Einführung in die Programmierung mit C++

Dynamische Speicherverwaltung

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Inhalt

Outline

Dynamische Speicherverwaltung

Allozierung. Initialisierung. Deallozierung


```
1  int main(int argc, char*[]) {
2    using namespace std;
3    if (argc>1) { // dynamics
4      int *p=new int; // allocation
5      *p=42; // initialization
6      cout << "&p=" << &p << endl;
7      cout << "p=" << p << endl;
8      cout << "*p=" << *p << endl;
9      cout << "&*p=" << &*p << endl;
10     delete p; // deallocation
11   }
12   return 0;
13 }
```

erzeugt bei Aufruf mit mindestens einem Kommandozeilenargument z.B. die folgende Ausgabe:

```
&p=0x7ffd46b7f030
p=0x55da6554deb0
*p=42
&*p=0x55da6554deb0
```

- Oft steht erst zur Laufzeit des Programms fest, wieviel Hauptspeicher benötigt wird, d.h. dieser kann nicht statisch (benannt auf *stack*) alloziert werden.
- Zeiger erlauben Allozierung (mittels new) von und Operationen (mittels Dereferenzierung) auf dynamischem (unbenanntem) Hauptspeicher (auch: *heap*)
- Dynamisch allozierter Speicher wird bei Verlassen des Gültigkeitsbereichs der Zeigervariablen nicht automatisch freigegeben; dies muss explizit mittels delete geschehen; sonst: *memory leaks*

Dynamische Speicherverwaltung

Varianten


```
1  int main(int argc, char*[]) {
2    if (argc>1) {
3      const int *p=new int(42);
4      int * const q=new int(42);
5      const int * const r=new int(42);
6
7      // ok:
8      ++p;
9      ++*q;
10     // errors:
11     ++*p;
12     ++q;
13     ++r;
14     ++*r;
15
16     delete r;
17     delete q;
18     --p;
19     delete p;
20   }
21   return 0;
22 }
```

- Exklusiver Lesezugrif auf Zeiger bzw. den durch sie referenzierten dynamischen Speicher kann mittels **const** erzwungen werden.
- Es ergeben sich diverse Kombinationen.
- Die systematische Verwendung von **const** ist anzuraten, da sicherer bezüglich Fehlverwendung, besser verständlich und potentiell effizienter.

AUFWACHEN!

Terminiert folgendes Programm bei Aufruf mit mindestens einem Kommandozeilenargument?

```
1  template<typename T>
2  void f(T*& p) { p=new T; }
3
4  int main(int argc, char*[]) {
5    if (argc>1) {
6      double *p=nullptr, *q=p+1;
7      while (p!=q) { q=p; f(p); }
8    }
9    return 0;
10 }
```

Wach?

Nein[1], aber folgendes schon:

```
1  template<typename T>
2  void f(T*& p) { p=new T; }
3
4  int main(int argc, char*[]) {
5    if (argc>1) {
6      double *p=nullptr, *q=p+1;
7      while (p!=q) { q=p; f(p); delete p; }
8    }
9    return 0;
10 }
```

WACH BLEIBEN! Wieviele Schleifeniterationen werden durchgeführt?

[1] bzw. Abbruch bei Erreichen des Speicherlimits

Noch Wach?

```
1  #include<iostream>
2
3  template<typename T>
4  void f(T*& p) { p=new T; }
5
6  int main(int argc, char*[]) {
7    if (argc>1) {
8      double *p=nullptr, *q=p+1;
9      while (p!=q) { q=p; f(p); delete p;
10       std::cout << p << " " << q << std::endl;
11     }
12   }
13   return 0;
14 }
```

führt zwei Schleifeniterationen durch und erzeugt dabei z.B. die Ausgabe

```
0x5636d39cceb0 0
0x5636d39cceb0 0x5636d39cceb0
```

Dynamische Speicherverwaltung

Objekte nutzerdefinierten Typs im dynamischer Speicher


```
1  template<typename F, typename S>
2  class paar {
3    F f; S s;
4  public:
5    paar(F f, S s) : f(f), s(s) {}
6  };
7
8  int main(int argc, char*[]) {
9    if (argc>1) {
10     paar<int,double> *p=
11       new paar<int,double>(42,3.14);
12     // do something useful
13     delete p;
14   }
15   return 0;
16 }
```

- Notwendigkeit der Allozierung von p erst zur Laufzeit (dynamisch) des Programms
- Zugriff auf Objektelemente (Daten und Funktionen) mittels `->` Operator, z.B. `p->s` (erfordert **friend**)
- Korrekte Deallozierung durch Standarddestruktor des Objekttyps (hier der Klasse paar)

AUFWACHEN!

Kann folgendes Programm übersetzt und fehlerfrei ausgeführt werden?

```
1  template<typename F, typename S>
2  class paar {
3    F f; S s;
4  public:
5    paar(F f, S s) : f(f), s(s) {}
6  };
7
8  int main(int argc, char*[]) {
9    if (argc>1) {
10     paar<int,double> *p=new paar<int,double>('a',42);
11     p=new paar<int,double>('b',24);
12     // do something useful
13     delete p;
14     delete p;
15   }
16   return 0;
17 }
```

Falls nicht, dann warum nicht?

Wach?

```
1  template<typename F, typename S>
2  class paar {
3    F f; S s;
4  public:
5    paar(F f, S s) : f(f), s(s) {}
6  };
7
8  int main(int argc, char*[]) {
9    if (argc>1) {
10     paar<int,double> *p=new paar<int,double>('a',42);
11     p=new paar<int,double>('b',24); // memory leak
12     // do something useful
13     delete p;
14     delete p; // double deallocation
15   }
16   return 0;
17 }
```

Bei Aufruf mit mindestens einem Kommandozeilenargument kommt es zu einem Laufzeitfehler aufgrund doppelter Deallozierung.

Outline

- Konsekutiver Speicherplatz für n Werte vom Typ T, d.h. (mindestens) n*sizeof(T) Bytes, wird im dynamischen Speicher reserviert.
- Zugriff auf i+1-tes Element erfolgt durch a[i].
- Initialisierung erfolgt durch Standardkonstruktor bzw. explizite Zuweisung.

```
1  int main(int argc, char*[]) {
2    int *a=new int[argc]; // allocation
3    for (size_t i=0;i<argc;i++)
4      a[i]=i; // initialization
5    // do something useful
6    delete [] a; // deallocation
7    return 0;
8  }
```

- Manuelle (keine automatische) Freigabe des allozierten Speichers (bei Verlassen des Gültigkeitsbereichs) von a erfolgt mittels delete [] a.
- Live → Inspektion des Speicherlayouts mit gdb und Visualisierung

Dynamische Felder sind als Zeiger (per Zeigerarithmetik modifizierbar) auf die Basisadresse des ersten Elements (liegt im dynamischen Speicher) implementiert.

```
1  #include<iostream>
2
3  int main(int argc, char*[]) {
4    int *a=new int[argc];
5    auto i=0;
6    for (;i<argc;i++) a[i]=i;
7    for (;i>0;i--)
8      std::cout << *(a+i-1) << std::endl;
9    for (;i<argc;i++)
10     std::cout << *(a++) << std::endl;
11   delete [] a; // corrupt pointer
12   return 0;
13 }
```

- Auf den Inhalt des ersten Elements eines dynamischen Feldes `a` kann, analog zu statischen 1D Feldern, alternativ mittels `a[0]` oder `*a` zugegriffen werden.
- Durch `a[i]` wird die Adresse `&a+i*sizeof(int)` dereferenziert. Alternativ kann man auch `*(a+i)` schreiben (Zeigerarithmetik).

Analog zu statischen Feldern findet zur Übersetzungszeit keine automatische Überprüfung der Zulässigkeit eines Elementzugriffs statt.

```
1  #include<iostream>
2
3  int main(int argc, char*[]) {
4    int *a=new int[argc];
5    for (auto i=0;i<argc<<1;i++) a[i]=i;
6    for (auto i=0;i<argc<<2;i++)
7      std::cout << a[i] << " ";
8    std::cout << std::endl;
9    delete [] a;
10   return 0;
11 }
```

Hier: shift-Operation `i << n` auf **int** `i` verschiebt die Binärdarstellung um `n` Stellen nach links (siehe auch to_bin), d.h. es ergibt sich bei Aufruf mit einem Kommandozeilenargument z.B. folgende Ausgabe

```
0 1 2 3 0 0 1041 0
```

AUFWACHEN!

Was wird bei Aufruf mit Kommandozeilenargument

1000000000

durch folgendes Programm (qualitativ) ausgegeben?

```
1  #include<iostream>
2
3  int main(int argc, char *argv[]) {
4    using namespace std; using T=double;
5    if (argc==2)
6      for (;;) {
7        T *p=new T[stoi(argv[1])];
8        cout << "&p=" << &p << endl
9             << "p=" << p << endl;
10     }
11   return 0;
12 }
```

Wach?

```
1  #include<iostream>
2
3  int main(int argc, char *argv[]) {
4    using namespace std; using T=double;
5    if (argc==2)
6      for (;;) {
7        T *p=new T[stoi(argv[1])];
8        cout << "&p=" << &p << endl
9             << "p=" << p << endl;
10     }
11   return 0;
12 }
```

```
&p=0x7fff99ad2838
p=0x7fb58ed13010
&p=0x7fff99ad2838
p=0x7fb3b1fad010
...
&p=0x7fff99ad2838
p=0x7ffc56fc2010
terminate called after throwing
    an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted (core dumped)
```

`&p` bleibt gleich; `p` wächst stetig bis zum Speicherlimit

Outline

Dynamische nD-Felder

Allozierung. Initialisierung. Deallozierung. (z.B. $n = 2$)


```
1  int main(int argc, char*[]) {
2    int **a=new int*[argc];
3    for (auto i=0;i<argc;i++) {
4      a[i]=new int[argc-i];
5      for (auto j=0;j<argc-i;j++)
6        a[i][j]=i+j;
7    }
8    // do something useful
9    for (auto i=0;i<argc;i++)
10     delete [] a[i];
11   delete [] a;
12   return 0;
13 }
```

Z.B. bei Aufruf mit zwei Kommandozeilenargumenten

```
a[0][0]=0
a[0][1]=1
a[0][2]=2
a[1][0]=1
a[1][1]=2
a[2][0]=2
```

- Dynamische nD-Felder werden als dynamische 1D-Felder von Zeigern auf dynamische 1D-Felder von Zeigern auf ... auf dynamische 1D-Felder des entsprechenden Datentyps alloziert.
- Die Deallozierung erfolgt in umgekehrter Reihenfolge.
- Der Elementzugriff erfolgt über verkettete Anwendung des Dereferenzierungsoperators []. Dabei erfolgt keine automatische Überprüfung der Zulässigkeit dieser Zugriffe durch den Compiler.
- Live → Inspektion des Speicherlayouts mit gdb und Visualisierung

Dynamische *n*D-Felder

Alternativer Elementzugriff


```
1  #include<iostream>
2
3  int main(int argc, char*[]) {
4    int **a=new int*[argc];
5    for (auto i=0;i<argc;i++) {
6      a[i]=new int[argc-i];
7      for (auto j=0;j<argc-i;j++) a[i][j]=i+j;
8    }
9    for (auto i=0;i<argc;i++)
10     for (auto j=0;j<argc-i;j++)
11       std::cout << "a[" << i << "][" << j
12             <<"]=" << *(*(a+i)+j) << std::
                 endl;
13   // do something useful
14   for (auto i=0;i<argc;i++) delete [] a[i];
15   delete [] a;
16   return 0;
17 }
```

- Der Zugriff auf die Elemente dynamischer mehrdimensionaler Felder kann alternativ mittels Zeigerarithmetik und expliziter Dereferenzierung der entsprechenden Zeiger erfolgen.
- Wie zuvor erfolgt keine automatische Überprüfung der Zulässigkeit dieser Speicherzugriffe durch den Compiler.

Dynamische *n*D-Felder

Zeigerarithmetik

Im Gegensatz zu statischen mehrdimensionalen Feldern sind die Elemente dynamischer mehrdimensionaler Felder nicht konsekutiv im Speicher abgelegt.

```
1  #include<iostream>
2
3  int main(int argc, char*[]) {
4    int **a=new int*[argc];
5    for (auto i=0;i<argc;i++) {
6      a[i]=new int[argc-i];
7      for (auto j=0;j<argc-i;j++) a[i][j]=i+j;
8    }
9    int *p=&a[0][0];
10   for (auto i=0;i<argc;i++)
11     for (auto j=0;j<argc-i;j++)
12       std::cout << "a[" << i << "][" << j
13             <<"]=" << *p++ << std::endl;
14   // do something useful
15   for (auto i=0;i<argc;i++) delete [] a[i];
16   delete [] a;
17   return 0;
18 }
```

Nebenstehendes Programm produziert bei Aufruf mit einem Kommandozeilenargument z.B. die Ausgabe

```
a[0][0]=0
a[0][1]=1
a[1][0]=0
```

und nicht, wie ggf. erwartet,

```
a[0][0]=0
a[0][1]=1
a[1][0]=1
```

Dynamische *n*D-Felder

Beispiel: 3-Tensor


```
1  int main(int argc, char*[]) {
2    int ***a=new int**[argc];
3    for (auto i=0;i<argc;i++) {
4      a[i]=new int*[argc];
5      for (auto j=0;j<argc;j++) {
6        a[i][j]=new int[argc];
7        for (auto k=0;k<argc;k++) {
8          a[i][j][k]=i+j+k;
9          std::cout << "a[" << i << "][" << j << "][" << k <<"]="
10                   << a[i][j][k] << std::endl;
11       }
12     }
13   }
14   // do something useful
15   for (auto i=0;i<argc;i++) {
16     for (auto j=0;j<argc;j++) delete [] a[i][j];
17     delete [] a[i];
18   }
19   delete [] a;
20   return 0;
21 }
```

Outline

- Dynamische 1D Felder sind Zeiger und können daher wie Zeiger übergeben werden, z.B.

```
1 template<typename T> void f2(int n, T *a) { ... }
2 template<typename T> void f3(int n, T*& a) { ... }
3 template<typename T> void f4(int n, T* const &a) { ... }
```

- Zusätzlich kann die Übergabe *by value* auch mittels

```
1 template<typename T> void f1(int n, T a[]) { ... }
```

  geschehen, was den Charakter des Zeigers als Feld explizit macht. Eine Dimensionsangabe ist nicht erforderlich, da C++ Compiler die Grenzen von 1D Feldern nicht überprüfen.

- Besser: Verwendung von std::vector

Zeiger in den dynamischen Speicher müssen *by reference* übergeben werden, wenn

- der Speicher innerhalb der Funktion alloziert werden soll, z.B.

```
1  template<typename T>
2  void my_new(unsigned int n, T*& a) { a=new T[n]; }
3
4  template<typename T>
5  void my_delete(T*& a) { delete [] a; }
6
7  int main(int argc, char*[]) {
8    int *a; my_new(argc,a);
9    for (auto i=0;i<argc;i++) a[i]=i;
10   // do something useful
11   my_delete(a);
12   return 0;
13 }
```

- die Funktion modifizierende Zeigerarithmetik implementiert.

Zeiger können *by reference* oder *by value* übergeben werden, wenn dynamisch allozierter Speicher innerhalb der Funktion deallokiert werden soll.

Dynamische *n*D-Felder sind Zeiger auf dynamische 1D-Felder von Zeigern auf dynamische 1D-Felder von Zeigern ... Es ergeben sich eine Vielzahl von Optionen für die Parameterübergabe, z.B.

- *by value*

```
1 // template<typename T> void f(T *p[]) { ... } // bzw.
2 template<typename T> void f(T **p) { ... }
```

- *by reference*

```
1 template<typename T> void f(T **&p) { ... }
```

- *by* const *reference* (sinnvoll?)

```
1 template<typename T> void f(T **const &p) { ... }
```

- Besser: Verwendung von std::vector

AUFWACHEN!

Verstehen sie folgendes C++ Programm.

```
1  template<typename T>
2  void f(T*& p) { p=new T; }
3
4  int main(int argc, char*[]) {
5    using T=double;
6    if (argc==2) {
7      T ***p=new T**; f(*p); **p=new T[42];
8      // do something useful
9      // deallocate!
10   }
11   return 0;
12 }
```

Wie wird `p` korrekt dealloziert?

Wach?

```
1  template<typename T>
2  void f(T*& p) { p=new T; }
3
4  int main(int argc, char*[]) {
5    using T=double;
6    if (argc==2) {
7      T ***p=new T**; f(*p); **p=new T[42];
8      // do something useful
9      delete [] **p; delete *p; delete p;
10   }
11   return 0;
12 }
```

Outline

Visualsieren sie folgende Zeigerketten ...

- ... im statischen Speicher

```
int i=3; int *ip=&i; int** ipp=&ip;
```

- ... (teilweise) im dynamischen Speicher

```
int** ipp=new int*; *ipp=new int; **ipp=3;
```

→ Tafel

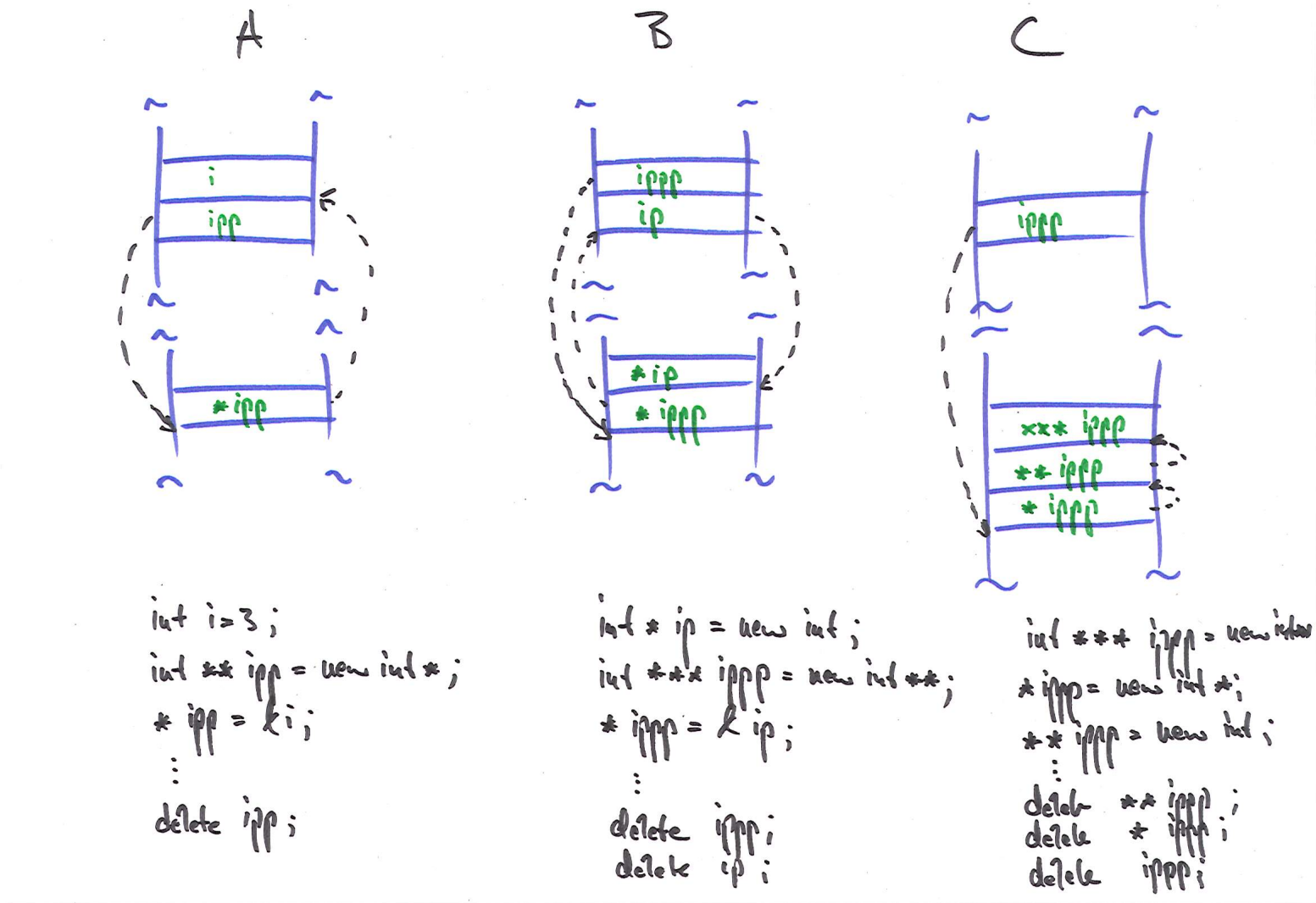
Implementieren sie

- benannten Zeiger auf unbenannten Zeiger auf benannten Speicher
- benannten Zeiger auf unbenannten Zeiger auf benannten Zeiger auf unbenannten Speicher
- benannten Zeiger auf unbenannten Zeiger auf unbenannten Zeiger auf unbenannten Speicher

Achten sie auf korrekte Freigabe des dynamischen Speichers.

Dynamischer Speicher und Zeigerketten

Noch Wach?



Zusammenfassung

Dynamische Speicherverwaltung

- **new** und **delete**
- Varianten
- Objekte nutzerdefinierten Typs

Dynamische 1D-Felder

- **new** und **delete** []
- Elementzugriff

Dynamische *n*D Felder

- **new** und **delete** []
- Elementzugriff
- Zeigerarithmetik

Dynamische Felder als Funktionsparameter

Zeigerketten