

7. Zeiger

Prof. Dr. Markus Gross
 Informatik I für D-ITET (WS 03/04)

- Zeiger (Pointers)
- Dynamische Speicherverwaltung
- `new` und `delete`



Eidgenössische Technische Hochschule Zürich
 Swiss Federal Institute of Technology Zurich

Copyright: M. Gross, ETHZ, 2003

2



Eidgenössische Technische Hochschule Zürich
 Swiss Federal Institute of Technology Zurich

Zeiger (Pointers) und Adressen

- Bei der Datenspeicherung muss dreierlei Information gespeichert werden
 - ◆ Wo ist das Datum gespeichert
 - ◆ Welchen Wert hat es
 - ◆ Welchen Typ hat es
- Zur Ermittlung der Speicheradresse eines Datums dient der *Adressoperator &*
- Beispiel:
 - ◆ `int value = 5; //`
 - ◆ `&value; //` liefert die Adresse (hexadezimal)
- Adressen können je nach Implementierung verschieden gross sein
 - ◆ Typischerweise 32 Bit (`0x1234ac3d`)

DON'T



PANIC

Pointers

- Eine *Pointervariable* speichert die Adresse eines Datentyps
- Sie kann zur *Referenzierung* einer Variablen herangezogen werden
- Sehr mächtiges Werkzeug in C/C++
- Syntax zur Definition verwendet den *Dereferenzierungsoperator **
 - ◆ `int value = 5; // int Variable`
 - ◆ `int *p_value; // Pointer auf int`
 - ◆ `p_value = &value; // Initialisierung auf
// Adresse von value`
 - ◆ `int new_value = *p_value; // Zugriff auf Inhalt
// der Speicheradresse durch Dereferenzierung`

Beispiel 4: Adressen

```
// pointer.cpp _ our first pointer variable
#include <iostream>
using namespace std;

int main()
{
    int updates = 6; // declare a variable
    int *p_updates; // declare pointer to an int
    p_updates = &updates; // assign address of int to pointer
    // express values two ways
    cout << "Values: updates = " << updates;
    cout << ", *p_updates = " << *p_updates << "\n";

    // express address two ways
    cout << "Addresses: &updates = " << &updates;
    cout << ", p_updates = " << p_updates << "\n";

    // use pointer to change value
    *p_updates = *p_updates + 1;
    cout << "Now updates = " << updates << "\n";
    return 0;
}
```

Pointer Variable

Initialisierung
des Pointers

Zugriff auf Members

Definition und Initialisierung

- Typ der Variablen, auf die der Pointer zeigt, muss bei der Initialisierung mit angegeben werden
 - ◆ `double *p_value;`
 - ◆ `double* p_value; // ebenfalls möglich`
 - ◆ `double *p1, p2; // Vorsicht !!!!`
- Pointer ist ein von `int` verschiedener Typ
- Daher kann man einem Pointer keinen Integer zuweisen
- Es wird eine *explizite Typenumwandlung (explicit typecast)* benötigt
 - ◆ `int *p1;`
 - ◆ `p1 = 0x4356ff0c; // type mismatch`
 - ◆ `p1 = (int *) 0x4356ff0c; // type cast`

Situation im Speicher

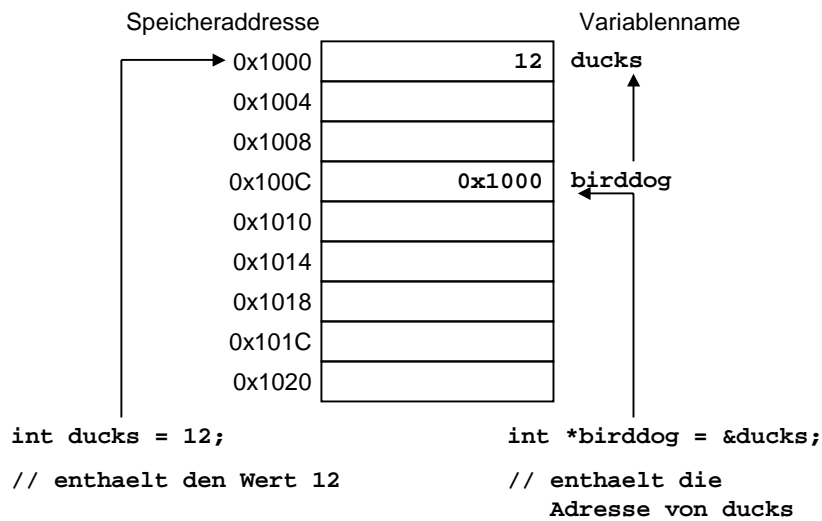
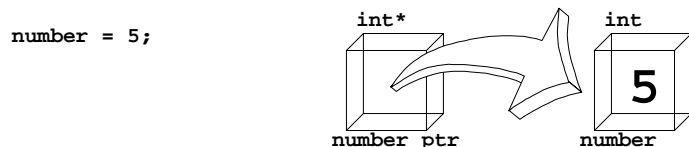
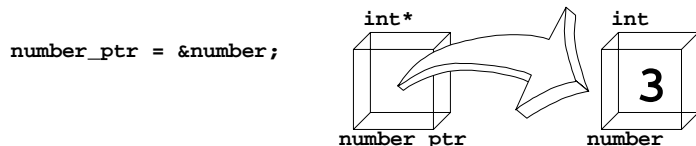
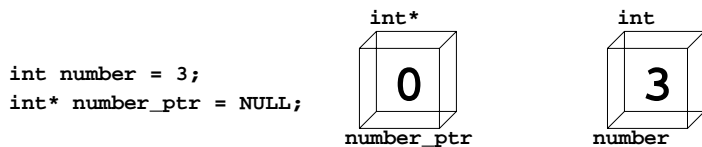
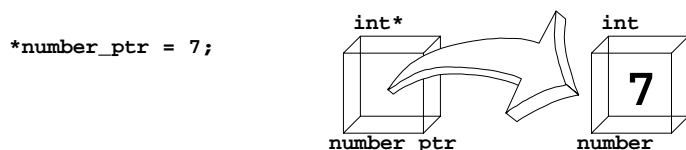


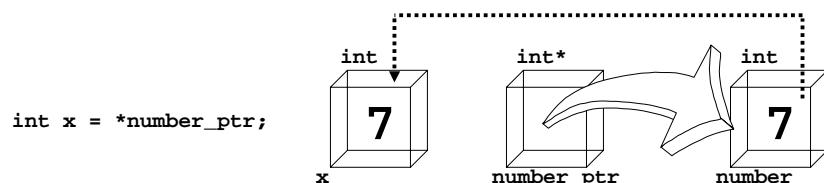
Illustration (I)



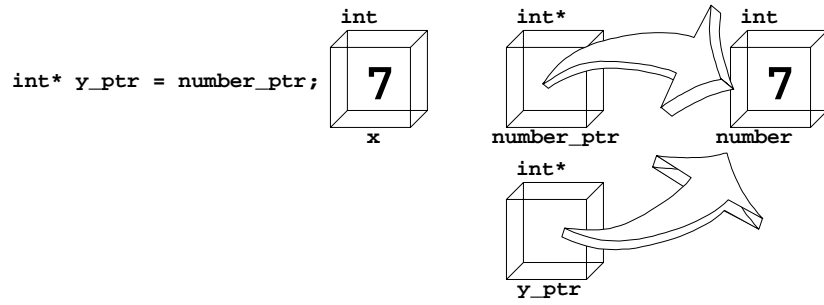
Pointers (II)



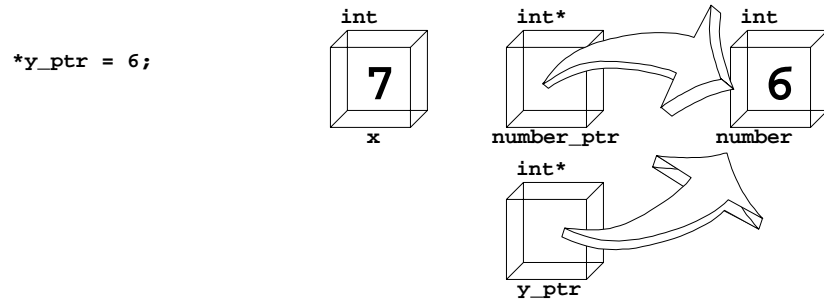
```
*number = 8; // FEHLER!! number ist kein Pointer
```



Pointers (III)

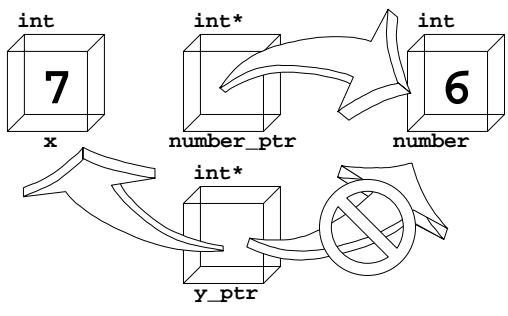


Pointers (IV)



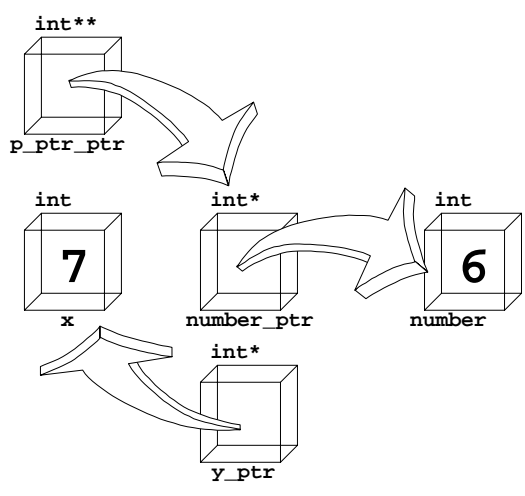
Pointers (V)

```
y_ptr = &x;
```

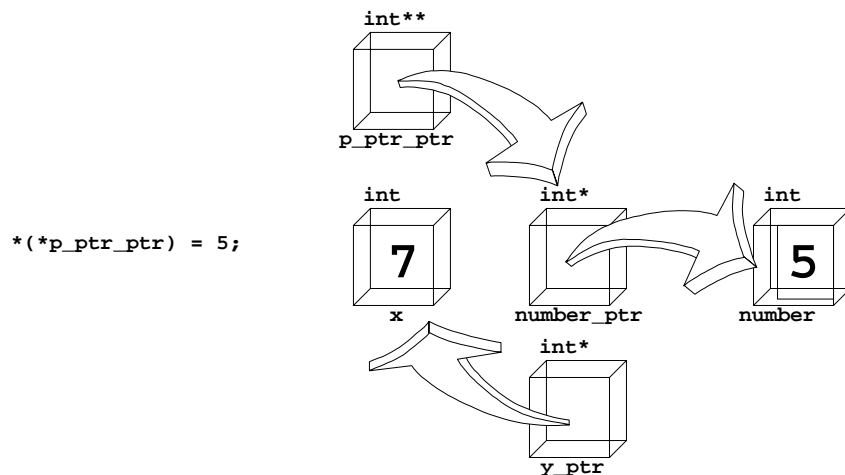


Pointers (VI)

```
int* *p_ptr_ptr;  
p_ptr_ptr = &number_ptr;
```



Pointers (VII)



Dynamische Speicherverwaltung

- Bisher wurde Grösse des Speicherplatzes für Variablen zur *Compilezeit* festgelegt
- Pointers erlauben eine *dynamische* Speicherverwaltung zur *Laufzeit*
- Die Definition eines Pointers reserviert nur Platz zur Speicherung einer Adresse
 - ◆ `double *p_value; // Speicher für Adresse, nicht // für den double`
- `new`-Operator kreiert erforderlichen Speicher
 - ◆ `double *p_value = new double; // Speicher für // Adresse UND für einen Wert vom Typ double`
- Dieser Speicher ist namenlos, kann jedoch über Pointer verwendet werden

Beispiel_5: New

```
// use_new.cpp _ using the new operator
#include <iostream>
using namespace std;

int main()
{
    int * pi = new int; // allocate space for an int
    *pi = 1001; // store a value there

    cout << "int ";
    cout << "value = " << *pi << ": location = " << pi << "\n";

    double * pd = new double; // allocate space for a double
    *pd = 10000001.0; // store a double there

    cout << "double ";
    cout << "value = " << *pd << ": location = " << pd << "\n";
    cout << "size of pi = " << sizeof pi;
    cout << ": size of *pi = " << sizeof *pi << "\n";
    cout << "size of pd = " << sizeof pd;
    cout << ": size of *pd = " << sizeof *pd << "\n";

    delete pi;
    delete pd;

    return 0;
}
```

Pointer Variable sowie Allokation von Speicher

Speicherung eines Wertes


sizeof-Operator

Dynamische Speicherverwaltung

- **new**-Operator liefert Anfangsadresse des allozierten Speichers
- Bei Misserfolg wird 0 zurückgeliefert (*null pointer*)
- Speicher muss explizit vom Benutzer freigegeben werden
 - ◆ Je nach Betriebssystem kann ein *Speicherloch (memory leak)* drohen
- **delete**-Operator gibt Speicher frei
 - ◆ `double *p_value = new double;`
 - ◆ `delete p_value; // Speicherfreigabe`
- Es muss dabei die gleiche Adresse verwendet werden (nicht die gleiche Pointer-Variable)



Dynamische Arrays

- Grosse Arrays wurden bisher *statisch* zur Compilezeit festgelegt
- Mit Pointern kann die Arraygrösse *dynamisch* zur Laufzeit gesetzt werden
 - ◆ `int *a_ptr = new int [10]; // allocate`
 - ◆ `delete [] a_ptr; // Speicher freigeben`
- Allgemeine Form zur Allokation dynamischer Arrays:
 - ◆ `type_name pointer_name = new type_name [nb];`
- Verwendung des Arrays über Pointername
- *Pointername wird als Arrayname verwendet* 
- Allgemein in C/C++: Name eines Arrays = Pointer auf Anfangsadresse
- Arrays werden intern über Pointers repräsentiert

Beispiel_6: Dynamische Arrays

```
// arraynew.cpp _ using the new
operator for arrays
#include <iostream>
using namespace std;
int main()
{
    double * p3 = new double [3];
    // space for 3 doubles

    p3[0] = 0.2;
    // treat p3 like an array name

    p3[1] = 0.5;
    p3[2] = 0.8;

    cout << "p3[1] is " << p3[1] <<
    ".\n";

    p3 = p3 + 1;
    // increment the pointer

    cout << "Now p3[0] is " << p3[0] <<
    " and ";
    cout << "p3[1] is " << p3[1] <<
    ".\n";
    p3 = p3 - 1;
    // point back to beginning
    delete [] p3;
    // free the memory
    return 0;
}
```

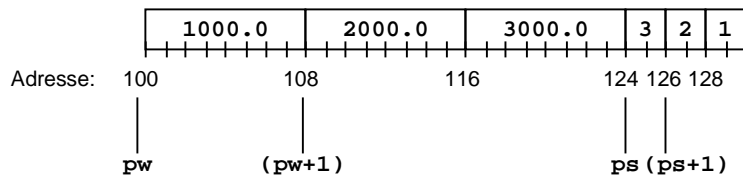
- Zugriff durch Pointername und Arrayklammern []
- `p3[0]` ist erstes Element
- Pointer können inkrementiert und dekrementiert werden
- `p3 = p3 + 1` setzt pointer auf nächstes Element im Array
- Adressierung immer relativ zur Adresse von `p3`
- Vorsicht geboten!!

Pointer-Arithmetik

- Pointer erlauben eine elegante Adressierung von Array-Elementen
- Verwendung von arithmetischen Operatoren (+,-) und Dereferenzierung (*)
- Inkremente und Dekremente sind jeweils an Grösse des Typs aligniert
- Wird vom Compiler berechnet und kann beliebig gross sein
- Arrayname ist auch bei statischen Arrays Adresse des ersten Elementes
 - ◆ `int i_array[10]; // statisches Array`
 - ◆ `int *i_ptr = i_array; // Pointer`
- `i_array` und `&i_array[0]` sind gleich

Illustration

```
double wages[3] = {1000.0, 2000.0, 3000.0};
short stacks[3] = {3, 2, 1};
double *pw = wages;
short *ps = &stacks[0];
```



wenn man 1 zu `pw` addiert wird der Wert um 8 Bytes erhöht weil `pw` ein Pointer ist der auf ein `double` zeigt

wenn man 1 zu `ps` addiert wird der Wert um 2 Bytes erhöht weil `ps` ein Pointer ist der auf ein `short` zeigt

Beispiel_7: Arrays und Pointers

```
// addptrs.cpp -- pointer addition
#include <iostream>
using namespace std;
int main()
{
    double wages[3] = {10000.0, 20000.0, 30000.0};
    short stacks[3] = {3, 2, 1};

    // Here are two ways to get the address of an array

    double *pw = wages; // name of an array = address
    short *ps = &stacks[0]; // or use address operator
                          // with array element
    cout << "pw = " << pw << ", *pw = " << *pw << "\n";
    pw = pw + 1;
    cout << "add 1 to the pw pointer:\n";
    cout << "pw = " << pw << ", *pw = " << *pw << "\n\n";

    cout << "ps = " << ps << ", *ps = " << *ps << "\n";
    ps = ps + 1;
    cout << "add 1 to the ps pointer:\n";
    cout << "ps = " << ps << ", *ps = " << *ps << "\n\n";

    cout << "access two elements with array notation\n";
    cout << stacks[0] << " " << stacks[1] << "\n";
    cout << "access two elements with pointer notation\n";
    cout << *stacks << " " << *(stacks + 1) << "\n";

    cout << sizeof wages << " = size of wages array\n";
    cout << sizeof pw << " = size of pw pointer\n";
    return 0;
}
```

Statische Arrays

Pointer-Arithmetik

Zugriff über Name

Pointer-Arithmetik

- Aufgrund der Prioritäten der Operatoren müssen Klammern verwendet werden


```
int i_array[10]; // statisches Array
*i_array + 5 // i_array[0] + 5
*(i_array + 5) // i_array[5]
```
- Allgemein ist


```
array_name[i] entspricht *(array_name + i)
```

Dynamische Strukturen

- Pointers eignen sich zur Verwaltung dynamischer Strukturen

```

struct men
{
    int good;
    int bad;
};
.....
men freshmen = {241,0};
men *m_ptr = &freshmen;
.....
int a = freshmen.good; // Zuweisung
int b = m_ptr->good;   // Zuweisung +
                       // Dereferenzierung

```

Dynamische Strukturen

- Bei Pointern auf Strukturen erfolgt Zugriff mittels -> Operator
- Greift auf Mitgliedsvariable zu UND dereferenziert
- Entspricht der Verwendung von * und .

```
m_ptr->good entspricht (*m_ptr).good
```

- Name eines Arrays kann nicht verändert werden
- Arrayname ist also **const**-Pointer
- Benutzereingabe der Arraygrenzen werden möglich

```

int size;
cin >> size;
int *p_int = new int [size];

```

Speicherklassen

- Gewöhnliche Variablen innerhalb von Funktionen haben Speicherklasse **automatic**
- Ihr Gültigkeitsbereich ist auf den aktuellen Block beschränkt (*block scope*)
- Werden bei Eintritt in den Block angelegt
- Erlöschen bei Austritt aus dem Block
- Werden auf dem *Stack* verwaltet
- Soll eine Variable während des gesamten Programmes Gültigkeit besitzen, so gibt es zwei Möglichkeiten
 - ◆ Definition ausserhalb von Funktionen (*globale Variable*)
 - ◆ Verwendung der Speicherklasse **static** (*statische Variable*)

Speicherklassen

- Statische Variablen werden durch das zusätzliche Keyword **static** definiert
`static int s = 5;`
- Werden automatisch initialisiert und behalten ihren Wert bei Beenden der Funktion (Ausnahme **main**)
- Können sinnvoll zur Speicherung funktionsunabhängiger Werte verwendet werden

Beispiel_8: Dynamische Structs

```
// newstrct.cpp _ using new with a structure
#include <iostream>
using namespace std;

struct inflatable // structure template
{
    char name[20];
    float volume;
    double price;
};

int main()
{
    inflatable * ps = new inflatable; // allot structure space

    cout << "Enter name of inflatable item: ";
    cin.get(ps->name, 20); // method 1 for member access

    cout << "Enter volume in cubic feet: ";
    cin >> (*ps).volume; // method 2 for member access

    cout << "Enter price: $";
    cin >> ps->price;

    cout << "Name: " << (*ps).name << "\n"; // method 2
    cout << "Volume: " << ps->volume << " cubic feet\n";
    cout << "Price: $" << ps->price << "\n"; // method 1

    delete ps;
    return 0;
}
```

Deklaration der Struktur

Pointer auf struct

Zugriff auf Mitglied

Beispiel_9: Dynamisches Array

```
cin >> limits;
// allocate 1D array of pointers
double **d2array = new double *[limits];

// allocate rows of array
for (int i = 0; i<limits; i++)
    d2array[i] = new double[limits];

// access array
for (i = 0; i<limits; i++)
    for (int j = 0; j<limits; j++) // i,j do not expire,
        d2array[i][j] = i*j; // block scope

cout << "Arraywerte:\n";
for (i = 0; i<limits; i++)
{
    for (int j = 0; j<limits; j++)
        cout << d2array[i][j] << " ";
    cout << "\n";
}

cout << "\n";

// free memory using delete
for (i = 0; i<limits; i++)
    delete [] d2array[i];

// now delete array of pointers
delete [] d2array;
```

Pointer auf Pointer

Allokation in Schleife

Zugriff konventionell

Speicherfreigabe