Praxisorientierte Einführung in C++
Lektion: "Dynamische Speicherverwaltung"

Christof Elbrechter

Neuroinformatics Group, CITEC

May 15, 2014

Table of Contents

Allgemeines

- Auf modernen Computern laufen viele Programme "gleichzeitig"
- Speicher ist eine Resource, die geteilt werden muß
- Betriebssystem verteilt Speicher an Anwendungen
- Anwendungen geben Speicher wieder frei (bei Programmende oder explizit)

Statische Speicherverwaltung

- Bisher: Lebenszeit aller Variablen entweder auto oder static
  - `auto`: Im Block in dem sie deklariert wurden
  - `static`: Beim ersten Aufruf der Funktion bis Programmende
  - Oder global: Lebenszeit entspricht der Lebenszeit des Programms

Dynamische Speicherverwaltung - Warum?

- Wenn Datenmenge, die zu verarbeiten ist, erst zur Laufzeit bestimmt ist:
- Viel Speicher "bunkern". Unpraktisch, weil:
  - Speicher eine "rare" Resource ist
  - Der Speicher vielleicht trotzdem nicht ausreicht
- Daher: Dynamische Speicherverwaltung
  - Programm kann zur Laufzeit Speicher reservieren und freigeben
  - C++ unterstützt dies mit eingebauten Operatoren

Dynamische Speicherverwaltung in C

- In C: `malloc()` und `free()`

```
void *malloc(AnzahlBytes);
```

- Gibt Zeiger auf Speicher für Anzahl Bytes (`char`) zurück
- Typunsicher, fehlerträchtig

```
void free(Zeiger);
```

- Gibt Speicher frei, auf den der Zeiger zeigt
- Was mit `malloc()` reserviert wurde muss mit `free()` freigegeben werden

malloc und free Beispiele

- **Wichtig:** Das hier dient nur zur Erklärung, wie das ganze in C funktioniert hat
- In C++ sollte mann malloc und free i.d.R. **nicht** verwenden
- Es sei denn mann verwendet C-Bibliotheken, die dieses verlangen

malloc = "memory allocate"

```
#include <malloc.h>
int main(){
  // reserviere Speicher fuerr 100 Integers
  int *n = (int*)malloc(100 * sizeof(int));

  // Speicher kann verwendet werden
  for(int i=0;i<100;++i){
    n[i] = 0;
  }

  // gibt Speicher wieder frei
  free(n);
}
```

malloc und free für Strukturen

Kleiner Vorausgriff[1]

Eine einfache 2D-Point Struktur

```
struct Point {
   /// sog. Standard-Konstruktor
   Point(){
      this->x = 0;
      this->y = 0;
   }
   /// spezieller Konstruktor
   Point(int x, int y){
      this->x = x;
      this->y = y;
   }
   int x;
   int y;
};

int main(){
     Point p;         // keine Argumente -> Standard-Konstruktor (setzt x und y auf 0)
     Point p2(3,4);   // mit Argumenten -> spezieller Konstruktor (setzt x auf 3 und y auf 4)
}
```

[1] aber Strukturen kommen ja gleich

malloc und free für Strukturen

Dynamische Anzahl an Punkten allozieren

```
int main(){
  Point *p = (Point*)malloc(100*sizeof(Point));
  // ABER: welcher Konstruktor wurde fuer die Elemente aufgerufen?
  // Antwort: gar keiner!

  // Elemente muessen "von Hand" initialisiert werden
  for(int i=0;i<100;++i){
    p[i] = Point();
  }

  free(p);
}
```

memset

- Für binäre Speicherinitialisierung

```
void memset(void *data, int byteValue, size_t numberOfBytesToSet);
```

C-Funktion memset

```
#include <cstring>

struct Point { ... };

int main(){
  Point *p = (Point*)malloc(100*sizeof(Point));

  // Setzt Speicher Byte-weise
  memset(p,0,100*sizeof(Point));

  free(p);
}
```

Die Operatoren `new`, `new[]`, `delete` und `delete[]`

- In C++: `new`, `new[]`, `delete` und `delete[]`
  - Mit `new` und `new[]` wird Speicher reserviert (und Konstruktor(en) aufgerufen)
  - Mit `delete` und `delete[]` wird Speicher freigegeben (und Destruktor(en) aufgerufen)
- Einfachste Form:

```
new X // X ist ein Typ
```

- Evaluiert zu einem Zeiger vom Typ `X*`
- Beispiel:

```
int *i = new int;
```

new

- `new X` reserviert genug Speicher, damit ein Objekt vom Typ `X` hineinpasst
  - Kann aber auch mehr sein. C++-Runtime muss mit Betriebssystem umgehen
- Der zurückgegebene Zeiger ist ein "Schlüssel", um an den Speicher (Das Objekt) zu kommen...
- ...und auch, um ihn wieder freizugeben
  - Das *vergessen* eines Zeigers führt zu Speicherlecks

new - Konstruktoraufrufe

Wichtig!

- Im Gegensatz zu der malloc-Variante, ruft new immer auch einen Konstrukor auf
- Entweder explizit speziellen Konstruktor ...
- ... oder implizit den Standard Konstruktor

```
int main(){
  // Erinnerung: Variablen auf dem Stack anlegen
  int i(5);
  Point p(3,4);

  // Auf dem Heap (mit new)
  int *i1 = new int;          // Standard int-Konstruktor (macht gar-nichts)
  int *i = new int(5);        // spezieller Konstruktor fuer int

  Point *p = new Point(3,4); // spezieller Konstruktor
  Point *p2 = new Point();   // Standard Konstruktor
  Point *p3 = new Point;     // Auch Standard Konstruktor (aequivalent)
}
```

Zugriff auf erstellte Objekte

- Der Zugriff auf die erstellten Objekte erfolgt mithilfe des Dereferenzierungsoperators *
- Genau, wie bei Zeigern, auf Elemente von Arrays

Beispiel

```
int *i = new int;
std::cout << *i << std::endl;
*i = 1;
std::cout << *i << std::endl;
```

- Bei Objekten kann alternativ auch der Zeiger-Operator verwendet werden

Zeiger-Operator

```
Point *p = new Point;
(*p).x = 4;             // mittels Dereferenzierung (aber: sehr schlechter Stil)

p->y = 3;               // Besser: Zeiger Operator
```

delete

- Freigabe des Speichers mit `delete`

```
delete Ausdruck
```

- `Ausdruck` ist Ausdruck, der zu einem Zeiger evaluiert.
- Beispiel:

```
int *i = new int(3);
delete i;
```

- Wenn es nicht zu jedem `new` ein korrespondierendes `delete` gibt, hat man Speicherlecks
  - Kann beliebig kompliziert werden

new []

- Mit new reserviert man Speicher für (und konstruiert) jeweils nur *ein* Objekt
- Wenn man weiß, dass man viele braucht gibt es new []

```
int *is = new int[100];  // 100 ints
Point *ps = new Point[100];   // 100 Points
```

- Damit hat man auch endlich eine Möglichkeit, dynamische Arrays anzulegen

Welcher Konstruktor?

- new [] ruft immer für alle Elemente den Standard Konstruktor auf
- D.h. es muss einen Standard Konstruktor geben

new[] und Konstruktoren

- Wie gesagt: new [] ruft für alle Elemente den Standardkonstruktor auf
- Können auch andere Konstruktoren aufgerufen werden?

```
stuct Point{  // Beispiel Stuktur
  int x,y;
  Point(){  // Standardkonstruktor
    this->x = 0;
    this->y = 0;
  }
  Point(int x, int y){
    this->x = x;
    this->y = y;
  }
};

int main(){
   Point *ps1 = new Point[5];   // alle (0,0) -> Standardkonstruktor
   Point *ps2 = new Point[5](); // geht auch: gleiches Ergebnis
   Point *ps3 = new Point[5](3,4);  // nicht gueltig (im gcc mit -fpermissive)
}
```

new[] und Konstruktoren

- Bei Standarddatentypen (wie z.B. `int` und `char`) macht der Standardkonstruktor **gar nichts**
- Speicher bleibt un-initialisiert
- Um mit `new []` initialisierten Speicher mit 0 zu initialisieren:
  - Konstruktorklammern mit angeben
  - Gilt nur für Standarddatentypen

```
int *as = new int[1000]; // uninitialisierter Speicher (LEIDER auch fast immer komplett 0)
int *bs = new int[1000](); // definitiv komplett 0
```

Arrayinitialisierung mit definiertem Wert

- Wie kann man denn nun N Elemente eines Typs mit einem bestimmten Initialisierungswert erstellen

Arrayinitialisierung mit definiertem Wert

- Wie kann man denn nun N Elemente eines Typs mit einem bestimmten Initialisierungswert erstellen
- $\rightarrow$ gar nicht! :-)

Arrayinitialisierung mit definiertem Wert

- Wie kann man denn nun N Elemente eines Typs mit einem bestimmten Initialisierungswert erstellen
- → gar nicht! :-)
- Ausweg: `std::vector`
  - Sehr sehr hilfreiches *Klassentemplate*
  - Enthält N elemente eines bestimmten Types

```
std::vector<float> v;  // 0 floats
std::vector<Point> p(5); // 5 Points alle standard-konstruiert
std::vector<Point> p(5,Point(3,4)); // 5 Points alle (3,4)

// sehr cool:
p.push_back(Point(5,6)); // haengt Element hinten an
p.resize(77);            // Groesse aendern
```

- Zum `std::vector` später noch (viel) mehr

Zugriff auf Elemente

- Zugriff auf Elemente erfolgt, genau wie bei Arrays fester Größe, mithilfe des Index-Operators

```
char *s = new char[100];
Point *p = new Point[100];
for (int i = 0; i < 100; ++i) {
  s[i] = 0;
  p[i] = Point(i,0);
  p[i].y = 7;
}
```

- ... oder mittels Zeigerarithmetik (aber: auch sehr schlechter Stil)

```
char *s = new char[100];
for (int i = 0; i < 100; ++i) {
  *(s+i) = 0;
}
```

delete[]

- Das Gegenstück zu `new[]` ist `delete[]`
- Speicher, der mit `new[]` reserviert wurde, *muss* mit `delete[]` wieder freigegeben werden
  - Sonst: Undefiniertes Verhalten
- bei `delete []` muss (und darf) die Anzal der Elemente, die glöscht werden sollen, **nicht** mit angegeben werden
- Analog zu `delete`, ruft `delete []` auch immer die sog. Destruktoren für jedes Element auf
- Das ist insbesondere wichtig, wenn Objekte selbst dynamisch Speicher allozieren[2]

```
int *myArray = new int[1024];
// ...
delete[] myArray;
```

[2] zu Destruktoren aber später mehr

Wichtig!

(Sehr) Wichtig!

- Speicher der mit `new` angelegt wurde **muss** mit `delete` freigegeben werden
- Speicher der mit `new []` angelegt wurde **muss** mit `delete []` freigegeben werden

- Wenn mit `new []` mehrere Elemente alloziert werden, muss `delete []` mit der Adresse des ersten Elementes aufgerufen werden

```
int *p = new int[77];
delete [] (p+5);  // ---->undefiniertes Verhalten!
```

Wann sollte man `new` oder `new []` verwenden

Faustregel

Verwende den Stack falls möglich!

- Allozieren und Freigeben von Speicher mit `new`/`new []` relativ langsam
- Wenn die Anzal von Elementen im Speicher bekannt und nicht extrem groß ist, verwende den Stack
- Beispiel: 3D- vs ND Vektor-Klasse
  - wenn man die Dimension der Vektoren nicht vorher weiß: Heap und `new`
  - wenn man aber weiss, dass man für ein Problem immer nur 3D-Vektoren benötigt ...
  - ... sollte man versuchen, auf dem Stack zu arbeiten
  - Das kann bestimmte Sachen locker 10 mal schneller machen

Performance Vergleich Stack vs. Heap

```
#include <ctime>
#include <iostream>
struct Data { char b[1024]; };
int main(){
  int NUM = 10000000;
  std::clock_t start = std::clock();
  for (int i = 0;i<NUM;++i){
    Data d;
  }
  std::clock_t dt = std::clock() - start;
  std::cout << "Stack Time " << float(dt)/CLOCKS_PER_SEC << " sec\n";

  start = std::clock();
  for(int i = 0; i<NUM; ++i) {
    Data *d = new Data;
    delete d;
  };
  dt = std::clock() - start;
  std::cout << "Heap Time " << float(dt)/CLOCKS_PER_SEC << " sec\n";
}
```

Ausgabe (Stack ist mehr als 20 mal schneller)

```
Stack Time 0.03 sec
Heap Time 0.7 sec
```

Nachtrag ..

- Ok, der Fairness halber muss erwähnt werden, dass das ganze natürlich stark von der Größe von Data abhängt
- Tatsächlich dauert das Allozieren von Speicher auf dem Stack immer gleich lang
- Es muss ja nur der Stack-Pointer verschoben werden
- Das Allozieren auf dem Heap skaliert aber mit der Größe des zu allozierenden Speichersegments

Unterschiedlich große Objekte

Größe in Bytes	1	100-500000	ab ca. 600000
Zeit Stack [sec]	0.02	0.02	0.02
Zeit Heap [sec]	0.48	0.7	55

- Tatsächlich ist das hier noch untertrieben, da keine wirkliche Fragmentierung des Speichers auftritt

new / delete

- Das war nicht die ganze Wahrheit
  - inplace-new u.A.
- Nach Strukturen / Klassen wissen wir mehr
  - Konstruktoren / Destruktoren
- Smart Pointer / Garbage Collection