Computer Vision
and Geometry Lab

Informatik I for D-MAVT

Exercise Session 6

© Roland Angst, 2010

Organisatorisches

- Teaching Assistant
  - Alexander Schwing (aschwing@student.ethz.ch)
  - http://www.alexander-schwing.de/Info1DMaVt/
- Übungsabgabe
  - Auf Papier, keine Emails
    - Später, wenn Programme umfangreicher werden, ev. auch Abgabe per Email möglich
  - Gruppenlösungen erlaubt (aber max. 2 Studenten pro Gruppe)
  - Abgabetermin eine Woche nach Vorbesprechung
- Testatbedingungen
  - 75% aller Übungen
    - Voraussichtlich 12 Serien (d.h. 9 gelöste Serien)

Themen dieser Stunde

- Arrays
- Structures
- Pointer
- Dynamischer Speicher

Arrays

- **Arrays**
- Structures
- Pointer
- Dynamischer Speicher

Arrays (Felder)

- Folge von Elementen **desselben** Typs, die in **aufeinanderfolgenden** Speicherzellen abgespeichert werden.
- Einfacher „Container“ zum Speichern mehrerer Variablen desselben Typs
- Statische Allokation:
  - `typename variablename[arraysize];`
  - `arraysize` muss eine Konstante sein (zur Kompilierzeit bekannter Wert)
- Dynamische Allokation: siehe später in Übungsstunde

Initialisierung eines Arrays

- Mehrere Möglichkeiten:
  - `int a[5] = {1, 2, 3, 4, 5};  // a = [1, 2, 3, 4, 5]`
  - `int a[5] = {1, 2, 3};  // a = [1, 2, 3, 0, 0]`
  - `int a[]  = {1, 2, 3, 4, 5};  // a = [1, 2, 3, 4, 5]`
  - `int a[3][2] = {{1,2},{3,4},{4,5}};`

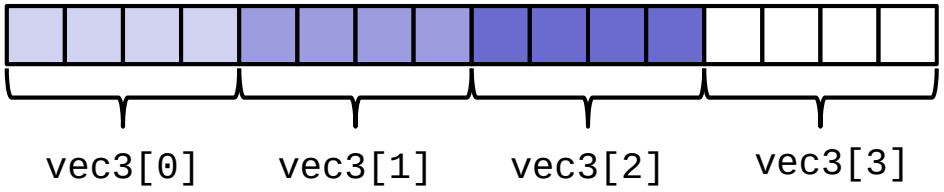
Zugriff auf Elemente eines Arrays

- Zugriff auf Arrayelemente
  - Indices beginnen bei `0` (und gehen bis `n-1`)
  - Unterschied zu Matlab (!!!)
- Beispiel:

```
float myarr[10];
myarr[0] = 0.0f;
myarr[1] = 1.0f;
for (unsigned int i=2; i < 10; i++)
  myarr[i] = myarr[i-2] + myarr[i-1];
```

Arrays im Speicher

- Benötigter Speicher:
  - `sizeof(type) × arraysize`
  - Grund: Elemente sind alle vom gleichen Typ & `arraysize` ist zur Compilezeit bekannt
- Physisches Layout im Speicher:
  - Sequentiell aufeinanderfolgend
  - Beispiel: `float vec3[3];// sizeof(float) == 4`



  - Was passiert bei: `vec3[3] = 0.5f;`

Überschreiten der Arraygrenze wird nicht kontrolliert!
Führt zu schlimmen Bugs!

Arrays für Strings

- C-Strings enden immer mit `\0` (Null-Charakter)
  - `char ok[3] = {`o`,`k`,`\0`};`
  - `char ok[3] = “ok“; // ein String Literal`
  - `// Dies ist ein array von characters, und kein String:`
    `char wrong[5] = {`w`,`r`,`o`,`n`,`g`};`
  - `char c = `c`; // ok`
    `char c = “c“; // type missmatch! String literal vs. char`
- String Literals
  - Werden automatisch mit Null-character abgeschlossen

Structures

- Arrays
- **Structures**
- Pointer
- Dynamischer Speicher

Structures

- Structures sind benutzerdefinierte Typen
  - Structures unterscheiden sich bzgl. Klassen lediglich darin, dass Members standardmässig `public` sind (siehe später)
- Neue Typdefinition:

```
struct MyNewDataType {
     member_type1 member1;
     member_type2 member2;
     ...
};
```

- Deklaration einer Variablen des neues Typs:

```
MyNewDataType myNewDataObject;
```

- Punktoperator für Zugriff auf Member-Variable:

```
myNewDataObject.member_type1 = ...;
```

Structures

- Membervariablen werden sequentiell gemäss Definition im Speicher abgelegt
- Benötigter Speicherplatz
  - Nicht zwingend gleich Summe des benötigten Speichers der einzelnen Membervariablen
    - Memory Alignment führt zu sogenanntem Padding
    - Also: Grösse des structs ist grösser gleich der Summe des erforderlichen Speicherplatzes der Membervariablen.
  - `sizeof` funktioniert auch auf selbst definierten structs!

Structures

- Beispiel (von Wikipedia):

```
struct MixedData {
 char Data1;
 short Data2;
 int Data3;
 char Data4;
};
```

8 Bytes without padding

```
struct MixedData /* after compilation */ {
 char Data1;
 /* For the following 'short' to be
    aligned on a 2 byte boundary */
 char Padding1[1];
 short Data2;
 int Data3;
 char Data4; char Padding2[3];
};
```

12 Bytes with padding

```
struct MixedData /* after reordering */ {
 char Data1;
 char Data4; /* reordered */
 short Data2;
 int Data3;
 };
```

8 Bytes with padding

Structures

- Zuweisung

```
struct MyStruct {
     int i;
     float f;
};
MyStruct m1, m2 = {1, 2.0f};
m1 = m2; // m1.i = m2.i; m1.f = m2.f;
```

- Ein Struct kann ein L-Wert sein (im Gegensatz zu Arrays)

Structures

- Verschachtelung
  - Die Typen der Membervariablen eines Structs können beliebig sein
  - Auch andere selbst definierte Typen
- Beispiel:

```
struct MyStruct {
     int i;
     MyStruct myObject;
};
```

Wieviel Speicherplatz würde ein solcher struct belegen?

  - Keine rekursive Verschachtelung!
    - Mittels Pointern erreichen wir aber das Gewünschte!

Pointer

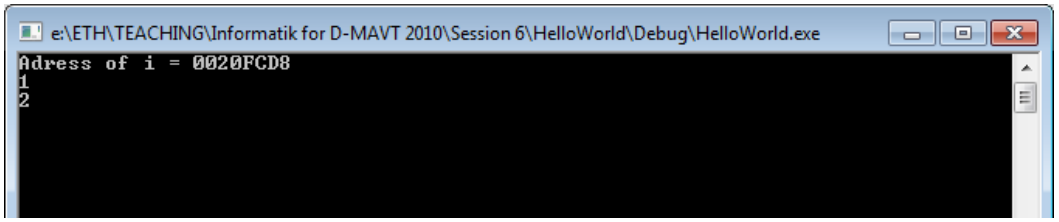
- Arrays
- Structures
- **Pointer**
- Dynamischer Speicher

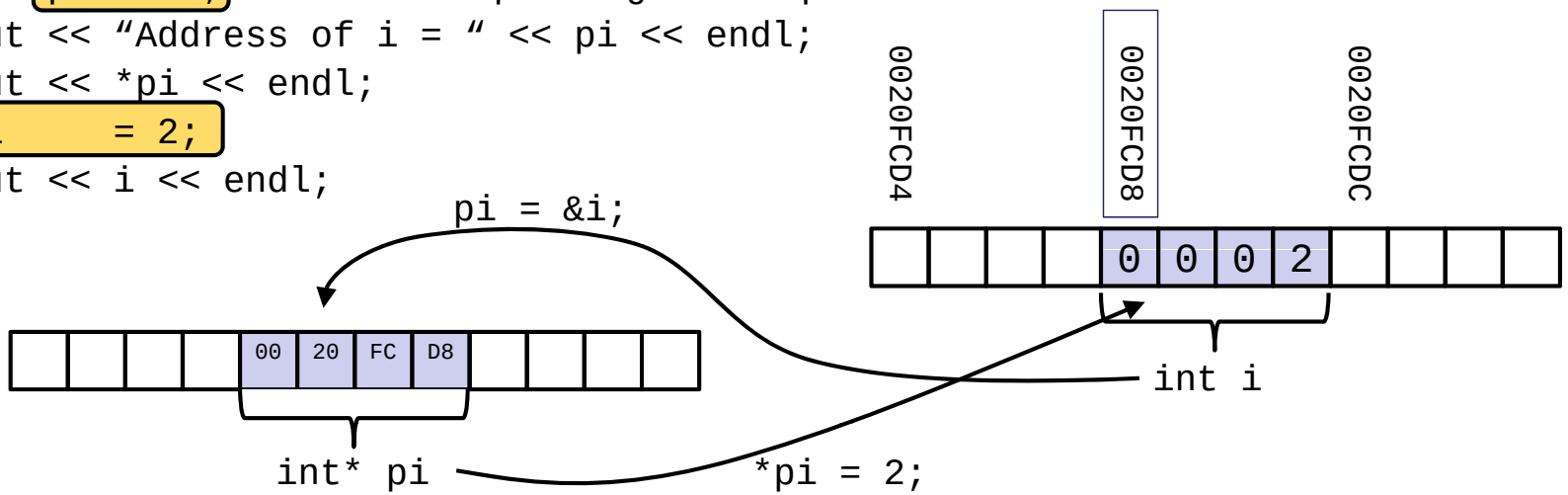
Pointer

- Zu jedem Datentyp gibt es einen entsprechenden Pointer-Datentyp (`int*, float*, MyOwnDataType*`)
  - Ein Pointer speichert eine Adresse
  - Diese Adresse entspricht dem Speicherort einer Variablen
  - Der Typ dieser Variablen ist durch den Typ des Pointers gegeben: der Basistyp des Pointers
    - Die Bytes an der vom Pointer referenzierten Speicheradresse werden gemäss diesem Typ interpretiert

Pointer

- Referenzierungsoperator (&)
  - Gibt Addresse der Variablen zurück (erstes Byte der Variablen)
    - ```
      int   i = 1;
      cout << " Address of i = " << &i << endl;
      ```
- Dereferenzierungsoperator (*)
  - Zugriff auf Speicherbereich, der vom Pointer adressiert wird
    - ```
      int   i = 1;
      int* pi = &i;          // pi zeigt auf Speicherbereich von i
      cout << "Address of i = " << pi << endl;
      cout << *pi << endl;
      *pi     = 2;
      cout << i << endl;
      ```





Pointer-Quiz

- Wieviel Speicher belegt ein Pointer?
  - Unabhängig vom Basistyp! Systemabhängig, meist jedoch 4 Bytes.
- Was geschieht bei folgendem Statement?

```
int x = 0;
*&x   = 1;
```

- Und was bei folgendem Statement?

```
int i = 0;
int* x = &i;
x = &*x;
```

Pointers

- Vorteile
  - Sehr flexibel: Pointers ermöglichen
    - komplizierte Datenstrukturen
    - Object-Oriented Programming
    - Dynamische Speicherallokation
    - ...
- Nachteile:
  - Ungültige Speicherzugriffe werden möglich
    - Überschreiben oder Auslesen von ungültigen Speicherbereichen
  - Grund für viele Bugs

Pointer Arithmetik

```
struct MyType {
    bool b;
    float f;
};
MyType a = {false, 0.0f};

MyType x[100];
MyType* p = x; // p zeigt auf &x[0]
for (unsigned int i = 0; i < 100; i++, p+=1)
    *p = a;
```

- Addition eines Integers `i` zu einem Pointer `p` addiert zum Wert des Pointers das i-fache der für den Basistyp `baseType` benötigten Speichermenge
  - `p + i == (baseType*) ( (char*) p + i*sizeof(baseType) ) // is true!`
- Pointer Arithmetik ist nützlich um über alle Elemente einer Datenstruktur zu iterieren
  - Siehe später auch Konzept von Iteratoren…

Const-Pointer

- Pointer auf konstante Variable
  - `const int* pci; // pointer pi refers to const int`
- Constant Pointer auf Variable
  - `int* const cpi; // constant pointer cpi refers to int variable`
- Frage: was ist `const int* const cpci` ?
  - Constant pointer auf constant int.

Const-Pointer Quiz

- ```
  int i;
  const int ci;
  int* pi = NULL;
  const int* pci = NULL;
  ```
- ```
  pi = &i;
  *pi = 3;
  ```
- ```
  pci = &i;

  *pci = 3;
  ```
- ```
  pi = &ci;
  pci = &ci;
  ```
- ```
  pci = &ci;
  *pci = 3;
  ```
- ```
  int* const cpi = NULL;
  cpi = &i;
  ```
- ```
  int* const cpi = &i;
  const int* const cpci = &ci;
  ```
  ```

Anonyme Pointer

- Wie kann ein Pointer definiert werden, dessen Basistyp nicht bekannt ist?
  - `void* ptr;`
  - Zeiger auf einen Speicherbereich ohne spezifizierten Basistyp
- Beispiel:
  - ```
    int* pi;
    void* ptr = pi;     // ok.
    pi = (int*) ptr;    // explicit type cast required
    ```
- Was passiert bei:

  ```
  void* ptr;
  ptr = ptr + 1;
  ```
  ```

Pointers & Structs

- Zurück zum Beispiel der verschachtelten structs:

```
struct MyStruct {
     int i;
     MyStruct myObject;
};
```

Nicht möglich!

- Aber mittels Pointern möglich:

```
struct MyStruct {
     int i;
     MyStruct* myObject;
};
```

- Pfeiloperator

```
MyStruct m;
MyStruct* pm = &m;
pm->i; // short-hand for (*pm).i
```

Dynamischer Speicher

- Arrays
- Structures
- Pointer
- **Dynamischer Speicher**

Dynamischer Speicher: Wieso?

```
cout << “Wieviele Ziffern hat deine Telefonnummer?“ << endl;
int n;
cin >> n;
 /* Dynamischer Allokation des Arrays auf dem Heap */
int* telnr = new int[n];

cout << “Gebe jetzt bitte deine Telefonnummer ziffernweise ein“ << endl;
for (int i=0; i < n; i++)
      cin >> telnr[i];
...
// Freigeben des dynamischen Speichers wenn er nicht mehr benutzt wird:
delete[] telnr;
```

- Probleme:
  - Was passiert für Telefonnummern mit mehr als 20 Ziffern?
  - Verschwendung von Speicherplatz für Telefonnummern mit weniger als 20 Ziffern
- Lösung: dynamische Arrays variabler Länge

Dynamischer Speicher: Wieso?

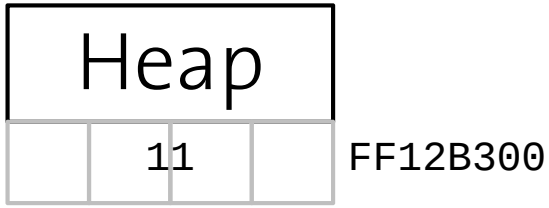
- Rückgabewerte von Funktionen
  - Datenstrukturen, komplexer als basic types (`bool`, `int`, ...)
- Rückgabe von mehreren Werten
  - Funktionen können lediglich eine einzige Variable mittels `return` zurückgeben
- Details siehe später wenn Funktionen behandelt werden...

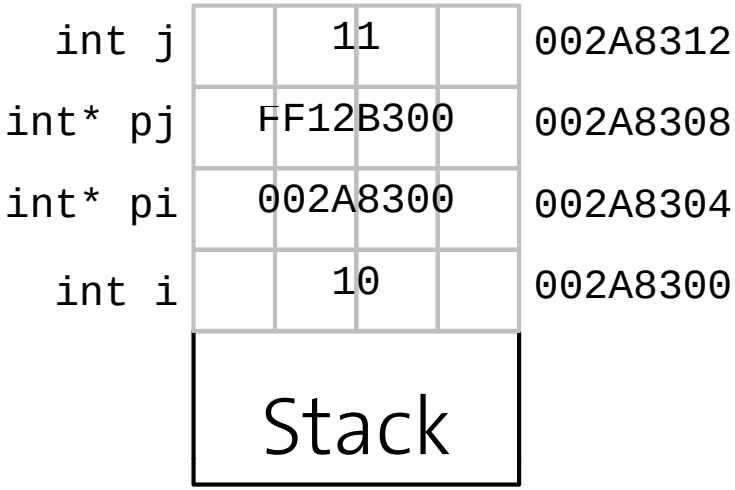
Arten von Variablen

- Statische Variablen / Spezielle Variablen
  - Im Moment nicht von Interesse
- Lokale Variablen
  - Variablen, die lediglich innerhalb eines Codeblocks gültig sind (z.B. innerhalb einer Schleife oder einer Funktion)
  - Compiler kümmert sich um Speicherallokation und –deallokation auf dem Stack
- Dynamische Variablen
  - Variablen, deren Speicherplatz im sogenannten *free store* (meist auf Heap) explizit mittels `new` alloziert wird und mittels `delete` freigegeben wird

Stack vs. Heap

```
int i = 10;
int* pi = &i;
int* pj = new int;
{
  int j = 11;
  *pj = j;
}
delete pj;
```





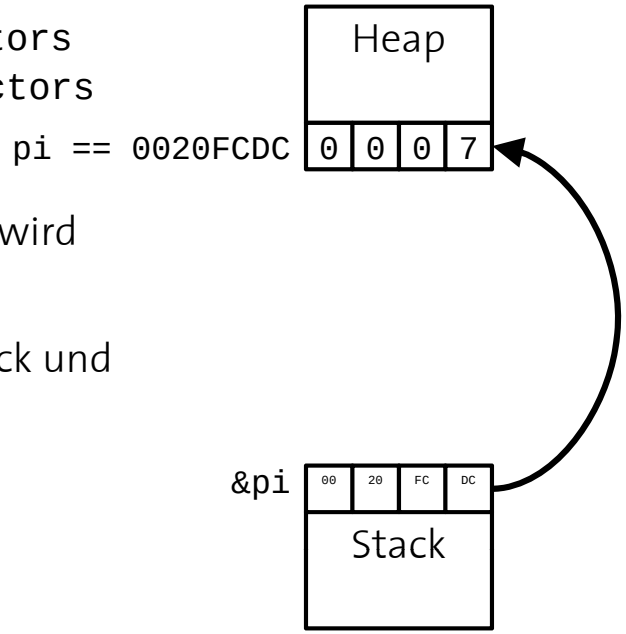
Pointer stellen Verbindung zwischen Stack und dynamischen Speicherbereich (free store) her

Dynamische Speicherallokation

- Operator **new**
  1. Reserviert Speicherbereich für neue Variable (Standardmässig im Heap)
  2. Ruft einen Constructor der Variablen auf (Details siehe später...)
  3. Gibt einen Pointer auf den neuen Speicherbereich zurück
  - Beispiel:

```
int* pi = new int; // Aufrufen des Default-Constructors
int* pi = new int(7);  // Aufrufen des Copy-Constructors
```

  - Speicher für Integer Pointer `pi` wird im Stack reserviert
  - Speicher für den vom Pointer referenzierten Integer jedoch wird im Heap reserviert
  - Pointer stellt somit Verbindung zwischen Variablen auf Stack und free store her



Dynamische Speicherallokation

- Operator `delete`
  1. Aufrufen des Destruktors des zu löschenden Objects (siehe später)
  2. Freigeben des zuvor mittels `new` reservierten Speicherbereichs
- Funktioniert auch für Arrays (mit angepasster Syntax):
  -
```
// n muss nicht zur Compilezeit bekannt sein!
int* ai = new int[n];
...
delete[] ai;
```

Low-Level Details (Advanced)

- `new` und `delete` sind Operatoren
  - Definiert im Headerfile <new>
  - Definiert jeweils 3 Versionen für `new, delete, new[], delete[]`
  - Die zwei wichtigsten Versionen sind:

```
void* operator new (std::size_t size) throw (std::bad_alloc);
```

    - "allocates *size* bytes of storage space, aligned to represent an object of that size, and returns a non-null pointer to the first byte of this block. On failure, it throws a bad_alloc exception." [http://www.cplusplus.com/reference/std/new/operator%20new/]

```
void  operator delete (void* ptr) throw ();
```

    - "deallocate the memory block pointed by *ptr* (if not-null), releasing the storage space previously allocated to it by a call to operator new and making that pointer location invalid." [http://www.cplusplus.com/reference/std/new/operator%20delete/]

Low-Level Details (Advanced)

- `void* operator new (std::size_t size) throw (std::bad_alloc);`
  - "operator new can be called explicitly as a regular function, but in C++, new is an operator with a very specific behavior: An expression with the new operator, first calls function operator new with the size of its type specifier as first argument, and if this is successful, it then automatically initializes or constructs the object (if needed). Finally, the expression evaluates as a pointer to the appropriate type." [http://www.cplusplus.com/reference/std/new/operator%20new/]
  - Beispiel:
    ```
    MyClass* p = (MyClass*) operator new (sizeof(MyClass));
    ```
    - Dadurch wird aber Constructor von MyClass **nicht** aufgerufen!
- `void  operator delete (void* ptr) throw ();`
  - "operator delete can be called explicitly as a regular function, but in C++, delete is an operator with a very specific behavior: An expression with the delete operator, first calls the appropriate destructor (if needed), and then calls function operator delete to release the storage." [http://www.cplusplus.com/reference/std/new/operator%20delete/]

Vorbesprechung: Blatt 6, Aufgabe 1

- Structs & Pointers
  - Initialisierung
  - Zuweisungen
  - Referenzieren & Dereferenzieren

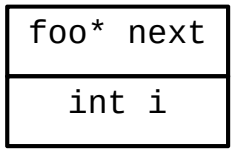
```
int main () {
  int n_lokal;
  int* ip_a = &n_lokal;
  int** ipp_b = &ip_a;
  point green = {0.75f, 1.0f, {0,255}};
  point blue = {0.5f,0.5f, {0,0,255}};
  point red = {1.0f, 0.75f, {255}};
  point* tmp;
  tmp = &blue; // a)
  ...
```

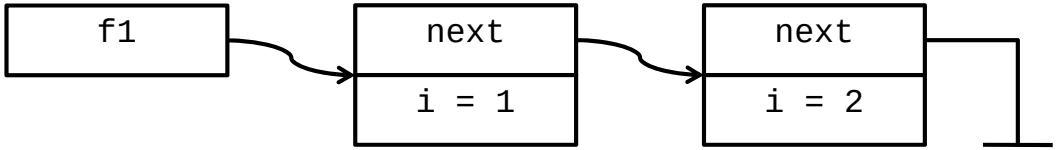
  - Antwort für a)

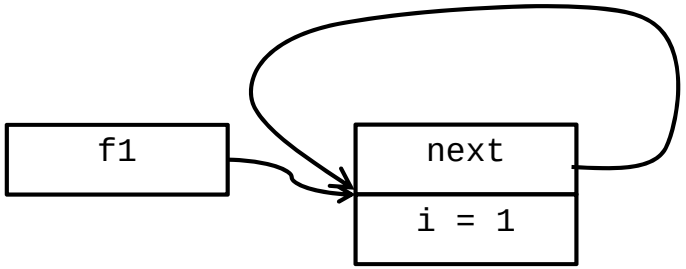
    ip_a: Zeiger auf n_lokal
    ipp_b: ...
    green: ...
    ...

Vorbesprechung: Blatt 6, Aufgabe 2

- „Verkettete“ Structures
  - ```
    struct foo {
      foo* next;
      int i;
    };
    ```
  - ```
    foo *f1 = NULL;
    f1 = new foo;
    f1->i = 1;
    f1->next = new foo;
    f1->next->i = 2;
    f1->next->next = NULL;
    ```
  - ```
    foo *f1 = NULL;
    f1 = new foo;
    f1->i = 1;
    f1->next = f1;
    ```






    ```

Vorbesprechung: Blatt 6, Aufgabe 3

- Arrays dynamisch allozieren
  - Siehe Beispiel bei Besprechung von dynamischen Speicher (Telefonnummer)
- Tipps:
  - Indices starten in C++ bei `0` und gehen bis `n-1`
  - Nicht vergessen den dynamisch allozierten Speicher am Ende des Programms wieder frei zu geben!

Vorbesprechung: Blatt 6, Aufgabe 4

- Datenstruktur zum Lösen eines Problems entwerfen
- Structures erlauben, eigene Datentypen zu definieren
- Seid kreativ!
  - Aber nicht zu kreativ… ☺

Fragen

- Zur Vorlesung?
- Zur Übung?