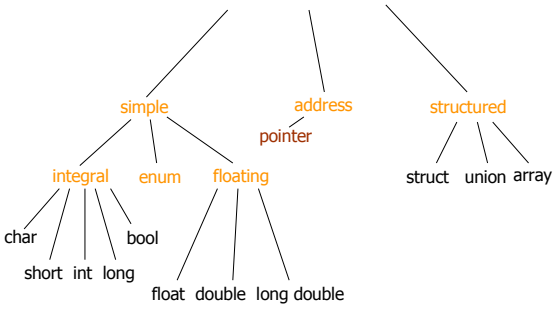
Grundlagen der Programmierung in C
Pointer & Co.

Wintersemester 2005/2006
G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Das C++ Typsystem



- Problem:
  - Variablename ist *fest* mit Speicherbereich verbunden
  - Ziel: Programmstück, das beliebige Speicherbereiche verarbeiten kann, ohne vorher extra Kopie zu machen (vorausgesetzt, der Typ stimmt)
- Beispiel:
  - Annahme: **`Polynom`** ist Struct/Array mit 100 Koeffizienten

```
Polynom p1, p2, p3;
...
if ( bedingung )
    p1 = p2 ;           // kopiert 100 Koeff.!
else
    p1 = p3 ;           // dito
bearbeite p1
wieder zurück kopieren  // kopiert 100 Koeff.!
```

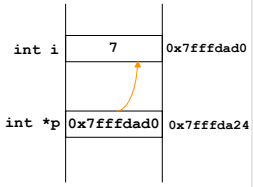

- Lösung: "Zeiger"

```
Polynom p2, p3;
Polynom-Zeiger p1;
...
if ( bedingung )
    p1 zeigt ab jetzt auf p2
else
    p1 zeigt ab jetzt auf p3
bearbeite das, worauf p1 zeigt
```

- "Pointer sind das Goto der Datenstrukturen"
  - Existieren auch in Java & Python, sieht man bloß nicht

Was ist ein Pointer?

- Erinnerung:
  - Variable = Name für Speicherbereich = Name für Adresse + Typ
  - Typ (z.B. int) definiert, wie Bits interpretiert werden sollen, die an dieser Adresse gespeichert sind
  - Jede Variable ist genau einem Adreßbereich fest zugeordnet
- Pointer:
  - Variable, wie alle anderen auch
    - Hat Wert
    - Steht irgendwo im Speicher an bestimmter Adresse
    - Hat Typ
  - Typ = Bedeutung des Wertes = Adresse einer anderen Variable!



Eigenschaften von Pointern

- Auf Wert einer anderen Variablen / Speicherbereich zugreifen, ohne deren Namen zu verwenden (oder kennen)!
- Ansonsten fast alle Fähigkeiten der "normalen" Variablen
  - Arithmetik
  - Zuweisen
  - Vergleichen

Deklaration

- Deklaration:

  Typ varname;*

  wobei *Typ* ein bekannter Typ ist.
- Beispiele:

```
int* pi;
float** pf;
```

```
struct S { … };
S* ps;
```

- In gew. Sinn orthogonal zum Konzept "Typ":
  - Zu jedem Typ **T** gibt es einen "Pointer-Typ" **T***
- Andererseits ist *Pointer-Taking* integraler Bestandteil des C++-Typsystems

Verwendung: Adressoperator

- Pointer auf Adreßbereich einer Variablen zeigen lassen:

  pointervar = & *var;*

  wobei *var* vom Typ T ist und *pointervar* vom Typ T*.
- Neuer Operator **&** heißt "Adressoperator"
- Beispiele:

```
int* pi;
int i = 17;
pi = & pi;
```

```
struct S { … };
S s;
S* ps = & s;
```

```
float f;
float *pf = &f;
float** ppf = &pf;
```

```
float f;
// folgendes geht nicht
float** ppf = & & pf;
```


Verwendung: Dereferenzierung ("dereferencing")

- Umkehrung des &-Operators
- Syntax:

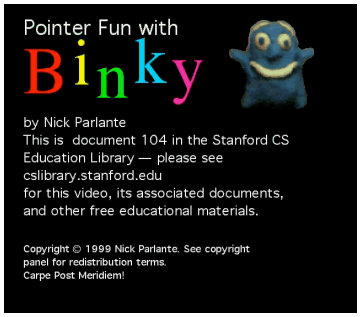
  * *ptr-expr*

  wobei *ptr-expr* ein Ausdruck ist, der einen Typ **T*** liefert;

  Resultat hat dann den Typ **T**.
- Neuer Operator * (Stern-Operator)
- Beispiele:

```
int i=1, j=0;
p = &i;      // p zeigt auf i
i = i + *p;  // verdoppelt i
p = &j;      // p zeigt jetzt auf j
*p = 42;     // j ist jetzt 42 (i gleibt unverändert).
```


"Der Film"



Bem.: "pointee" = "das warauf der Zeiger zeigt"

Konsequenzen des neuen Sprachkonstruktes

- Neues Sprachkonstrukt ("Feature")
- Welche Wechselwirkungen hat dieses Feature mit allen anderen? Passen alle anderen damit zusammen? Gibt es Sonderfälle?
- "Komplextät" einer Sprache wird bestimmt durch die Anzahl solcher Wechselwirkungen und – insbesondere – der Sonderfälle!
  - n Features -> n^2 viele mögliche Wechselwirkungen!
  - Sprachdesign: möglichst keine Sonderfälle (Orthogonalität)

Pointer auf Structs

- Kommt sehr häufig vor (insbesondere später bei Klassen)
- Wie alle anderen Pointer auch
- Zugriff auf Members eines Structs:

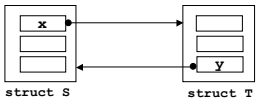
```
struct S { float x, y };
S * p;
... (*p).x ...
```

- Abkürzende Schreibweise

```
... p->x ...
```

Forward-Deklaration von Structs

- Problem: wie deklariert man folgende 2 Structs?



- Beobachtung: der Pointer **`T*`** ist immer gleich groß, unabhängig von der Größe von **`T`**
- Lösung: Forward-Deklaration

```
struct T;

struct S
{
    T* x;
    ...
};

struct T { ... };
```


Vergleich von Pointern

- Pointer kann man auf == und != vergleichen
  - Wie bei allen anderen Typen auch
  - Gleichheit bedeutet: zeigen auf dieselbe Variable
- Alle anderen Vergleiche sind auch erlaubt
  - Selten gebraucht

Weak Typing

- Eine (von zwei) Definitionen für *Strong Typing* :=
  ein Speicherblock ist zu genau einem Objekt (z.B. ein Double) zugeordnet, dieser Block hat genau einen Typ, und es gibt *keine* Möglichkeit im Programm, diesen Speicherblock als anderen Typ zu *interpretieren*.
- Natürlich darf man das Objekt kopieren, die Kopie in einen anderen Typ verwandeln, und dann diese Kopie in einen anderen Speicherblock schreiben.
- Definition *weakly typed* := nicht strongly typed.

Type-Safety

- Definition für *Typ-sicher* :
  Ein Sprachkonstrukt ist *typ-sicher* , wenn dadurch keine Uminterpretierung (im Sinne der starken Typisierung) möglich wird.
- Alternative Definition für *weakly typed* :
  Je mehr typ-unsichere Sprachkonstrukte eine Sprache hat, desto schwächer typisiert ist sie.
- Deswegen ist Pointer-Zuweisung verboten (i.A.)

```
Typ1* p1;
Typ2* p2;
p2 = p1;      // error!
```

wenn ***`Typ1`*** und ***`Typ2`*** verschieden sind!

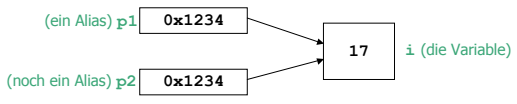
"Null"-Pointer

- Problem: wie unterscheidet man gültigen Pointer von Pointer, der auf nichts zeigen soll?
- Adresse 0 bzw. Wert **`NULL`** ist genau dafür reserviert
- Was passiert, wenn man Null-Pointer dereferenziert?
  - Core Dump (rel. einfacher Bug)
  - Passiert oft auch bei uninitialisierten Pointern oder "wilden" Pointern (schon schwerer zu finden)
- Beispiel:

```
char* findIt( char* s, char c )
{
    while ( *s != '\0' )
    {
        if ( *s == c )
            return s;
      s += 1;
    }
    return NULL;          // oder 0
}
```


Aliasing

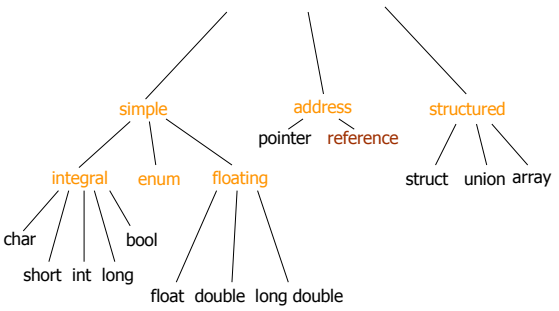
- Dieselbe Variable kann jetzt über viele verschiedene Wege (Pointer oder Referenz) erreicht werden
- Nennt man "Aliasing"



- Problem für Compiler bei Optimierung
- Beispiel:

```
int i;
int* ip1 = &i;
int* ip2 = &i;
*ip1 = 17;
foo( *ip2 );
```

Das C++ Typsystem



Referenzen

- Problem der Pointer:
  - Wert (Adresse, auf die er zeigt) kann sich beliebig ändern
- Lösung: neues Sprachkonstrukt "Referenz"
- Eigenschaften:
  - Hat *immer* einen Typ (kein `void*` möglich)
  - Zeigt *immer* auf dieselbe Variable
  - Verhält sich also wie ein konstanter Pointer
- Manchmal sehr praktisch
  - Versteckt Indirektion vor dem Programmierer
  - Compiler kann mehr optimieren
  - Weniger zu tippen

Syntax

- Deklaration & Initialisierung:

  `Typ & refname = varname;`

  - Beachte: Keine Deklaration ohne Initialisierung!
- Verwendung:

  `refname`

  - Beachte: kein Derefenzierungsoperator!
- Referenz = anderer Name (Alias) für dasselbe Objekt
- Beispiele:

```
int i = 17, j = 42;
int & ri = i;
i += ri;            // verdoppelt i
ri = j;             // jetzt i == 42
```


Referenzen auf Structs

- Hier sind Referenzen oft recht bequem:

```
struct S { float x, y };
S s;
...  s.x  ...  // Member-Zugriff
S * ps;
...  p->x  ... // Member-Zugriff über Pointer
S & rs = s;
...  pr.x  ... // Member-Zugriff über Referenz
```


Vergleich Pointer vs. Referenz

Pointer

- Zeigt explizit auf ein anderes Objekt
- Kann man auf NULL testen
- Kann auf beliebig viele verschiedene Objekte zeigen = variabler Zeiger
- Void-Pointer
- Bei Auswertung explizit als Pointer zu erkennen

Referenz

- Ist ein zweiter Name (Alias) für ein Objekt
- Sollte immer auf etwas zeigen
- Kann nur für ein Objekt Alias sein (kann nicht nachträglich geändert werden) = konstanter Zeiger
- Immer getypt
- Bei Auswertung nicht als Referenz zu erkennen

Kombination von Pointern und Referenzen

- Was tut folgender Code?

```
int j = 1;
int & r = j;
int * p = r;
*p = 2;
int * & t = p;
t = NULL;
```

- Solch ein Gemisch sollte man möglichst vermeiden!

Mehrfachbedeutung von * und &

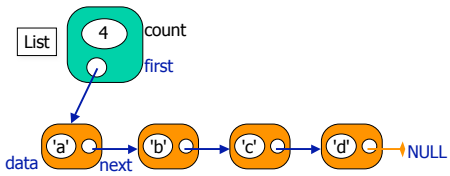
- Bedeutung ist abhängig vom Kontext!

Symbol	In einer Deklaration	In einem Ausdruck
unäres `&` (ampersand)	Referenz `int i; int &r = i;`	Adress-Operator `p = & i;`
unäres `*` (star)	Pointer `int * p;`	Dereferenzierung `*p = 7;  i = *p + 3;`

- Daneben gibt es noch die binären Operatoren `*` und `&` !
- Alternative wäre:
  - Alle C++-Programmierer kaufen sich eine neue Tastatur ☺

Beispiel: Verkettete Listen (Linked Lists)

- Sehr häufige dynamische Datenstruktur
- Besteht aus Folge von (gleichartigen) Elementen
  - Jedes kennt Vorgänger und Nachfolger
  - Man kennt den Anfang (Kopf, head) der Liste

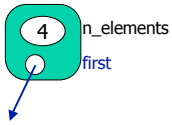


- Ein Element:

```
struct ListElement
{
    float x, y;
    int z;
    ListElement* next;
};
```

- Der "Anker":

```
struct List
{
    ListElement* first;
    ListElement* last;
    int n_elements;
    ...
};
```

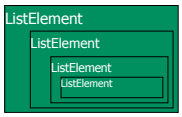


- Achtung:

```
struct ListElement
{
    float x, y;
    int z;
    ListElement next;
};
```

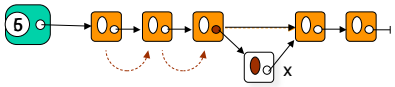
klappt nicht (* fehlt)!

- Das wäre eine "rekursive" Datenstruktur ☺



Einfügen

- Durch "Umbiegen"der Zeiger




```
// insert x (= ListElement*) after n-th element
ListElement* e = list.first;
int i = 1;
while ( i < n && e->next )
{
    i ++ ;
    e = e->next;
}
// Nachbedinung: e zeigt auf Elem.,
// hinter dem x eingefuegt werden soll
X->next = e->next;
e->next = x;
```



```
// insert x (= ListElement*) after n-th element
if ( n == 0 )
{
    // Einfügen als neuer Kopf der Liste
    x->next = list.first;
    list.first = x;
}
else
{
    ListElement* e = list.first;
    int i = 1;
    while ( i < n && e->next )
    {
        i ++ ;
        e = e->next;
    }
    // Nachbedinung: e zeigt auf Elem.,
    // hinter dem x eingefuegt werden soll
    x->next = e->next;
    e->next = x;
}
```

Code mit "Randfall"

Entfernen (ohne Löschen)

- Durch analoges Umbiegen der Zeiger