C++ Programmierung

Prof. Dr. Uwe Wienkop

C++ Gliederung

- Dynamische Strukturen
- Programmierparadigmen
- Klassen
- Typen und Deklarationen
- Zeiger und Strukturen
- Funktionen
- Überladen von Operatoren
- Abgeleitete Klassen
- Namensbereiche
- Templates
- Klassenhierarchien
- Ausnahmebehandlung

Einschübe:

- Einstieg "Graphikprogrammierung"
- Quelldateien und Programme
- Umgang mit größeren Programmierprojekten (make)
- Versionskontrollsysteme (Beispiel SCCS)
- Ereignisgesteuerte Programmierung / MFC
- Standardbibliothek

Organisatorisches

- **Literatur:**
  - Die C++ Programmiersprache, Bjarne Stroustrup, Addison-Wesley, 1998, ISBN 3-8273-1296-5 (sehr empfehlenswert!)
  - Weiter mit C++, Roman R. Gerike, Heise, 1992, ISBN 3-88229-009-9
  - Diverse Tutorials, Newsgroups im WWW (siehe meine WWW-Intranet-Seite)
- **Unterlagen:**
  - alle schriftlichen Unterlagen/Folien können kopiert werden
- **Sprechstunde**
  - Freitags, 9:45-11:00 Uhr, nach Anmeldung
    eMail: Uwe.Wienkop@fh-nuernberg.de
- **Erwartungen:**
  - Aktive Teilnahme an Vorlesung und Übung
  - Aktives eigenständiges Programmieren (mind. 1 Übungsaufgabe pro Rubrik!)
  - Auseinandersetzen mit den Übungsaufgaben, zumindest Fragen formulieren können!
  - Gemeinsam Spaß beim Studium von C++ haben

Abschnitt I

- Wdhg. Dynamische Strukturen
- Programmierparadigmen
- **Klassen**
- C++ Erweiterungen
  (Typen und Deklarationen, Zeiger und Strukturen, Funktionen)

Wiederholung C

- **Schleifen (for, while-do, do-while, continue, break)**
- **Fallunterscheidungen (if-then-else, switch-case, bedingte Auswertung)**
- **Datentypen (char, int, unsigned, long, float, double)**
- **Sichtbarkeiten/Speicherklassen (static, auto, extern, register)**
- **Parameterübergabe an Funktionen (call-by-value, call-by-ref.)**
- **Strukt. Datentypen: Strukturen, enum, union, Felder, typedef**
- **Pointer, P.-Arithmetik, Verändern von P.-Objekten, P. auf Strukturen**
- **Ein- / Ausgabe (stream, formatierte E/A [scanf, printf, cout, cin])**
- **Präprozessor (#define, #include, #if - #else - #end)**
- **C-Compiliervorgang: Compile+Link, Dateitypen .c, .h, .o**
- **Guter Programmierstil, Programmierrichtlinien**

- **Aufgaben:**
  - Die einzelnen Konstrukte kurz vorstellen.
  - Was macht man mit dem Konstrukt? Was ist zu beachten? Fehlermöglichkeiten?

Statische vs. dynamische Objekte

- **Statische Objektdefinitionen**
  - char ch;
  - unsigned char xx = 130;
  - char *text = "Dies ist ein Text";
  - short sbuf[] = { 1, 5, 10, 15};
  - int j = 1;
  - double *dp = &d;
  - unsigned ui = 2000000;
  - long list[10] = { 1, -2, -5, 10, 15 };
  - int feld[10][20];

- **... sind über Namen aufrufbar**
- **... kommen im Programmablauf nur einmal vor**
- **... Jede Instanz bzw. die Anzahl der Instanzen (bei Feldern) muß zur Compile-Zeit bekannt sein**

malloc & new

- **malloc - alte, d.h. C-Form der Speicheranforderung**
  void *malloc(size_t size);
  - char *buf = malloc(1000); // 1000 Bytes werden angefordert
  - struct adresse {
    char strasse[30];
    int hausnr;
    };
  - adresse *meine_adresse = malloc(sizeof(adresse)); // besser:
  - adresse *meine_adresse = (adresse *) malloc(sizeof(adresse));
- **Freigabe mit void free(void *memblock);**

- **C++ Form der Speicheranforderung: new**
  [::] new [placement] type-name [initializer]
  - adresse *meine_adresse = new adresse;
  - adresse *adr_feld = new adresse[10];
- **Freigabe mit delete**
  - delete meine_adresse;

Verwaltung dynamischer Objekte durch eine lineare Liste

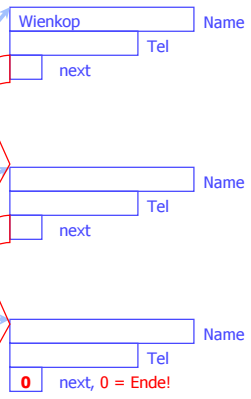
```
struct Adr_t            // Definition der Adreßstruktur
{
   char    Name[30];
   char    Tel[20];
   Adr_t   *next;
};

Adr_t   *Anf;           // Aufbauen einer linearen Liste:
Adr_t   *Laufz;
Adr_t   *aktuell;

Anf = new Adr_t;        // Erzeugen eines ersten Objekts
strcpy(Anf->Name, "Wienkop" );
aktuell = new Adr_t;    // Erzeugen des zweiten Objekts

Anf->next = aktuell;    // Verbinden von erstem und zweiten Element

aktuell = new Adr_t;    // Erzeugen des dritten / aller weiteren Elemente
Laufz = Anf;            // Suche nach dem letzten Element
while (Laufz->next)     // Solange es noch weitere Elemente gibt:
   Laufz = Laufz->next; // Geh' zum nächsten Element
                        // Sonst: Letztes Element gefunden:
Laufz->next = aktuell; // dieses mit dem aktuellen Element verbinden
```



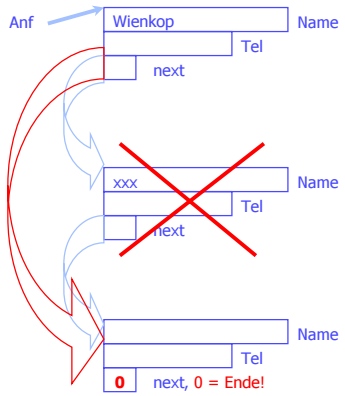
Löschen eines Objekts der linearen Liste

Aufgabe: Löschen eines Elements (Name ist angegeben) aus der linearen Liste
Die Variable 'Anf' zeigt auf den Anfang der Liste

```
int lösche(char *ElemName)
{
    Adr_t       *Laufz = Anf;
    Adr_t       *tmp;

    if (strcmp(Anf->Name, ElemName) == 0)
    {                       // Ist schon das erste Element das gesuchte?
        Anf = Anf->next;
        delete Laufz;
        return 1;
    }

    while (Laufz->next &&       // Nach benanntem Element suchen!
           strcmp(Laufz->next->name, ElemName) != 0)
        Laufz = Laufz->next;    // Wichtig! Laufz muß eins vor dem
                                // gesuchten Element stehen bleiben!
    if (Laufz->next)
    {
        tmp = Laufz->next;
        Laufz->next = Laufz->next->next; // Element ausklinken
        delete tmp;
        return 1;
    }
    else
        return 0;
}
```



Ein-/Ausgabe in C++

- **C**
  - #include <stdio.h>
  - scanf("%d %lf %f", &i, &d, &f);
  - printf("%d %f %f", i, d, f);
- **C++**
  - #include <iostream.h> evtl. #include <iostream>
  - cin >> i >> d >> f;
  - cout << i << d << f;
- **cin (console in)**
  - >> "Schieben" der Tastatureingabe in die nachfolgend aufgeführten Variablen
  - kein Adreßoperator und keine Typangabe nötig; cin "weiß", welchen Typ die Variablen haben und wie die Eingabe somit zu interpretieren ist
- **cout (console out)**
  - << "Schieben" der Variablenwerte auf die Konsole
  - Typangabe wiederum nicht nötig
  - Ausgabefeldbreite, Hex.-Ausgabe etc. können über Manipulatoren eingestellt werden

Ein-/Ausgabe in C++
Beispiele

- **Beispiel aus Übungsaufgabe "Annuitätentilgung"**
  - #include <iostream> // benötigt u.a. für Std.-Manipulatoren ohne Parameter
  - #include <iomanip> // benötigt für Std.-Manipulatoren mit Parameter, s.u.
  - using namespace std;
  - cout << setw(2) << Jahr << " | "<< setw(5) << rd <<" DM | "<< setw(3) << JZ <<" DM | " << setw(3) << JT << " DM" << endl;

- **Standard-Manipulatoren (Auswahl, s. B.S. pp. 670 ff.)**
  - setprecision(int n) - n Ziffern nach dem Dezimalpunkt für alle weiteren Ausgaben
  - setw(int n) - nächstes Feld hat n Zeichen
  - setfill(int c) - c als Füllzeichen verwenden
  - left, right - links- bzw. rechtsbündig formatieren
  - setbase(int b) - Integer zur Basis b ausgeben
  - dec, hex, oct - alle weiteren Ausgaben zur gewählten Zahlenbasis
  - fixed - Gleitkommaformat dddd.dd
  - scientific - wissenschaftliches Format d.dddEdd
  - endl - "\n" - Zeilenende

Was gefällt / gefällt nicht an C ???

- **Gut ...**
  - Ermöglicht systemnahe Programmierung
  - Effizienter Code
  - Eigenschaften des Programms sind deterministisch (i.e. Laufzeit, Garbage-Collection)
  - Eigene Ansicht: Häufig Möglichkeit zur eleganten Formulierung eines Algorithmus
- **Gefällt nicht ...**
  - Flache Programmstruktur (keine explizite Kennzeichnung von Unterprogrammen)
  - Wesentliche Datentypen (Bool, String, const) fehlen
  - Modularisierung nur auf Dateiebene (global, static), jedoch kein Modulkonzept
  - Keine benutzerdefinierten Datentypen
  - Initialisierungsfunktionen müssen explizit aufgerufen werden (Gefahr bei Vergessen)
  - Engere Kopplung von Manipulationsfunktionen an Strukturen wäre wünschenswert
  - Datenmanipulation an Strukturen auch ohne Zugriffsfunktionen jederzeit möglich
  - Alle Funktionen brauchen einen eindeutigen Namen (auch gleichartige!) --> add_item_list, add_item_fifo, add_item_btree, ...
  - In Makros kein Schutz vor Typfehlern, Fehler sind sehr schwer zu finden, etc.
  - Keine Behandlung von Ausnahmen (Fehlerfälle, Überlauf, Unterlauf, etc.)
  - Strukturierungs- / Modularisierungsmöglichkeiten nicht besonders gut

Übersicht C++ (1)

- **Flache Programmstruktur (keine lokalen Unterprogramme)**
  - Bleibt auch in C++, jedoch wesentlich bessere Gliederung durch Klassenkonzept!
- **Bessere Modularisierungsmöglichkeiten**
  - Möglichkeit der Zuordnung zu Namensbereichen, Klassen, Strukturen (auch mehrstufig)
- **Wesentliche Datentypen (Bool, String, const) ergänzt!**
  - Bool nun vorhanden
  - typisierter const nun vorhanden
  - String, Liste, Queue über Standardbibliothek
- **Unterstützung benutzerdefinierter Datentypen, die sich ähnlich wie eingebaute Typen verwenden lassen**
  - Bei C++ über Klassen und struct die Möglichkeit, eigene Datentypen zu definieren
  - Durch Operator-Overloading können Operatoren wie +, =, [ ], etc. auf diesen eigenen Typen definiert werden

Übersicht C++ (2)

- **Automatischer Aufruf von Initialisierungsfunktionen (d.h. man wird nicht mehr vergessen, sie aufzurufen!)**
  - Klassen und struct können mit einer Initialisierungsfunktion versehen werden, die bei der Erzeugung eines solchen Objekts automatisch aufgerufen wird
  - Versehentliches Vergessen des Aufrufs – d.h. Arbeiten mit uninitialisierten Objekten – ist nicht mehr möglich
- **Engere Kopplung von Manipulationsfunktionen an Strukturen**
  - Klassen und struct können in C++ direkt die Manipulationsfunktionen enthalten
  - Funktionsname besteht aus Klassenname::Funktionsname (Strukturierung und Hierarchisierung)
- **Datenmanipulation an Strukturen jenseits der Manipulationsfunktionen kann vollständig unterbunden werden**
  - Für Klassen und struct können private und öffentliche Bereiche angegeben werden
  - Daten und Funktionen in öffentlichen Bereichen können von außen aufgerufen bzw. manipuliert werden
  - Daten in privaten Bereichen sind gegen Zugriff von außen geschützt
  - private Funktionen können von außen nicht aufgerufen werden

Übersicht C++ (3)
Beispiel

```
class/struct Stack
{
private:
   int     *buf;
   int     top;
   int     max_size;

public:
   Stack(int s = 10); // Konstruktor
   ~Stack();          // Destruktor

   void    push(int data);
   int     pop();
};
```

- **Datenmanipulation nur über Manipulationsfunktionen**
  - private / öffentliche Bereiche
- **Explizite Initialisierungsfunktionen, sogenannte Konstruktoren**
- **Enge Kopplung von Manipulations-funktionen an Strukturen**
  - push ist als Stack::push abgespeichert, d.h. untrennbar von seiner Klasse

Übersicht C++ (4)

- **Kennzeichnung von Anwender- und Implementierungsschnittstelle**
  - bei Namensbereichen können mehrere Schnittstellen definiert werden, so daß ein Anwender, die für ihn wichtigen Funktionen leicht erkennen kann. Eine weitere Schnittstelle kann dann die Implementierung beschreiben
  - Für Klassen/Struct Möglichkeit zur Auszeichnung öffentlicher und privater Bereiche
- **Funktionen brauchen keinen eindeutigen Namen, sondern eindeutige Charakterisierung anhand der Aufrufparametertypen ("Überladen")**
  - Der Überlade-Mechanismus von C++ ermöglicht, nicht nur den Funktionsnamen, sondern auch die übergebenen Parametertypen zu berücksichtigen.
    bool add_item(list &Item); → ruft Listenverarbeitungsfunktion add_item
  - bool add_item(fifo &Item); → dto. für Fifos
  - bool add_item(btree &Item); → dto. für BTrees
  - Entscheidend sind nur die Parametertypen, **nicht** der Rückgabewerttyp!

Übersicht C++ (5)

- **Verbesserung gegenüber C-Makros (Erkennen von Typfehlern, leichteres Erkennen von Syntaxfehlern ...)**
  - C++ bietet eine inline-Funktion, wodurch dem Compiler geraten wird, diese Funktion direkt in-line zu erzeugen
  - inline-Funktionen sehen ansonsten genauso aus wie "normale" Funktionen, bieten somit Syntaxprüfung, Typprüfung, formale Parameter etc.
- **Behandlung von Ausnahmen (Fehlerfälle, Überlauf, Unterlauf, etc.)**
  - C++ bietet hier ein geschlossenes Konzept mit try – throw und catch.
  - Hiermit können Fehlerzustände eindeutig identifiziert und vor allem von der richtigen Bearbeitungsinstanz aufgefangen werden
- **Strukturierungs- / Modularisierungsmöglichkeiten**
  - C++ bietet hier eine Vielzahl an Verbesserungen. Durch Klassen/structs mit Zugriffsfunktionen und Operatoren wird eine bessere Kapselung von Daten erreicht
  - Vererbung ermöglicht eine gute Strukturierung bei gleichzeitiger Erweiterbarkeit
  - Namensbereiche verhelfen zu besserer Modularisierung des Programms

Programmierparadigmen

- **... ein Stil, eine Leitlinie, auf welche Weise ein Problem mit Hilfe einer Programmiersprache gelöst wird**
  - "Eine Sprache unterstützt einen Programmierstil, falls sie Sprachmittel bereitstellt, mit denen dieser Stil bequem (angemesssen einfach, sicher und effizient) angewendet werden kann.
  - Eine Sprache unterstützt eine Technik nicht, falls solche Programme nur mit außergewöhnlichem Aufwand oder Wissen geschrieben werden können; in dem Fall ermöglicht sie nur die Technik (Beispiel: Objektorientierte Prog. in C)
- **Es gibt nicht das "beste Paradigma" !**
  - Jedes Paradigma hat seine Berechtigung und Anwendungsgebiete, in denen es herausragende Eigenschaften (angemessen einfach, sicher und effizient) zeigt.
  - !Achtung! Die Qualität eines Paradigmas zeigt sich nicht so sehr in kleinen Beispielen von wenigen Zeilen Code, sondern in großen Projekten mit vielen Entwicklern und einigen hunderttausend Zeilen Code
- **Wesentlich ist, mit allen Paradigmen (Sprachmitteln, s.o.!) vertraut zu sein und diese je nach Anforderung zielstrebig einsetzen zu können**

Paradigmen
... und weshalb sind/könnten sie sinnvoll (sein)?

- **Prozedurales Programmieren**
  - Entscheiden Sie, welche Prozeduren Sie benötigen; Verwenden Sie den besten Algorithmus, den Sie finden können
- **Modulares Programmieren**
  - Entscheiden Sie, welche Module Sie haben wollen. Unterteilen Sie das Programm so, daß die Daten in Modulen gekapselt sind
- **Datenabstraktion / Benutzerdefinierte Typen**
  - Entscheiden Sie, welche Typen Sie benötigen;
    Erstellen Sie für jeden Typ einen vollen Satz an Operationen
- **Objektorientiertes Programmieren**
  - Entscheiden Sie, welche Klassen Sie brauchen;
    Erstellen Sie für jede Klasse einen vollständigen Satz an Operationen;
    Verdeutlichen Sie Gemeinsamkeiten durch den Einsatz von Vererbung
- **Generische Programmierung**
  - Entscheiden Sie, welche Algorithmen sie benötigen. Parametrisieren Sie sie so, daß sie für eine Vielzahl von geeigneten Typen und Datenstrukturen arbeiten

Paradigmen
Prozedurales Programmieren

Entscheiden Sie, welche Prozeduren Sie benötigen;
Verwenden Sie den besten Algorithmus, den Sie finden können

- **Beispiel Quadratwurzel:**

```
double sqrt(double arg)
{
   // Code zum Berechnen der Quadratwurzel
   return Ergebnis;
}
```

- **Zu lösende Programmieraufgabe wird durch eine Hierarchie von Funktionen bzw. Funktionsaufrufen beschrieben**
  - Dieser Stil eignet sich sehr gut für arithmetische Berechnungen sowie auch für sonstige Berechnungen, in denen aufgrund von gegebenen Eingabeparametern entsprechende Rückgabewerte bestimmt werden sollen.
- **Themen in der Literatur**
  - verschiedene Arten der Argumentübergabe
  - Unterscheidungsmöglichkeiten verschiedener Argumentarten
  - unterschiedlichste Funktionsarten (z.B. Prozeduren, Routinen und Makros) usw.

Paradigmen
Modulares Programmieren (1)

Entscheiden Sie, welche Module Sie haben wollen.
Unterteilen Sie das Programm so, daß Daten in Modulen gekapselt sind.

- **Schwerpunkt des Programmdesigns: Organisation der Daten**
- **Ein Satz verwandter Prozeduren zusammen mit den von ihnen manipulierten Daten wird häufig als Modul bezeichnet (-> Datenkapselung)**

- **Beispiel: Definition eines Stacks (Wesentliche Probleme):**
  - Bereitstellung einer **Schnittstelle** zur Benutzung des Stacks (z.B. Funktionen wie push() und pop())
  - Sicherstellen, daß auf die **Repräsentation** des Stacks (z.B. ein Feld aus Elementen) nur über die **definierte Schnittstelle** zugegriffen werden kann.
  - Sicherstellen, daß der Stack for der ersten Benutzung **initialisiert** wird

Modulares Programmieren (2)
`namespace { ... }`

- **Daten, Strukturen und Funktionen werden in einem gemeinsamen Namensbereich zusammengefaßt**
- **Anwenderschnittstelle:**

```
namespace Stack  // Anwenderschnittstelle zeigt nur für Anwender notwendige Informat.
{
    void   push(int);
    int    pop();
}
```

- **Implementierungsschnittstelle:**

```
namespace Stack  // Implementierungsschnittstelle zeigt Implementierungsdetails
{                // auch doppelte Deklaration erlaubt, solange identisch!!!
    const int  max_size = 200;
    int        v[max_size];
    int        top = 0;

    void  push(int);        // Implementierung: Stack::push()
    int   pop();            // Aufruf analog!
}
```

Übungsaufgabe: FIFO/Stack-Implementierung

- **Schreiben Sie ein Programm, daß ein FIFO oder einen Stack für Zeichenketten der max. Länge von 30 Zeichen implementiert.**
  - Die zu implementierenden FIFO und Stack sollen jeweils 12 Einträge speichern können. Im Fehlerfall ist "Speicherüberlauf" bzw "Speicherunterlauf" auszugeben.
  - Implementieren Sie die folgenden Funktionen:
    - int FifoIn(char *) // Schreibt die Zeichenkette in das FIFO; liefert 1 bei Erfolg
    - int FifoOut(char *) // Kopiert die Daten aus dem Fifo in den übergebenen Speicherbereich; Liefert wieder 1 bei Erfolg
    - int Push(char *) // dto.
    - int Pop(char *) // dto.

- **main (Im Hauptprogramm sollen in einer Schleife Befehle abgefragt werden)**
  - FI - Abfrage eines weiteren Textes und Einspeichern per FifoIn
  - FO - Ausgeben des nächsten Eintrags aus dem FIFO
  - SI/SO - dto. für Stackoperationen
  - E - Ende der Schleife

Datenabstraktion / Benutzerdefinierte Typen

Entscheiden Sie, welche Typen Sie benötigen;
Erstellen Sie für jeden Typ einen vollen Satz an Operationen.

- **Schwerpunkte:**
  - Instanziierung von Modulen, nicht mehr nur eine Instanz!
  - von Daten hin zu intuitiv verwendbaren Datentypen
  - Gleichzeitig: Sichere Kennzeichnung von öffentlichen und privaten Teilen

```
class Stack
{
   int         *buf;
   int         top;
   int         max_size;

public:
   Stack(int s = 10);    // Konstruktor
   ~Stack();             // Destruktor

   void push(int data);  // Operationen
   int  pop();
};
```

Objektorientiertes Programmieren

Entscheiden Sie, welche Klassen Sie brauchen;
Erstellen Sie für jede Klasse einen vollständigen Satz an Operationen;
Verdeutlichen Sie Gemeinsamkeiten durch den Einsatz von Vererbung.

- Problem: Erweiterung von bestehenden benutzerdef. Datentypen
- Objektorientierung: ein bereits definierter Typ bildet die Grundlage für einen neuen Typ (Vererbung, Ableitung)
- Weitere Elemente und Elementfunktionen können hinzugefügt werden (Hierarchie, Vererbungsbaum, Mehrfachvererbung möglich)

```
class Form
{
    Position    Mitte;
    Farbe       Col;
    // ...
public:
    Position    Where() { return Mitte; }
    void        MoveTo(Position To)
       { Mitte = To; /* ... */ Zeichne(); }
    virtual     void  Zeichne() = 0;
    virtual     void  Drehe() = 0;
};
```

Ableitung

```
class Kreis : public Form
{
    int     radius;
public:
    void    Zeichne() { /* ... */ }
    void    Drehe() {} // Null-Funktion
};
```

Generische Programmierung

Entscheiden Sie, welche Algorithmen sie benötigen.
Parametrisieren Sie sie so, daß sie für eine
Vielzahl von geeigneten Typen und Datenstrukturen arbeiten

- Beispiel: integer-Stack; float, double, string, etc. - Stack ???
- Implementierung ist bis auf die Typen identisch!
- Template-Konzept (Typinfos werden bei der Definition des DT mitangegebenen)
- Der Compiler generiert dann den benötigten Stacktyp

```
template <class T> class Stack
{                          // T: Stacktyp!
   T            *buf;
   int          top;
   int          max_size;

public:
   Stack(int s = 10);   // Konstruktor
   ~Stack();            // Destruktor
   void push(T data);
   T    pop();
};
```

Verwendungsbeispiele:

```
Stack<char>          sc;
Stack<complex>       scompl;
Stack< list<int> >   sli;
```

Typen und Deklarationen

Bekannte Typen aus C (1)

Frage: Weshalb sind in C Variablen typisiert???

- **char, unsigned char, Zeiger auf ~**
  - char ch;
  - unsigned char xx = 130;
  - char text[] = "Dies ist ein Text";
  - char hallo[] = {'H', 'a', 'l', 'l', 'o' };
  - char *bufp = text;
  - char *jahreszeiten[] = {"Fruehjahr", "Sommer", "Herbst", "Winter");
  - Größe eines char z.B. bei Visual C ???
- **short, unsigned short, Zeiger auf ~**
  - short sh = 10;
  - short lsh1 = 35000;
  - unsigned short lsh2 = 35000;
  - short sbuf[] = { 1, 5, 10, 15};
  - short *psbuf = &sbuf[0];
  - Größe eines short z.B. bei Visual C ???

Bekannte Typen aus C (2)

- **int, unsigned int, long, unsigned long, Zeiger auf ~**
  - int i;
  - int j = 1;
  - unsigned ui = 2000000;
  - long list[10] = { 1, -2, -5, 10, 15 };
  - int feld[10][20];
  - int *lp1 = feld;
  - int *lp2 = &feld[1][0];
  - int *lp3 = feld+20;
  - Größe eines int/long z.B. bei Visual C ??? (char, short, int und long --> limits.h)

Bekannte Typen aus C (3)

- **float, Zeiger auf float**
  - float f = 3;
  - float g = 3.0;
  - float h = 3.0f;
  - float exp(float, float);
  - Größe eines float z.B. bei vc ???
- **double, Zeiger auf double**
  - double d = 3.0;
  - double *dp = &d;
  - double *add(int, int);
  - Größe eines double??? (float, double --> float.h)
- **Speichergrößen:**
  - **1 = sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)**
  - **1 <= sizeof(bool) <= sizeof(long)**
  - **sizeof(char) <= sizeof(wchar_t) <= sizeof(long)**
  - **sizeof(float) <= sizeof(double) <= sizeof(long double)**
  - **sizeof(N={short, int, long}) = sizeof(signed N) = sizeof(unsigned N)**

Boolesche Werte

- **Neuer Datentyp bool: Explizite Werte true und false**
  - Aber auch: 0 ~ false, sonst ~ true
  - Dies gilt insbesondere für Zeiger: NULL ~ false, sonst ~ true
- **Beispiele:**

```
bool    a = true;
bool    b = false;
void f(int a, int b)
{
    bool b = a == b;
    bool c = a+b;
    // ...
}
bool    isdigit(char ch)
{
    return (ch >= '0' && ch <= '9');
}
```


Aufzählungstypen (enumeration --> enum)

- **Beispiel:**
  - enum {ASM, AUTO, BREAK }; // unbenannte Aufzählung
  - enum keywords {ASM, AUTO, BREAK }; // benannte Aufzählung
- **Jede Aufzählung ist ein unterschiedlicher Typ! Der Typ eines Enumerators ist seine Aufzählung (z.B. AUTO ist vom Typ keyword)**
- **Ein Enumerator kann durch einen konstanten Ausdruck eines integralen Typs (z.B. int) initialisiert werden.**
  - enum e1 {dunkel, hell}; // Bereich 0:1
  - enum e2 {a=3, b=9}; // Bereich 0:15
  - enum e3 {min= -10, max=1000000}; // Bereich –1048576:1048575
- **Konvertieren von Werten eines integr. Typs in einen Aufzählungstyp:**
  - enum flag {x=1, y=2, z=4, e=8}; // Bereich 0:15
  - flag f1 = 5; // Typfehler: 5 ist nicht vom Typ flag
  - flag f2 = flag(5); // OK: flag(5) ist vom Typ flag und im Bereich von flag
  - flag f4 = flag(99); // undefiniert: 99 ist nicht im Bereich von flag
- **Aufzählung ist ein benutzerdefinierter Typ**
  - Möglichkeit zur Def. eigener Operationen wie etwa ++ und <<

Konstanten

- **C++ : benutzerdefinierte Konstante: const** ~~**#define Konstante 55**~~
- **Verwendungsmöglichkeiten**
  - bei vielen Objekten ändert sich der Wert nach der Initialisierung nicht mehr
  - häufig wird über Zeiger gelesen, aber nie geschrieben
  - die meisten Funktionsparameter werden gelesen aber nicht verändert
  - Konstanten führen zu besser wartbarem Code als direkt im Code eingesetzte Literale
- **Da der Konstante nicht zugewiesen werden kann, muß eine Konstante initialisiert werden. Beispiel:**

```
const int   model = 90;          // model ist eine Konstante
const int   v[]= {1, 2, 3, 4};   // v[i] ist eine Konstante
const int   x;                   // Fehler: keine Initialisierung
```

- **const beschränkt die Möglichkeiten, wie ein Objekt benutzt werden kann, anstatt anzugeben, wie die Konstante angelegt werden soll**

```
void g(const X *p)
{   // *p kann hier nicht verändert werden, p jedoch sehr wohl!
}
model = 200;        // Fehler: model ist eine Konstante
```


Konstanten (2)
Konstante Zeiger und Zeiger auf Konstante

- **Verwendung von Zeigern:** **beteiligte Objekte:**
  - der Zeiger selbst — X const* p --> konstanter Zeiger
  - das Objekt auf das er zeigt — const X *p --> Zeiger auf eine Konstante

```
void f1(char *p)
{
    char      s[] = "Gorm";

    const char *pc = s;          // Zeiger auf Konstante
    pc[3] = 'g';                 // Fehler: pc zeigt auf Konstante
    pc = p;                      // OK

    char *const cp = s;          // Konstanter Zeiger
    cp[3] = 'a';                 // OK
    cp = p;                      // Fehler: cp ist konstant

    const char *const cpc = s;   // Konstanter Zeiger auf Konstante!
    cpc[3] = 'a';                // Fehler: cpc zeigt auf Konstante
    cpc = p;                     // Fehler: cpc ist konstant
}
```

Referenzen (1)

- **Referenz ist ein alternativer Name für ein Objekt; Anwendungen:**
  - Angabe von Argumenten und Rückgabewerten für Funktionen/überladene Operatoren
  - Schreibweise X& bedeutet Referenz auf X.
- **Referenz muß initialisiert werden. Beispiel:**

```
int        i = 1;
int        &r1 = i;        // OK: r1 initialisiert
int        &r2;            // Fehler: Initialisierer fehlt
extern     int  &r3;       // OK: r3 wird anderswo initialisiert
```

- **! Die Initialisierung einer Referenz ist etwas völlig anderes als eine Zuweisung an sie. Kein Operator arbeitet mit einer Referenz. Beispiel:**

```
void g()
{
   int     ii = 0;
   int     &rr = ii
   rr ++;                  // ii wird auf 1 inkrementiert
   int     *pp = &rr;      // pp zeigt auf ii!
}
```

- **In einigen Fällen kann ein Compiler eine Referenz wegoptimieren, so daß es zur Laufzeit kein Objekt gibt, das die Referenz repräsentiert!**

Referenzen (2)

- **Anwendungsbeispiel: Call-by-reference**
  - Semantik der Argumentübergabe ~ Initialisierung, daher wird beim Aufruf das Argument i von inc() ein weiterer Name für den Aufrufparameter.

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

- **Beispiel: Assoziative Liste**
  - Die grundlegende Idee ist, daß der string einen assoziierten Gleitkommawert hat. Es soll eine Funktion werte() definiert werden, die eine Datenstruktur mit einem Paar für jeden unterschiedlichen string, der ihr übergeben wurde, verwaltet. Beispiel:

```
struct Paar
{
   string  name;
   double  wert;
};
```

Referenzen
Beispiel Assoziative Liste

```
double &werte(const string &s)
// verwaltet Menge von Paar: suche nach s, liefere den Wert, falls gefunden
// andernfalls erzeuge ein neues Paar und liefere den Standardwert 0
// Erhöht in diesem Fall die globale Variable paare_anzahl
{
   for (int i=0;i<paare_anzahl; i++)
      if (s == paare[i].name)
   return paare[i].wert;

   Paar p = {s, 0};
   paare[paare_anzahl] = p;
   return paare[paare_anzahl++].wert;
}
```

- **Die Funktion werte() findet für ihr Argument das zugehörige Gleitkomma-objekt (nicht den Wert!); sie liefert dann eine Referenz darauf zurück.**

```
string     buf;
while (cin >> buf)
   werte(buf)++;     // Inkrementiert die von werte() gelieferte Referenz
```

- **Achtung: Mit den Referenzen ist es möglich, daß ein Funktionsaufruf auf der linken als auch auf der rechten Seite einer Zuweisung verwendet wird.**

Referenzen
Beispiel: Verwendung bei Zeigern

```
#include <iostream>

void f(char * test)                | f(char * &test)
{
   char *p = "Ach so, na dann...!";

   cout << "Orginal:     " << test << endl;
   test = p;
   cout << "Modifiziert: " << test << endl;
}
void main()
{
   char *cp = "Hallo, ich bin's";

   f(cp);
   cout << "cp nach f(): " << cp << endl;
}
```

```
Orginal:     Hallo, ich bin's       | Hallo, ich bin's
Modifiziert: Ach so, na dann...!    | Ach so, na dann...!
cp nach f(): Hallo, ich bin's       | Ach so, na dann...!
```

Funktionen

- **Funktionsdeklaration**
  - bestimmt den Namen der Funktion, den Typ des zurückgelieferten Wertes (falls es einen gibt) und die Anzahl und Typen der Argumente
  - Deklarationen findet man häufig zusammengefaßt in den Header (.h) – Dateien.
- **Argumentübergabe: Argumenttypen werden überprüft und eine implizite Typkonvertierung wird bei Bedarf durchgeführt**

```
double   sqrt(double);
double   sr2 = sqrt(2);          // ruft sqrt() mit dem Arg. double(2)!
double   sq2 = sqrt("drei");     // Fehler: keine implizite Konvertierung möglich
```

- **Funktionsdefinition: Eigentliches Hinschreiben des Codes**

```
extern   void swap(int *, int *);    // Deklaration

void swap(int *p, int *q)            // Definition
{
  int tmp = *p; *p = *q;  *q = tmp;
}
```

- **Der Typ der Definition und alle Deklarationen müssen denselben Typ angeben!**

Funktionen
Statische Variable

- **Eine lokale Variable wird initialisiert, wenn der Kontrollfluß ihre Definition erreicht**
- **Bei einer "normalen", lokalen Variablen geschieht dies bei jedem Aufruf, bei statischen, lokalen Variablen nur beim ersten Aufruf:**

```
void f(int a)
{
   while (a--)
   {
      static  int  n = 0;      // Einmal initialisiert
              int  z = 0;      // n-mal initialisiert

      cout << "n == " << n++ << ", x == " << x++ << endl;
   }
}
```

- **Ausgabe f(3):**

```
n = 0, x = 0
n = 1, x = 0
n = 2, x = 0
```

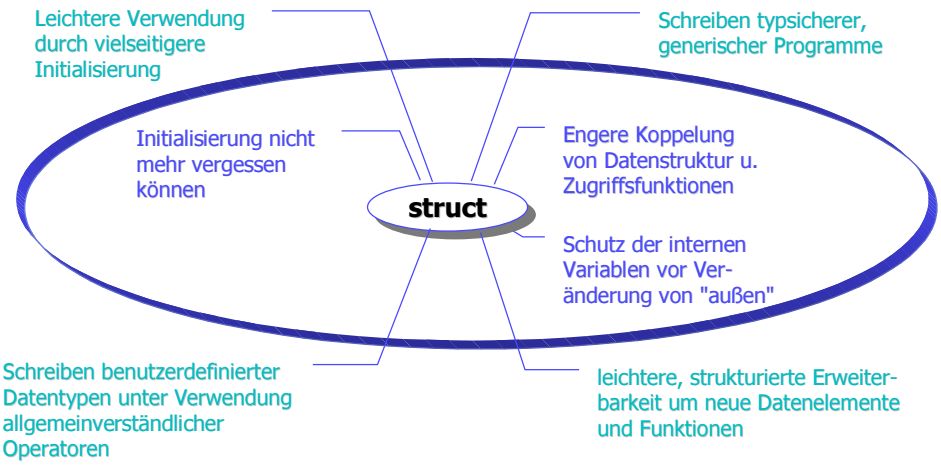
- **Eine statische Variable erzeugt für eine Funktion ein "Gedächtnis"**

Funktionen
Argumentübergabe

- **Übergabe einer Kopie der Variablen/Struktur (call-by-value)**
  - Die Argumente werden nicht nach außerhalb der Funktion verändert
- **Übergabe der Adresse einer Variable/Struktur**
  - Zugriff auf die Daten über Zeiger, Argumentvariable können verändert werden
- **Übergabe einer Referenz auf die Variable/Struktur mit der Möglichkeit der Manipulation (by-reference)**
  - Kein *-Zugriff nötig
- **Klassifizierung von Parametern als const**
  - Bsp. strcpy(char *ziel, **const** char *quelle)
  - Quelle kann zwar referenziert, jedoch nicht verändert werden!
- **Klassifizierung von Zeigerparametern als const**
  - Bsp.: double calc(double **const** * datum)
  - Zeiger kann in der Funktion nicht verändert werden!
- **Klassifizierung von Referenzparametern als const**
  - Bsp: void f(**const** Gross &arg)
  - Zeigt an, daß das Referenzobjekt in der Funktion nicht verändert wird/werden kann

Benutzerdefinierte Typen
- Klassen -

Erweiterung des Konzepts "Struktur"



Konstruktoren

- **Konstruktionsbeispiele für eine Datumsklasse mit Tag, Monat, Jahr**
  - `Datum heute = Datum(1, 4, 1998);`
  - `Datum heiligabend(24, 12, 1997); // Abgekürzte Schreibweise`
  - `Datum meinGeburtstag;            // Fehler: Parameter fehlen`
  - `Datum version10(10, 12);         // Fehler: 3. Parameter fehlt`
- **Häufig sinnvoll, ein Objekt auf verschiedene Arten initialisieren zu können (--> Angabe von mehreren Konstruktoren):**

```
class Datum
{
     int   t, m, j;
public:
     Datum(int, int, int);    // Tag, Monat, Jahr
     Datum(int, int);         // Tag, Monat, aktuelles Jahr
     Datum(int);              // Tag, aktueller Monat, aktuelles Jahr
     Datum();                 // Standardinit.: heutiges Datum
     Datum(const char*);      // Datum aus String-Repräsentation
};
```

- **Alternativ:**

```
     Datum(int tt=0, int mm=0, int jj=0);  // Defaultwerte
```

  ^ Problem: Wo wird der Defaultwert zwischengespeichert?

Statische Klassenelemente

- **Problemfall**
  - Defaultbelegung für alle Instanzen einer Klasse --> globale Variable?
- **Statische Elemente**
  - Variable, die Teil einer Klasse, jedoch nicht Teil eines Objekts dieser Klasse ist
- **Statische Elementfunktionen**
  - eine Funktion, die Zugriff auf Elemente einer Klasse benötigt, jedoch nicht für ein bestimmtes Objekt aufgerufen werden muß

```
class Datum {
    int t, m, j;
    static Datum stdDatum;
public:
    Datum(int tt=0, int mm=0, int jj=0);

    /* ... */
    static void setzeStd(int,int,int);
};
```

Def. des Datum-Konstruktors:

```
Datum::Datum(int tt, int mm, int jj)
{
    t = tt ? tt : stdDatum.t;
    m = mm ? mm : stdDatum.m;
    j = jj ? jj : stdDatum.j;
}
```


Verwendung statischer Klassenelemente / Elementfunktionen

```
class Datum {
   int t, m, j;
   static Datum standardDatum;
public:
   Datum(int tt=0, int mm=0, int jj=0);
   /* ... */
   static void setzeStandard(int,int,int);
};
```

- **Setzen des Standarddatums:**

```
void f()
{
    Datum::setzeStandard(4,5,1945);
}
```

- **Statische Elemente (Funktionen und Daten) müssen definiert werden!**
  - `Datum Datum::standardDatum(16,12,1770);`
  - ```
    void Datum::setzeStandard(int t, int m, int j)
    {
       Datum::standardDatum = Datum(t,m,j);
    }
    ```
    ```

Selbst-Referenz, this (1)

- Die (Zugriffs-) Funktionen zum Ändern eines Datums (addiereJahr, addiereMonat, addiereTag), können etwa wie folgt definiert werden:

```
inline void Datum::addiereJahr(int n)
{
   j += n;
}
```

- addiereJahr (wie auch alle anderen) liefert keinen Wert zurück
- Häufig ist es sinnvoll, eine Referenz auf das veränderte Objekt liefern zu lassen, um solche Operationen aneinanderhängen zu können

```
void f(Datum &d)
{
   // ...
   d.addiereTag(1).addiereMonat(1).addiereJahr(1);
   // ...
}
```

- Hierzu muß jede der Funktionen eine Referenz auf ein Datum liefern

Selbst-Referenz, this (2)

```
class Datum
{
   // ...
   Datum &addiereJahr(int n); // addiere n Jahre
   // usw.
};
```

- Jede (nicht-statische) Elementfunktion weiß, für welches Objekt sie aufgerufen wurde und kann sich explizit auf dieses beziehen:

```
Datum &Datum::addiereJahr(int n)
{
   if (t == 29 && m == 2 && !schaltjahr(j+n))   // 29.Feb.!!
   {
      t = 1, m = 3;
   }
   j += n;
   return *this;
}
```

- nicht-konstante Elementfunktion: "this" hat den Typ X *const
- konstante Elementfunktion: "this" hat den Typ const X *const

Kopieren von Klassen-Objekten (1)

- **Standardmäßig können Objekte kopiert werden**
  - Zuweisung eines Objekts an ein anderes der gleichen Klasse (operator=)
  - Insbesondere kann ein Objekt mit einer Kopie eines Objekts derselben Klasse initialisiert werden (Copy-Konstruktor)

```
Datum t = heute; // Initialisierung durch eine Kopie
```

- **automatische Standardform des Kopierens**
  - Kopieren aller Klassenelemente
  - *Immer sinnvoll?*
- **Alternative:**
  - Definition eines (eigenen) Copy-Konstruktors X::X(const X&)

Kopieren von Klassen-Objekten (2)
Probleme mit dem Default-Copy-Konstruktor

- **Klasse enthält Elemente mit Zeigern auf weitere Objekte**
- **Nach Kopieren durch Default-Copy-Konstruktor oder operator= gibt es in zwei Objekten Zeiger auf das gleiche (Speicher-) objekt**

```
class Stack {
   int          *buf;          // Membervariablen
   int          top;
   /* ... */
};
```

- **Verwendungsmöglichkeiten / Probleme**

```
void h()
{
   Stack   s1;
   Stack   s2 = s1;
   Stack   s3;

   s3 = s2;
}
```

  - Zwei Aufrufe des Default-Konstruktors (s1, s3)
  - Drei Aufrufe des Destruktors!
  - Nicht mehr referenzierter Speicher in s3!

Kopieren von Klassen-Objekten (3)
Eigener Copy-Konstruktor

```
class Stack {
    // ...
    Stack(const Stack&);               // Copy-Konstruktor
    Stack& operator=(const Stack&);    // Zuweisungsoperator
}
```

```
Stack::Stack(const Stack& s)  // Copy-Konstruktor
{
   buf = new int[size = s.size];
   for (int i=0; i <= top; i++)
     buf[i] = s.buf[i];
}
```

```
Stack &Stack::operator=(const Stack &s) // Zuweisungsoperator
{
   if (this != &s) {     // Vorsicht bei Selbstzuweisung s = s
     delete [] buf;
     buf = new int [size = s.size];
     for (int i=0; i <= top; i++)
          buf[i] = s.buf[i];
   }
   return *this;
}
```