18. Dynamische Datenstrukturen I

Dynamischer Speicher, Adressen und Zeiger, Const-Zeiger, Arrays, Array-basierte Vektoren

Wiederholung: `vector<`T`>`

- Kann mit beliebiger Grösse `n` initialisiert werden
- Unterstützt diverse Operationen:

```
e = v[i];          // Get element
v[i] = e;          // Set element
l = v.size();      // Get size
v.push_front(e); // Prepend element
v.push_back(e);  // Append element
...
```

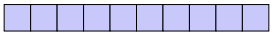
- Ein Vektor ist eine *dynamische Datenstruktur*, deren Grösse sich zur Laufzeit ändern kann

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<`T`>` (später, nur kurz angeschnitten)

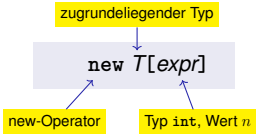
Vektoren im Speicher

Bereits bekannt: Ein Vektor belegt einen *zusammenhängenden* Speicherbereich

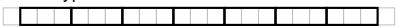


Frage: Wie *alloziert* (reserviert) man einen Speicherblock *beliebiger* Grösse zur Laufzeit, also *dynamisch*?

Der `new`-Ausdruck für Arrays

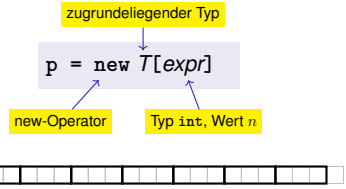


- **Effekt**: Neuer zusammenhängender Bereich im Speicher für n Elemente vom Typ *T* wird alloziert



- Dieser Speicherbereich wird *Array* (der Länge n) genannt

Der `new`-Ausdruck für Arrays



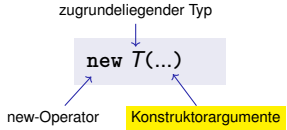
- **Typ**: Ein Zeiger $T*$ (mehr dazu gleich)
- **Wert**: Die Startadresse des Speicherbereichs

Ausblick: `new` und `delete`

```
new T[expr]
```

- Bisher: Speicher (lokale Variablen, Funktionsargumente) „lebt“ nur innerhalb eines Funktionsaufrufs
- Aber jetzt: Speicherblock im Vektor darf nicht vor dem Vektor selbst „sterben“
- Mittels `new` allozierter Speicher wird *nicht* automatisch dealloziert (= freigegeben)
- Zu jedem `new` gehört ein `delete`, das den Speicher explizit freigibt → **in zwei Wochen**

Der `new`-Ausdruck (ohne Arrays)



- **Effekt**: Speicher für ein neues Objekt vom Typ *T* wird alloziert . . .
- . . . und mit Hilfe des passenden Konstruktors initialisiert
- **Wert**: Adresse des neuen T-Objekt, **Typ**: Zeiger $T*$
- Auch hier gilt: Objekt „lebt“ bis es explizt gelöscht wird (Nutzen wird später klarer werden)

Zeiger-Typen

`T*` Zeiger-Typ für zugrunde liegenden Typ `T`

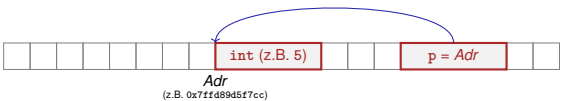
Ein Ausdruck vom Typ `T*` heisst *Zeiger (auf `T`)*

```
int* p; // Zeiger auf einen int
std::string* q; // Zeiger auf einen std::string
```


Zeiger-Typen

Wert eines Zeigers auf `T` ist die *Adresse* eines Objektes vom Typ `T`

```
int* p = ...;
std::cout << p; // z.B. 0x7ffd89d5f7cc
```



Adress-Operator

Frage: Wie kommt man an die Adresse eines Objekts?

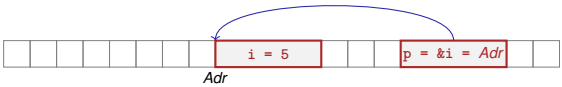
1. Direkt beim Erzeugen eines neuen Objekts mittels `new`
2. Für bestehende Objekte: Mittels des *Adress-Operators* `&`

`&expr` ← expr: L-Wert vom Typ T

- **Wert** des Ausdrucks: Die *Adresse* des Objekts (L-Wertes) *expr*
- **Typ** des Ausdrucks: Ein Zeiger $T*$ (vom Typ T)

Adress-Operator

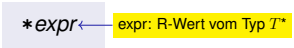
```
int i = 5; // i initialisiert mit 5
int* p = &i; // p initialisiert mit Adresse von i
```



Nächste Frage: Wie „folgt“ man einem Zeiger?

Dereferenz-Operator

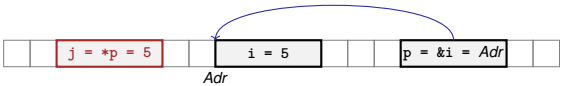
Antwort: Mittels des *Dereferenz-Operators* $*$



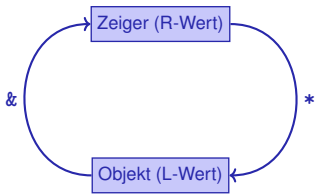
- **Wert** des Ausdrucks: Der *Wert* des Objekts an der durch *expr* bestimmten Adresse
- **Typ** des Ausdrucks: T

Dereferenz-Operator

```
int i = 5;
int* p = &i; // p = Adresse von i
int j = *p; // j = 5
```



Adress- und Dereferenzoperator



Zeiger-Typen

Wo ein $T*$ draufsteht, muss auch ein T drin sein

```
int* p = ...;  // p zeigt auf einen int
double* q = p; // aber q auf einen double → Kompilerfehler!
```

Eselsbrücke

Die Deklaration

```
T* p;       // p ist vom Typ „Zeiger auf T“
```

kann gelesen werden als

```
T *p;       // *p ist vom Typ T
```

Obwohl das legal ist, schreiben wir es nicht so!

Null-Zeiger

- Spezieller Zeigerwert, der angibt, dass (noch) auf kein Objekt gezeigt wird
- Repräsentiert durch das Literal `nullptr` (konvertierbar nach `T*`)

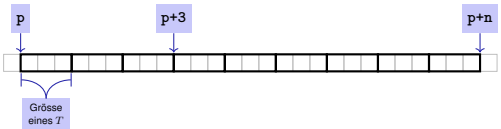
```
int* p = nullptr;
```

- Kann nicht dereferenziert werden (Laufzeitfehler)
- Dient der Vermeidung undefinierten Verhaltens

```
int* p; // p could point to anything
int* q = nullptr; // q explicitly points nowhere
```


Zeiger-Arithmetik: Zeiger plus `int`

```
T* p = new T[n]; // p points to first array element
```

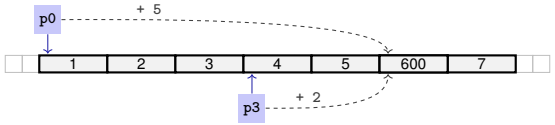


Wie zeigt man auf hintere Elemente? → *Zeiger-Arithmetik*:

- `p` gibt die *Adresse* des *ersten* Array-Elements, `*p` dessen *Wert*
- `*(p + i)` gibt den Wert des `i`-*ten* Array-Elements, für $0 \leq$ `i` $<$ `n`
- `*p` ist äquivalent zu `*(p + 0)`

Zeiger-Arithmetik: Zeiger plus `int`

```
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to 1st element
int* p3 = p0 + 3; // p3 points to 4th element
*(p3 + 2) = 600; // set value of 6th element to 600
std::cout << *(p0 + 5); // output 6th element's value (i.e. 600)
```



Zeiger-Arithmetik: Zeiger minus `int`

- Wenn *ptr* ein Zeiger auf das Element mit Index k in einem Array a der Länge n ist
- und der Wert von *expr* eine ganze Zahl i ist, $0 \leq k - i \leq n$,

dann liefert der Ausdruck

ptr - expr

einen Zeiger zum Element von a mit Index $k - i$.

Zeigersubtraktion

- Wenn *p1* und *p2* auf Elemente desselben Arrays `a` mit Länge n zeigen
- und $0 \leq k_1, k_2 \leq n$ die Indizes der Elemente sind, auf die *p1* und *p2* zeigen, so gilt

p1 - p2 hat den Wert k_1 - k_2

↑

Nur gültig, wenn *p1* und *p2* ins gleiche Array zeigen.

- Die Zeigerdifferenz beschreibt, „wie weit die Elemente voneinander entfernt sind“

Zeigeroperatoren

Beschreibung	Op	Stellig-keit	Prä-zedenz	Assoziativi-tät	Zuordnung
Subskript	`[]`	2	17	links	R-Werte → L-Wert
Dereferenzierung	`*`	1	16	rechts	R-Wert → L-Wert
Adresse	`&`	1	16	rechts	L-Wert → R-Wert

Präzedenzen und Assoziativitäten von +, −, ++ (etc.) wie in Kapitel 2

Zeigerwerte sind keine Ganzzahlen

- Adressen können als „Hausnummern des Speichers“, also als Zahlen interpretiert werden.
- Ganzzahl- und Zeigerarithmetik verhalten sich aber unterschiedlich.

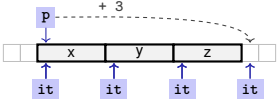
`ptr + 1` ist *nicht* die nächste Hausnummer, sondern die s-nächste, wobei s der Speicherbedarf eines Objekts des Typs ist, der `ptr` zugrundeliegt.

- Zeiger und Ganzzahlen sind nicht kompatibel:

```
int* ptr = 5; // Fehler: invalid conversion from int to int*
int a = ptr;  // Fehler: invalid conversion from int* to int
```

Sequenzielle Zeiger-Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```



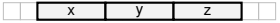

```
for (char* it = p;
     it != p + 3;
     ++it){

  std::cout << *it << ' ';
}
```

- `it` zeigt aufs erste Element
- Abbruch falls Ende erreicht
- Zeiger elementweise voranschieben
- Aktuelles Element ausgeben: 'x'

Wahlfreier Zugriff auf Arrays

```
char* p = new char[3]{'x', 'y', 'z'};
```



- Der Ausdruck `*(p + i)`
- kann auch geschrieben werden als `p[i]`
- z.B. `p[1] == *(p + 1) == 'y'`

Wahlfreier Zugriff auf Arrays

Iteration über ein Array mittels Indizes und *wahlfreiem Zugriff*:

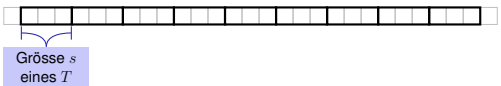
```
char* p = new char[3]{'x', 'y', 'z'};

for (int i = 0; i < 3; ++i)
  std::cout << p[i] << ' ';
```

Aber: Dies ist weniger *effizient* als der vorher gezeigte *sequenzielle* Zugriff mittels Zeiger-Iteration

Wahlfreier Zugriff auf Arrays

```
T* p = new T[n];
```



- Zugriff `p[i]`, also `*(p + i)`, „kostet“ Berechnung $p + i \cdot s$
- Iteration mittels *wahlfreiem Zugriff* (`p[0]`, `p[1]`, ...) kostet eine Addition und eine Multiplikation pro Zugriff
- Iteration mittels *sequentiellem Zugriff* (`++p`, `++p`, ...) kostet nur eine Addition pro Zugriff
- Sequenzieller Zugriff ist daher für Iterationen zu bevorzugen

Ein Buch lesen . . . mit wahlfreiem Zugriff . . . mit sequenziellem Zugriff

Wahlfreier Zugriff

- öffne Buch auf S.1
- klappe Buch zu
- öffne Buch auf S.2-3
- klappe Buch zu
- öffne Buch auf S.4-5
- klappe Buch zu
- ....

Sequenzieller Zugriff

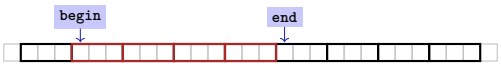
- öffne Buch auf S.1
- blättere um
- blättere um
- blättere um
- blättere um
- blättere um
- ...

Statische Arrays

- `int* p = new int[`$expr$`]` erzeugt ein dynamisches Array der Grösse $expr$
- C++ hat noch *statische* Arrays von der Vorgängersprache C geerbt: `int a[`$cexpr$`]`
- Statische Arrays haben unter anderem den Nachteil, dass ihre Grösse $cexpr$ eine Konstante sein muss. D.h. $cexpr$ kann z.B. `5` oder `4*3+2` sein, aber kein von der Tastatur eingelesener Wert `n`.
- Eine statisches Array-Variable `a` kann wie ein Zeiger verwendet werden
- Faustregel: Vektoren sind besser als dynamische Arrays, welche besser als statische Arrays sind

Arrays in Funktionen

Konvention in C++: Übergabe eines Arrays (oder eines Array-Ausschnitts) mit zwei Zeigern



- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element
- `[begin, end)` bezeichnet die Elemente des Array-Ausschnitts
- `[begin, end)` ist leer, wenn `begin == end`
- `[begin, end)` muss ein *gültiger Bereich* sein, d.h. ein echter (evtl. leerer) Array-Ausschnitt

Arrays in (mutierenden) Funktionen: `fill`

```
// PRE: [begin, end) ist ein gültiger Bereich
// POST: Jedes Element in [begin, end) wurde auf value gesetzt
void fill(int* begin, int* end, int value) {
  for (int* p = begin; p != end; ++p)
    *p = value;
}

...
int* p = new int[5];
fill(p, p+5, 1); // Array bei p wird zu {1, 1, 1, 1, 1}
```

Funktionen mit/ohne Effekt

- Zeiger können, wie auch Referenzen, für Funktionen mit Effekt verwendet werden. Beispiel: `fill`
- Aber viele Funktionen haben keinen Effekt, sie lesen Daten nur
- $\Rightarrow$ Verwendung von `const`
- Bisher, zum Beispiel:

```
const int zero = 0;
const int& nil = zero;
```


Positionierung von Const

Zur Determinierung der Zugehörigkeit des `const` Modifiers:

`const` T ist äquivalent zu T `const` (und kann auch so geschrieben werden):

```
const int zero = ...  ⟺  int const zero = ...
const int& nil = ...  ⟺  int const& nil = ...
```

Beide Schreibweisen werden in der Praxis genutzt

Const und Zeiger

Lies Deklaration von rechts nach links

`int const p;`	`p` ist eine konstante Ganzzahl
`int const* p;`	`p` ist ein Zeiger auf eine konstante Ganzzahl
`int* const p;`	`p` ist ein konstanter Zeiger auf eine Ganzzahl
`int const* const p;`	`p` ist ein konstanter Zeiger auf eine konstante Ganzzahl

Nicht-mutierende Funktionen: `print`

Es gibt auch *nicht* mutierende Funktionen, die nur lesend auf Elemente eines Arrays zugreifen

```
// PRE: [begin, end) ist ein gültiger Bereich
// POST: Die Werte in [begin, end) wurden ausgegeben
void print(
    int const* const begin,
    int const* const end) {

  for (int const* p = begin; p != end; ++p)
    std::cout << *p << ' ';
}
```

Const-Zeiger auf const-int

Ebenfalls (aber andere Schreibweise)

Zeiger, *nicht const*, auf const-int

Zeiger `p` kann selbst nicht `const` sein, da er verändert wird (`++p`)

`const` ist nicht absolut

- Der Wert an einer Adresse kann sich ändern, auch wenn ein `const`-Zeiger diese Adresse speichert.

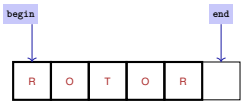
beispiel

```
int a[5];
const int* begin1 = a;
int*       begin2 = a;
*begin1 = 1;    // Fehler: *begin1 ist const
*begin2 = 1;    // ok, obwohl sich damit auch *begin1 ändert
```

- `const` ist ein Versprechen lediglich aus Sicht des `const`-Zeigers, keine absolute Garantie.

Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



Arrays, `new`, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse
- `new` T`[`n`]` alloziert ein T-Array der Grösse n
- T`* p = new` T`[`n`]`: Zeiger `p` zeigt auf das erste Array-Element
- Zeigerarithmetik ermöglicht Zugriff auf hintere Array-Elemente
- Sequenzielle Iteration über Arrays mittels Zeigern ist effizienter als wahlfreier Zugriff
- `new` T alloziert Speicher für (und initialisiert) ein einzelnes T-Objekt und liefert einen Zeiger darauf
- Zeiger können auf etwas (nicht) `const`es zeigen und selbst (nicht) `const` sein
- Mittels `new` allozierter Speicher wird *nicht* automatisch freigegeben (mehr dazu demnächst)
- Zeiger und Referenzen sind verwandt, beide „verweisen“ auf Objekte im Speicher. Siehe auch die Extrafolien `pointers.pdf`)

Array-basierter Vektor

- Vektoren ... da war doch was 🙄
- Nun wissen wir, wie man Speicherblöcke beliebiger Grösse allozieren kann ...
- ... und können einen Vektor, auf einem solchen Speicherblock aufbauend, implementieren
- `avec` – ein Array-basierter Vektor für `int`-Elemente

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<T>` (später, nur kurz angeschnitten)

Array-basierter Vektor `avec`: Klassensignatur

```
class avec {
  // Private (internal) state:
  int* elements; // Pointer to first element
  unsigned int count; // Number of elements

public: // Public interface:
  avec(unsigned int size); // Constructor
  unsigned int size() const; // Size of vector
  int& operator[](int i); // Access an element
  void print(std::ostream& sink) const; // Output elems.
}
```


Konstruktor `avec::avec()`

```
avec::avec(unsigned int size)
        : count(size) {

  elements = new int[size];
}
```

Grösse speichern

Speicher allozieren

Nebenbemerkung: Vektor wird nicht mit einem Standardwert initialisiert

Exkurs: Zugriff auf Membervariablen

```
avec::avec(unsigned int size): count(size) {
  this->elements = new int[size];
}
```

- `elements` ist eine Membervariable unserer `avec`-Instanz
- Diese Instanz ist mittels des *Zeigers* `this` zugreifbar
- `elements` ist eine Kurzschreibweise von `(*this).elements`
- Dereferenzieren eines Zeigers (`*this`) gefolgt von einem Memberzugriff (`.elements`) ist eine so häufig genutzte Operation, dass sie verkürzt geschrieben werden kann als `this->elements`
- Eselsbrücke: „Folge dem Zeiger zur Membervariablen“

Funktion `avec::size()`

```
int avec::size() const {
  return this->count;
}
```

Verändert den Vektor nicht

Grösse zurückgeben

Anwendungsbeispiel:

```
avec v = avec(7);
assert(v.size() == 7); // ok
```

Funktion `avec::operator[]`

```
int& avec::operator[](int i) {
  return this->elements[i];
}
```

i-tes Element zurückgeben

Elementzugriff mit Indexüberprüfung:

```
int& avec::at(int i) const {
  assert(0 <= i && i < this->count);

  return this->elements[i];
}
```


Funktion `avec::operator[]`

```
int& avec::operator[](int i) {
  return this->elements[i];
}
```

Anwendungsbeispiel:

```
avec v = avec(7);
std::cout << v[6]; // Outputs a "random" value
v[6] = 0;
std::cout << v[6]; // Outputs 0
```


Funktion `avec::operator[]` braucht's doppelt

```
int& avec::operator[](int i) { return elements[i]; }
const int& avec::operator[](int i) const { return elements[i]; }
```

- Die erste Memberfunktion ist *nicht const* und gibt eine *nicht-const*-Referenz zurück

```
avec v = ...; // A non-const vector
std::cout << v.get[0]; // Reading elements is allowed
v.get[0] = 123; // Modifying elements is allowed
```

- Sie wird auf nicht-const-Vektoren aufgerufen

Funktion `avec::operator[]` braucht's doppelt

```
int& avec::operator[](int i) { return elements[i]; }
const int& avec::operator[](int i) const { return elements[i]; }
```

- Die zweite Memberfunktion *ist const* und gibt eine *const*-Referenz zurück

```
const avec v = ...; // A const vector
std::cout << v.get[0]; // Reading elements is allowed
v.get[0] = 123; // Compiler error: modifications are not
    allowed
```

- Sie wird auf const-Vektoren aufgerufen

Siehe auch das diesem PDF angehängte Beispiel

Funktion `avec::print()`

Elemente mittels sequenziellem Zugriff ausgeben:

```
void avec::print(std::ostream& sink) const {
  for (int* p = this->elements;
       p != this->elements + this->count;
       ++p)
  {
    sink << *p << ' ';
  }
}
```

Zeiger auf erstes Element

Zeiger elementweise voranschieben

Abbruch falls hinter letztem Element

Aktuelles Element ausgeben

Funktion `avec::print()`

Abschluss: Ausgabeoperator überladen:

```
_____________ operator<<(_____________ sink,
                         _____________ vec) {
  vec.print(sink);
  return _____________;
}
```

```
std::ostream& operator<<(std::ostream& sink,
                         const avec& vec) {
  vec.print(sink);
  return sink;
}
```

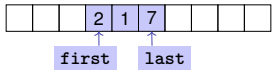

Weitere Funktionen

```
class avec {
  ...
  void push_front(int e)      // Prepend e to vector
  void push_back(int e)       // Append e to vector
  void remove(unsigned int i) // Cut out ith element
  ...
}
```

Gemeinsamkeit: Diese Operationen müssen die *Grösse* des Vektors verändern

Arrays vergrössern/verkleiner

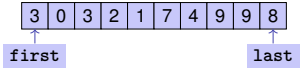
Ein allozierter Speicherblock (z.B. `new int[3]`) kann nicht nachträglich vergrössert/verkleinert werden



Möglichkeit:

- Mehr Speicher als initial nötig allozieren
- Befüllen aus der Mitte heraus, mittels Zeigern auf erstes und letztes Element

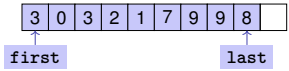
Arrays vergrössern/verkleiner



- Aber irgendwann sind alle Plätze belegt
- Dann nötig: Grösseren Speicherblock allozieren und Daten umkopieren

Arrays vergrössern/verkleiner

Elemente löschen erfordert verschieben (via kopieren) aller vorhergehenden oder nachfolgenden Elemente



Ähnlich: Einfügen an beliebiger Position