.1, int i = 0 );
//int f( double d, double e = 1.1, char c );
int f( char c, char d, int i = 2, char e = 'c' ):
```

Referenz als Rückgabewert
```
int& minimum1(const int &a,const int &b );
~~minimum1(2,3);~~
int a=1, b=2; minimum(a,b);
```

Funktionsaufrufe
```
class kreis { public:
  static kreis* groesserer_kreis(kreis *a, kreis *b); }; [...]
kreis *k1 = new kreis();  kreis k2;
cout << k1->groesserer_kreis(k1, &k2) <<endl; //[...]
```

Pointer
```
int i = 10;
int *ptr1 = &i;
int *ptr2 = ptr1;
int feld[3] = {20, 21, 22};
ptr2 = feld;
int ptr3 = feld[1];
printf("ptr1 = %d, ptr2 = %d\n", *ptr1, *ptr2);
```

Objekterzeugung
```
Stack:
• die Variable (hier c) mit Speicherplatz für ein Objekt der
Klasse angelegt
• ein Objekt der Klasse erzeugt (Konstruktoraufruf) und
• im Speicher der Variablen abgelegt.
Heap:
• Speicherplatz für ein Objekt der Klasse angelegt
• ein Objekt der Klasse erzeugt (Konstruktoraufruf)
• im Speicher der Variablen abgelegt
• die von new gelieferte Speicherdresse in Variabler (hier
pc) gespeichert
```

Heap
```
cin >> laenge; int* array;
array = new int[laenge];
array = (int*) malloc(sizeof(int) * laenge)
```

Default-Konstruktoren
```
kreis(); //Konstruktor
~kreis(); //Destuktor
kreis(const kreis &k); //Kopierkonstruktor
  demo obj3 = obj1; //Kopierkonstruktor
  demo obj3( obj1 ); //Kopierkonstruktor
```

Zuweisungsoperator (Objekte kopieren)
```
class Ba{}; class Ab : public Ba {};
Ba basis; Ab abl;
basis = abl; //Zuweisungsoperator (Deklaration vorher)
~~abl = basis;~~
```

Konstruktor/Destruktor
```
Head/Stack:  Basiskonstruktor → Ableitungkonstruktor
Stack:  Ableitungsdestruktor → Basisdestruktor
Heap:
ohne virt. Des.: Basisklassendestruktor
mit virt. Des.: Ab.sdestruktor → Basisklassendestruktor
```

Ohne virtual in Basisfkt bei Vererbung
```
Stack: Basisfunktion
Heap: Basisfunktion
```
Mit virtual in Basisfkt bei Vererbung
```
Stack: Basisfunktion
Heap: Abgeleitete Fkt (späte Bindung)
```

Abstrakt
```
Deklaration abstrakte Methode: virtual void fkt() = 0;
Abstrakte Klasse : Enthält min. eine abstrakte Methode
```

Virtual
```
class basis {
  public:
   void func() { cout << "Basis" << endl; }
   basis() { cout << "Kon. Basis" << endl; }
   ~basis() { cout << "Des. Basis" << endl; }
};
class abgeleitet : public basis {
  public:
   void func() { cout << "Ab." << endl; }
   abgeleitet() { cout << "Kon. Ab." << endl; }
   ~abgeleitet() { cout << "Des. Ab." << endl; }
};
class vBasis {
  public:
   virtual void func() { cout << "Basis" << endl; }
   vBasis() { cout << "Kon. Basis" << endl; } //Kein virt.
   virtual ~vBasis() { cout << "Des. Basis" << endl; }
};
class vAbgeleitet : public vBasis {
  public:
   virtual void func() { cout << "Ab." << endl; }
   vAbgeleitet() { cout << "Kon. Ab." << endl; }
   virtual ~vAbgeleitet() { cout << "Des. Ab." << endl; }
};
int main() {
 { //Identisch fuer vBasis,vAbleitung
   abgeleitet abObj;//Kon Basis, Kon Ab.
   basis baObj = abObj;//flache Kopie
   baObj.func(); //Basis
   basis *baPtr = &abObj;
   baPtr->func();//Basis
 }//Des. Basis baObj, Des. Ab. abObj, Des. Basis abObj
 { basis * b[2];
   b[0] = new basis();//Kon. Basis
   b[1] = new ableitung();//Kon Basis, Kon Ab.
   b[1]->func(); //Basis
   delete b[0];//Des Basis
   delete b[1];//Des Basis
  }
 { vBasis * vb[2];
   vb[0] = new vBasis();//Kon. Basis
   vb[1] = new vAbgeleitet();//Kon Basis, Kon Ab.
   vb[1]->func(); //Ab.
   delete vb[0];//Des Basis
   delete vb[1];//Des Ab., Des Basis
 }
}
```