Programmeermethoden

Pointers

Walter Kosters en Jonathan Vis

week 10: 11–15 november 2024

www.liacs.leidenuniv.nl/~kosterswa/pm/

Een pointer is in feite gewoon een *geheugenadres*. Het geheugen kun je je voorstellen als een lineaire lijst met bytes.

Als `p` een pointer is naar een geheel getal `i` (gedefinieerd via `int * p = &i;`), is `p` het adres van `i`, en is `i` ook als `*p` te benaderen:

```
i = 196;   ⟺   *p = 196;
```

Met behulp van pointers kun je het geheugen dynamisch beheren, dat wil zeggen tijdens het runnen van het programma.

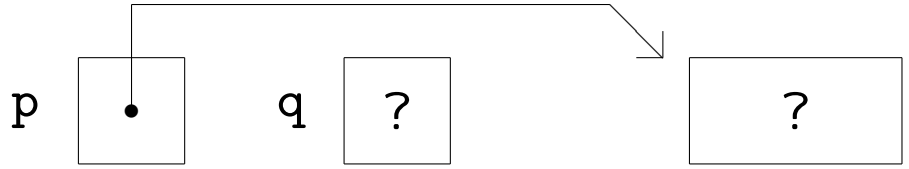
Stel je wilt jaargangen Donald Duck opbergen in een bibliotheek, in verschillende kasten, en maar één cataloguskaartje gebruiken.

Oplossing: plak de plek van de eerstvolgende jaargang achterin de vorige. Dus achterin 2007 zit een sticker met "jaargang 2008 staat op plek . . . ", achterin 2023 staat "dit is de laatste". En de eerste is 1952.

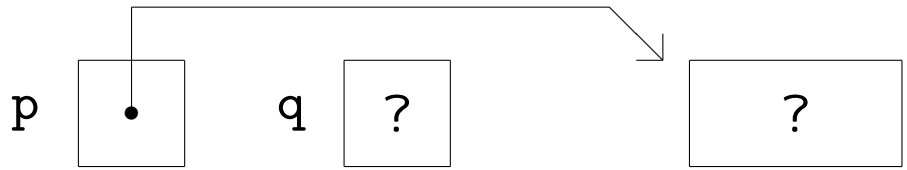
Voorbeeld — 1

⋈ `int * p;  int * q;` — twee pointers naar `int` (eger)

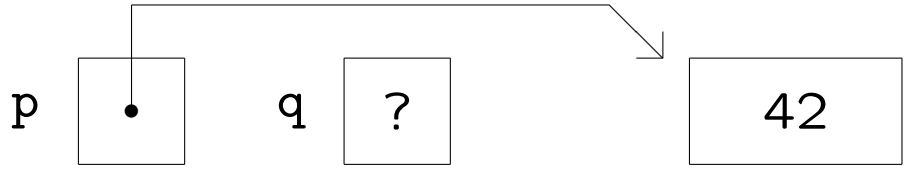
⋈ `p = new int;` — maak nieuwe `int`
en stop diens adres in `p`



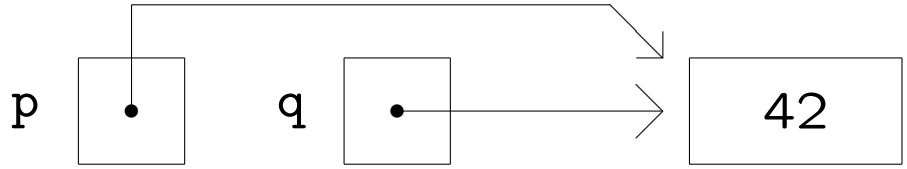
Voorbeeld — 2

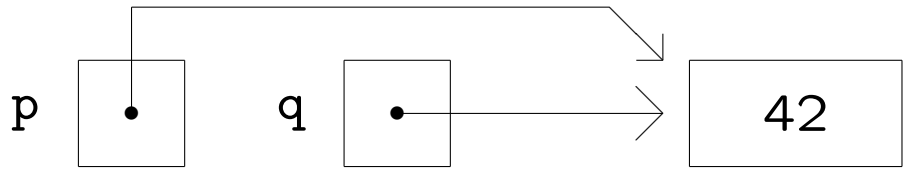


⋈ `*p = 42;` — stop 42 in die `int`

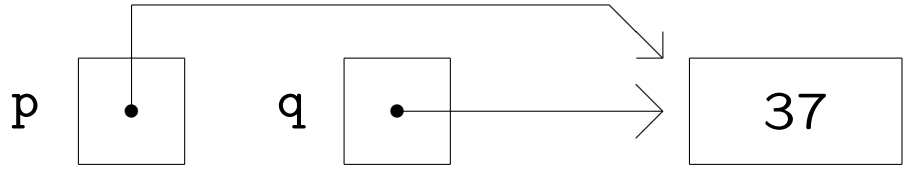


⋈ `q = p;` — laat `q` diezelfde `int` aanwijzen

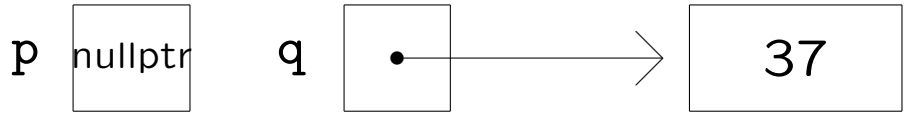




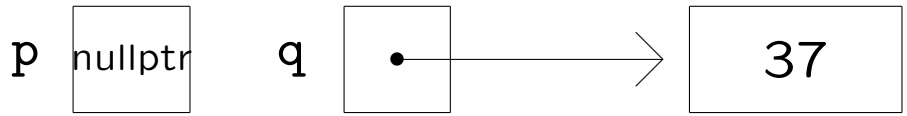
⋈ `*q = 37;` — verander de `int` via `q`



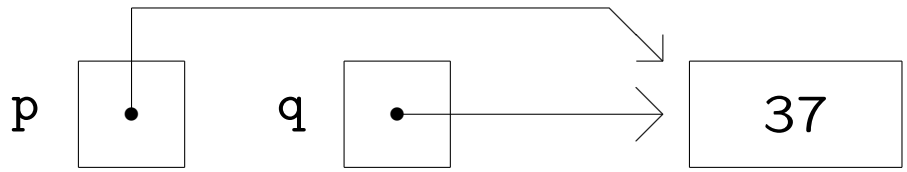
⋈ `p = nullptr;` — laat `p` naar "niets" wijzen



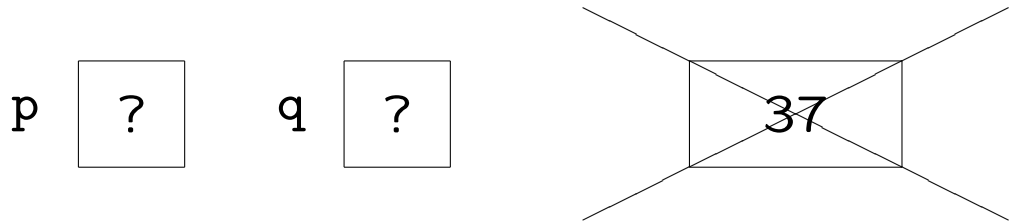
Voorbeeld — 4



⋈ `p = q;` — zet `p` weer "terug"



⋈ `delete q;` — blaas (via `q`) de `int` op



Bij het deleten wordt de geheugenruimte weer vrijgegeven. Bij `delete q;` wordt niet `q` vrijgegeven, maar datgene waar `q` naar wijst!

Voorlopig bevat `*q` nog wel de oude waarde (37) maar op enig moment niet meer . . .

Goed: `delete q; q = nullptr;`, want nu weet je zeker dat `q` naar "niets" wijst.

Sinds 2011 gebruiken we `nullptr` in plaats van `NULL` (dat is gewoon 0); doe zonodig `g++ -std=c++11 -Wall -Wextra ...`

Als je een pointer afloopt ("dereferencen") moet je zeker weten dat deze niet de `nullptr` is!

Fout is zoiets als

```
q = new int; q = p;
```

Je maakt nu namelijk eerst een nieuwe `int`, laat `q` daarnaar wijzen, en verandert `q` dan. Het adres van die eerste `int` is nu "voor altijd" zoek, en die `int` verspilt ruimte!

Slecht is:

```
p = nullptr; delete p;
```

Een "nullptr" kun je niet deleten — als je het toch doet, gebeurt er niks.

```
int i;        // een integer
int* p;       // een pointer naar een integer
p = &i;       // p wijst i aan: p is nu het adres van i
i = 12;       // verandert i
*p = *p + 8; // oftewel *p += 8;, verandert i opnieuw
int *q = &i; // q wijst ook i aan; geen new's,
              // en zeker geen delete's!
```

Let op: `int* p, q;` betekent dat `p` een pointer naar een integer is, en `q` een integer; bij `int* p; int* q;` en ook bij `int *p, *q;` heb je twee pointers naar een integer.

En wat betekent `*p++`? Is dat `(*p)++` of `*(p++)`?

En met klassen erbij:

```
class wagon {
  public:
    int hoogte;
    ...
};//wagon

wagon *p;  // p is pointer naar (= adres van) een wagon
```

Nu kun je de hoogte van de door `p` aangewezen wagon `*p` (de member-variabele `hoogte`) benaderen via `(*p).hoogte`, maar ook via (nieuwe notatie) `p->hoogte`.

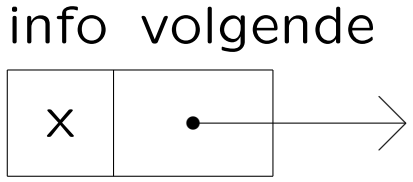
Enkelverbonden pointerlijst — 1

We maken nu een enkelverbonden pointerlijst bestaande uit vakjes:

```
class vakje {
  public:
    char info;
    vakje* volgende;
};//vakje
```

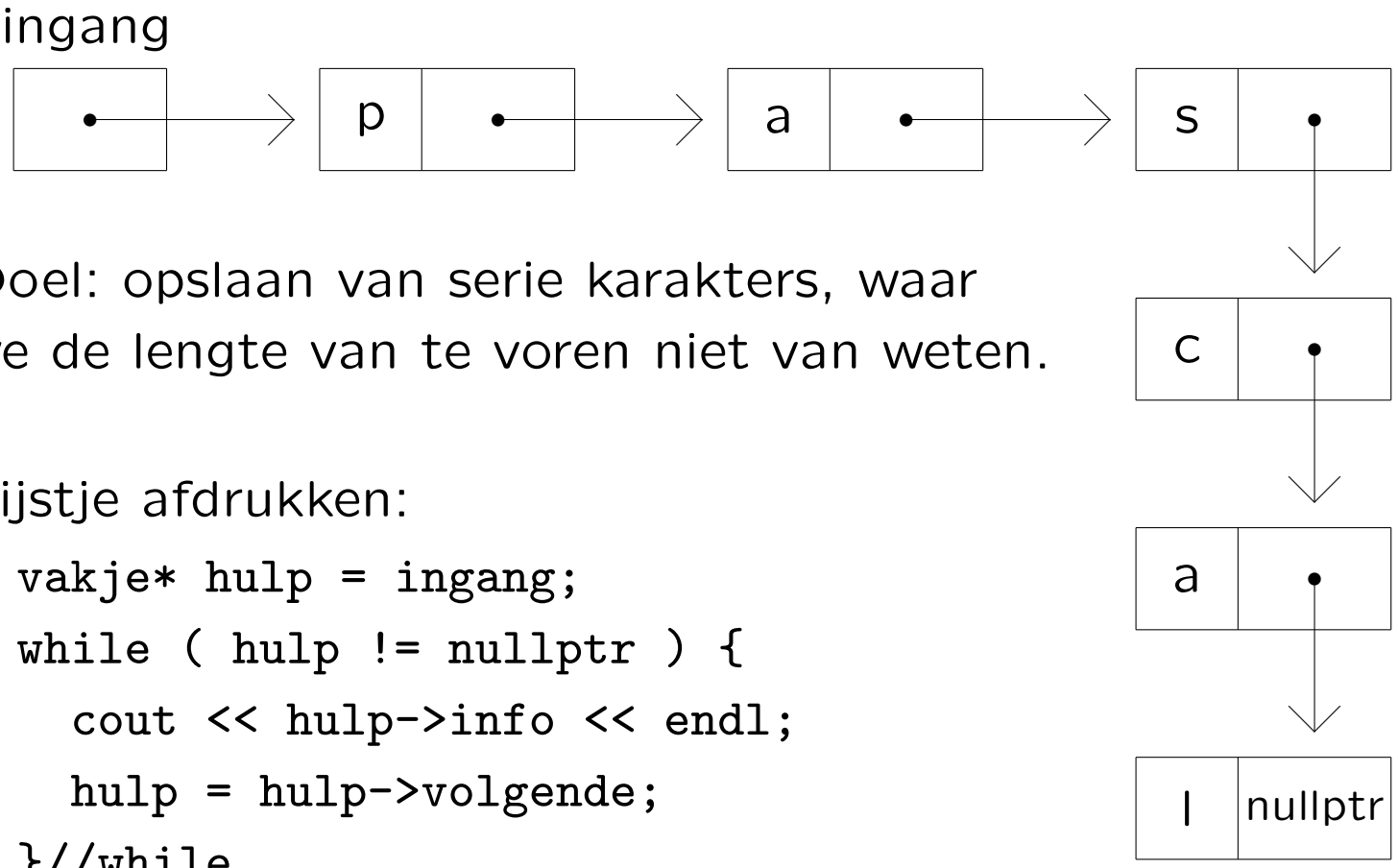
Vaak wordt hiervoor in plaats van `class` het iets eenvoudigere `struct` gebruikt.

Zo'n vakje ziet er uit als:



Enkelverbonden pointerlijst — 2

We willen bijvoorbeeld maken:



Doel: opslaan van serie karakters, waar we de lengte van te voren niet van weten.

Lijstje afdrukken:

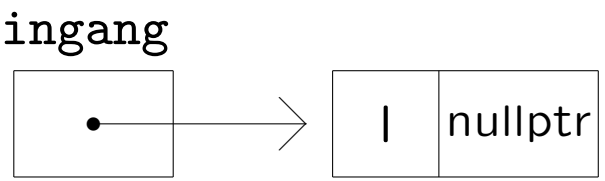
```
vakje* hulp = ingang;
while ( hulp != nullptr ) {
  cout << hulp->info << endl;
  hulp = hulp->volgende;
}//while
```

Enkelverbonden pointerlijst — 3

Hoe bouwen we zo'n lijst vanaf niets op?

```
vakje* ingang;
ingang = new vakje;
ingang->info = 'l';
ingang->volgende = nullptr;
```

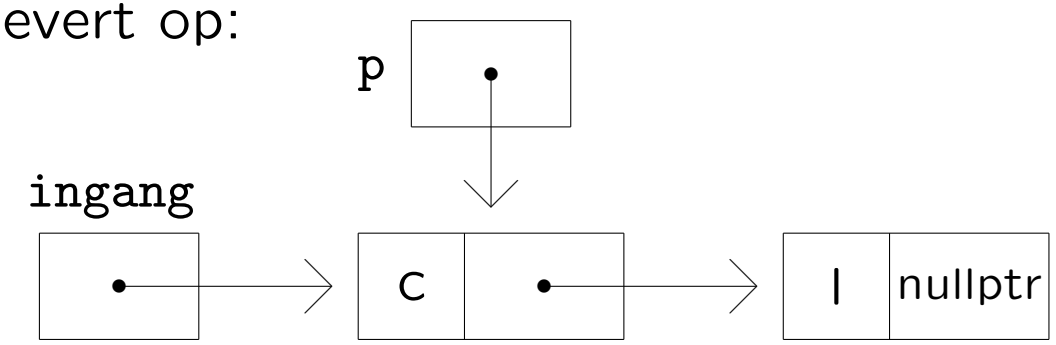
Dit levert op:



Hoe duwen we er nog een vakje voor?

```
vakje* p;
p = new vakje;
p->info = 'c';
p->volgende = ingang;
ingang = p;
```

Dit levert op:

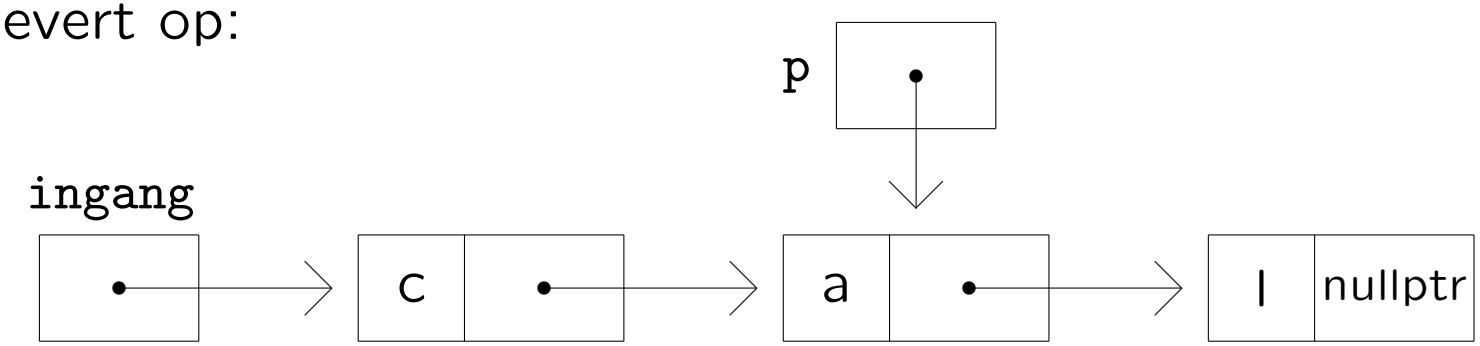


Let op: nu *niet* `delete p;` zeggen!

Hoe stoppen we er een vakje tussen?

```
vakje* p;
p = new vakje;
p->info = 'a';
p->volgende = ingang->volgende;
ingang->volgende = p;
```

Dit levert op:



Etcetera . . .

Maar het kan uiteraard beter met een functie:

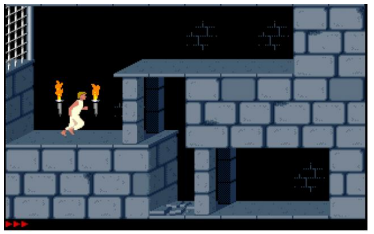
```
class vakje { public:
  char info;
  vakje* volgende;
};//vakje

void zetervoor (char letter, vakje* & ingang) {
  vakje* p;                          // let op de &
  p = new vakje;
  p->info = letter;
  p->volgende = ingang;
  ingang = p;  // en nu NIET delete p;!
}//zetervoor

ingang = nullptr; // met vakje* ingang;
zetervoor ('l',ingang); zetervoor ('a',ingang);
zetervoor ('c',ingang); zetervoor ('s',ingang);
zetervoor ('a',ingang); zetervoor ('p',ingang);
```

's Avonds lijst bewaren (zonder pointers):

```
vakje* hulp;
while ( ingang != nullptr ) {
  uitvoer.put (ingang->info);
  hulp = ingang;
  ingang = ingang->volgende;
  delete hulp;
}//while
```



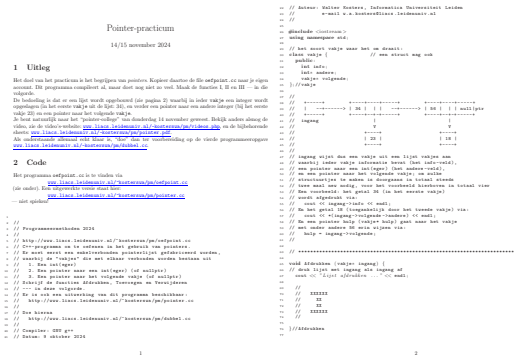
's Ochtends lijst weer opbouwen (met pointers):

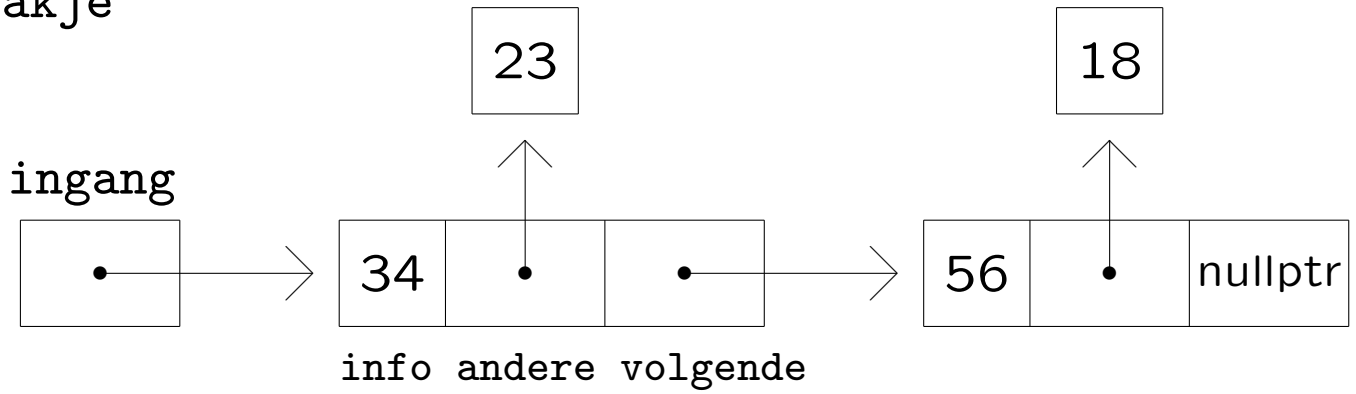
```
ingang = nullptr; char letter = invoer.get ( );
while ( ! invoer.eof ( ) ) {
  zetervoor (letter,ingang);
  letter = invoer.get ( );
}//while
```

Helaas . . . verkeerd om: nog eens wegschrijven en weer opbouwen, of `zeterachter` schrijven (laatste onthouden).

In het werkcollege (14–15 november 2024) gebruiken we:

```
class vakje {
  public:
    int info;
    vakje* volgende;
    int* andere;
};//vakje
```





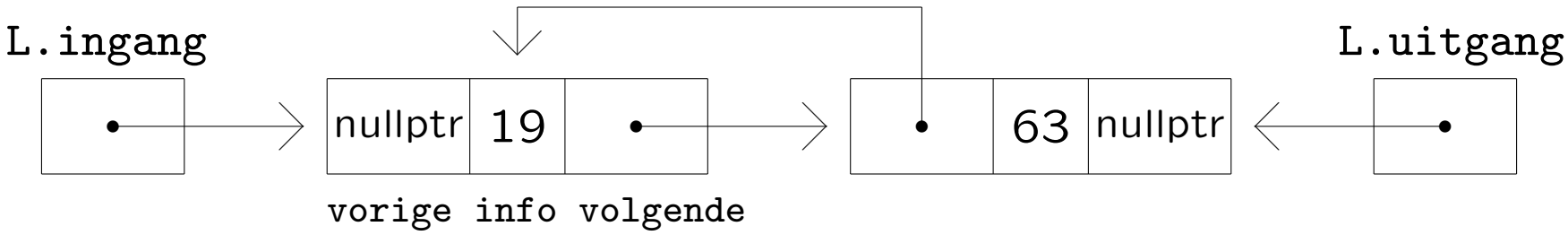
www.liacs.leidenuniv.nl/~kosterswa/pm/pmwc10.php

En voor een dubbel-verbonden pointerlijst:

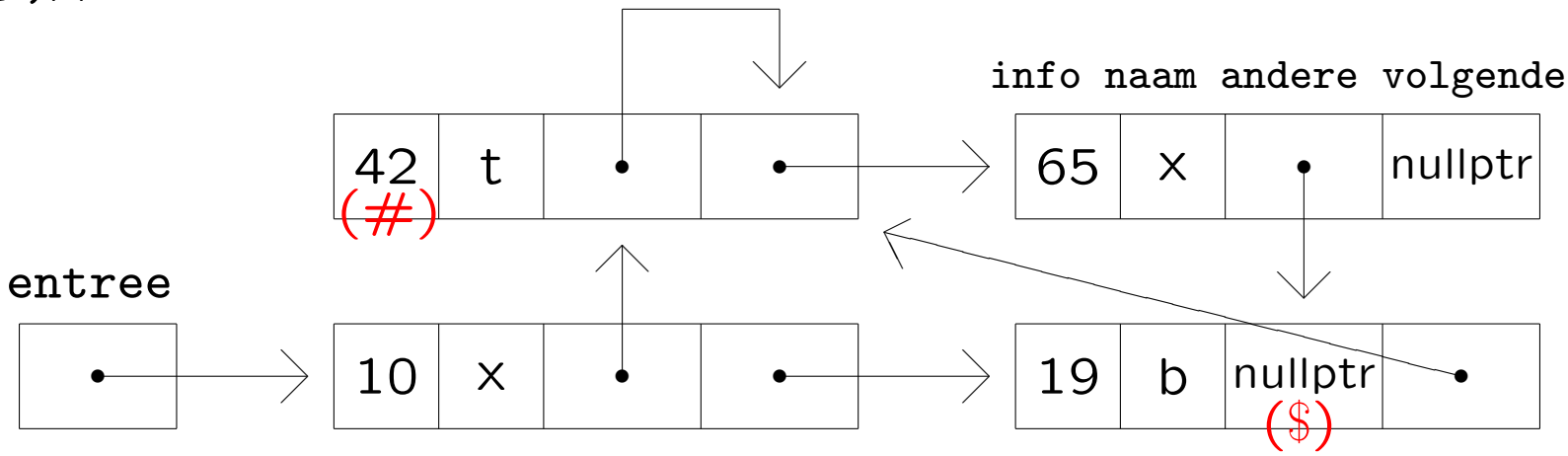
```
class element {
  public:
    element* vorige;
    int info;
    element* volgende;
};//element

lijst L;
```

```
class lijst {
  private:
    element* ingang;
    element* uitgang;
  public:
    void afdrukkenVA ( );
    ...
};//lijst
```



```
class vanalles { public:
  int info; char naam;
  vanalles* volgende; vanalles* andere;
};//vanalles
```



($) `entree->volgende->andere`

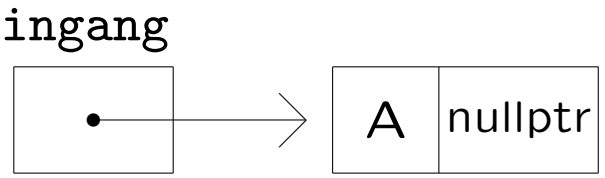
(#) `entree->volgende->volgende->info` of `entree->andere->info`

Controle op geheugenlekkages: `valgrind ./hetprogramma`

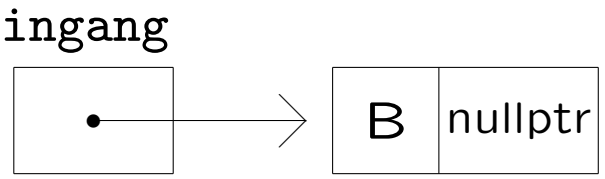
Bekijk de volgende functie:

```
void demo (char letter, vakje*   ingang) {
  if ( ingang != nullptr ) ingang->info = letter;
}//demo
```

We doen `ingang = nullptr; zetervoor ('A',ingang);`:



Daarna `demo ('B',ingang);` *met* en *zonder* `&` voor de parameter `ingang`. In beide gevallen krijgen we:



Maar nu de volgende functie:

```
void demoanders (vakje*   ingang) {
  ingang = nullptr;
}//demo
```

Met `&` zal de pointer `p` (als deze niet `nullptr` was) bij aanroep `demoanders (p);` veranderen, zonder `&` niet.

NB Dat wat er al dan niet veranderen kan is de *pointer*. Dat waar de pointer naar wijst kan altijd veranderen!

Pointers en arrays hebben veel met elkaar te maken.

Stel dat we hebben `int A[10];` en `int * p;`. Dan kun je het volgende doen:

```
p = A; // p wijst A[0] aan
p++;   // p wijst A[1] aan
p++;   // p wijst A[2] aan
cout << A[2] << " is gelijk aan " << *p << endl;
```

Dus `p` loopt het array langs, en `p++;` gaat naar het volgende array-element, waarbij de grootte van (in dit geval) `int`, `sizeof (int)` dus, gebruikt wordt als "stapgrootte".

Zoeken in pointerlijst

```
class vakje { public:
  char info; vakje* volgende; };//vakje

// Vindt eerste vakje met letter erin (uit lijst met
// ingang), als zo'n vakje bestaat; anders nullptr
vakje* vind (char letter, vakje* ingang) {
  vakje* hulp = ingang;  // NIET eerst hulp = new vakje;
  while ( hulp != nullptr )
    if ( letter == hulp->info )
      return hulp; // of met bool gevonden ...
    else
      hulp = hulp->volgende;
  return nullptr;  // en geen delete's!
}//vind
```

Zoeken in pointerlijst — recursief

```
// Vindt eerste vakje met letter erin (uit lijst met
// ingang), als zo'n vakje bestaat; anders nullptr
// nu recursief
vakje* vindrecursief (char letter, vakje* ingang) {
  if ( ingang == nullptr )
    return nullptr;
  else if ( ingang->info == letter )
    return ingang;
  else // komt letter voor in rest van de lijst?
    return vindrecursief (letter, ingang->volgende);
}//vindrecursief
```

Let op: de functie retourneert een pointer!

Opgave 58

Wat gebeurt er met en zonder &?

```
void tjatja (int* & r, int* & s) {
  r = new int; *r = 1;
  *s = 96;
}//tjatja
int main ( ) {
  int* p; int* q;
  p = new int; *p = 3;
  q = new int; *q = 4;
  cout << *p << *q << endl;
  tjatja (p,q);
  cout << *p << *q << endl;
  return 0;
}//main
```

Met &: 3 4 1 96

Zonder &: 3 4 3(!) 96

In C, dat alleen call by value heeft, moet je `wissel` als volgt schrijven (zie later):

```
void wissel (int *a, int *b) {
  int hulp = *a;  // *a is de int waar a naar wijst
  *a = *b;
  *b = hulp;
}//wissel
```

Voorbeeldaanroep, waarbij `&a` het adres van `a` betekent:
`a = 8; k = 2; wissel (&a,&k);`

NB De functie mag weer `wissel` heten, omdat de types van de parameters anders zijn; dit fenomeen heet overloading.

Merk op dat `a = b; b = a;` niet werkt! Blijkbaar heb je een hulpvariabele nodig.

Of toch niet:

```
void wisseltruc (int & a, int & b) {
  a = a + b;  // a = a_oud + b_oud
  b = a - b;  // b = a_oud
  a = a - b;  // a = b_oud
}//wisseltruc
```

De aanroep `wisseltruc (x,x)` maakt helaas `x` gelijk aan 0. En werkt niet voor bijvoorbeeld strings. En deze getructe functie mag overigens geen `wissel` heten.

Vierde programmeeropgave — 1

Programmeermethoden 2024
Vierde programmeeropgave: Koffiesweeper

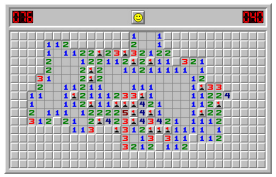
De *vierde* programmeeropgave van het vak **Programmeermethoden** in het najaar van 2024 heet *Koffiesweeper*; zie ook het **elfde werkcollege**, en lees geregeld deze pagina op WWW.

De opgave

Voor deze programmeeropgave gaan we het eenpersoons spel *Koffiesweeper* programmeren, beter bekend als *Minesweeper* (zie **minesweeper.online** en **Wikipedia**). Het is de bedoeling een klasse `koffiebord` te maken, die onder meer memberfuncties heeft als `drukaf`, `menszet` en `randomzet`. Uiteraard heeft deze klasse ook een constructor en een destructor. Verder moeten gedane zetten met behulp van een stapel ongedaan gemaakt kunnen worden.

Maak vanaf het begin gebruik van een aantal voorbeeldfiles, van waaruit de opgave stap voor stap kan worden gedaan. De files zijn:

- File met main: **hoofd.cc**.
- Headerfile met klassen (en `leesgetal`): **koffiebord.h**.
- Bijbehorende C++-file: **koffiebord.cc**.
- De bijpassende **makefile** (let op de TABs).



Het spel *Koffiesweeper*, een getrouwe kopie van het welbekende *Minesweeper*, verloopt als volgt. De speler ziet een rechthoek met *m* (hoogte) bij *n* (breedte) vakjes, alle gesloten. Op een aantal vakjes staat (verborgen) een kop koffie of thee. De speler krijgt te horen hoeveel dat er in totaal zijn. De speler kan een vakje selecteren door de coordinaten te geven. Als hier een kop koffie staat heeft de speler onmiddellijk verloren: de koffie moet meteen worden opgedronken. Zo niet, dan ziet de speler een getal tussen 0 en 8: het aantal directe buurvakjes, horizontaal, verticaal en diagonaal, dat een kop koffie bevat.

We spelen het spel als volgt. Eerst mag de grootte van het bord gekozen worden: het aantal rijen *m* en het aantal kolommen *n*, en welk percentage van de vakjes (ongeveer) gevuld is met een kop koffie (gebruik `random ( )` uit `<cstdlib>`; denk aan `srand ( )`; en gebruik de functie `leesgetal` van de derde opgave); het exacte aantal wordt aan de speler bekend gemaakt. De speler kan kiezen of hij/zij zelf één spelletje speelt, of dat er volledig random wordt gespeeld, waarbij het aantal spelletjes gekozen mag worden (steeds met een nieuwe beginconfiguratie). Het eerst geopende vakje bevat nooit een kop koffie.
Als de mens speelt wordt steeds de stand —in eenvoudig formaat— op het scherm getoond, en kan de speler zijn/haar zet doen (een vakje openen), of juist de laatste zet terugnemen (zie straks), of een random zet laten doen, of een vakje markeren met een `'K'`. Als er een reeds eerder geopende plek wordt geselecteerd, moet de speler natuurlijk opnieuw kiezen. Het aantal gedane zetten wordt ook steeds getoond.
De menselijke speler kan een vakje markeren waarvan hij/zij denkt dat het een kop koffie bevat; deze vorm van selecteren telt niet mee voor het aantal zetten. Als een vakje met 0 koffie-buren wordt geopend, kunnen al diens buren veilig worden opgevraagd. Schrijf hiertoe een *recursieve* functie die dit automatisch doet. De speler wint als alle vakjes die geen kop koffie bevatten zijn geopend.

Als het programma volledig random speelt, wordt in een tweetal arrays, één voor de gewonnen en één voor de verloren spelletjes, bijgehouden hoeveel zetten het telkens duurde. Na afloop wordt dan geprint hoeveel spelletjes *z* zetten duurden (*z* = 0, 1, ...), ten behoeve van een grafiek, zie onder.

Schrijf een functie voor de klasse `koffiebord` die een *pointerstructuur* aanlegt, waarbij ieder vakje, naast bijvoorbeeld een `int` en enkele `bool`'s als inhoud, tevens een array met 8 pointers naar de onmiddellijke buren heeft: middenboven (0), rechtsboven (1), rechts (2), rechtsonder (3), middenonder (4), linksonder (5), links (6) en linksboven (7). De vakjes aan de randen bevatten uiteraard diverse `nullptr`'s. Het bord is dus *niet* een *m* bij *n* array, maar een zeer ingewikkelde pointerstructuur.

Bij de menselijke speler moeten alle complete borden op een *stapel* worden bijgehouden, en deze kunnen daarmee teruggenomen worden. Zodra een speler zet, wordt een kopie van het bord opgeslagen. Dit onderdeel is zeker niet eenvoudig; mocht het ontbreken, dan kost dat een punt.

Het is de bedoeling om een vijftal files te produceren: de eerste bevat `main` en het menutje, de tweede (zeg `koffiebord.h`, zie boven) bevat de klasse-definitie voor `koffiebord`, en de derde (zeg `koffiebord.cc`, zie boven) bevat de functies uit die klasse. Evenzo zijn er files `stapel.h` en `stapel.cc`, indien van toepassing. Maak als het kan ook een `makefile`.
Code::Blocks-gebruikers: doe deze opgave liever op een Linux-machine. Maar het kan wel: open een nieuw project via "File -- New project -- Empty project", vul wat in, en voeg de drie files toe via "Project -- Add files", en daarna het project compileren op de gebruikelijke manier. Of, op eigen risico, lees **over projecten**.

Opmerkingen
Gebruik geschikte (member)functies. Bij deze opgave mogen wederom bij elke functie tussen begin-{ en eind-} *hooguit circa 30* niet al te volle of complexe regels staan! Elke functie dient van commentaar voorzien te zijn. Als uitzondering mag `main` langer zijn, als daarin met het menu gewerkt wordt. Let op goed parametergebruik: alle parameters, met uitzondering van membervariabelen, in de heading doorgeven, en de variabele-declaraties zowel bij `main` als bij de andere functies aan het begin. De enige te gebruiken headerfile is in principe `iostream`, en eventueel `ctime` en `cstdlib` (voor de random-generator). Zeer ruwe indicatie voor de lengte van de gezamenlijke C++-files: 600 regels. Denk aan het infoblokje.

Uiterste inleverdatum: **maandag 9 december 2024, 18:00 uur**.
Manier van inleveren:

1. Digitaal als team de C++-code **inleveren** via Brightspace > Course Tools > Assignments. Stuur geen executable's, LaTeX-files of PDF-files, maar lever alleen de vijf (of drie) C++-files en de `makefile` digitaal in! Noem de file `koffiebord.cc` hier bij voorkeur zoiets als `garfunkelsimongobord4.cc`, dit voor de opdracht van het duo Simon-Garfunkel, en analoog de andere files. De laatst voor de deadline ingeleverde versie wordt nagekeken.
2. En *ook* een papieren versie van het verslag (inclusief de C++-code van alle files, en de `makefile`) deponeren in de speciaal daarvoor bestemde doos "Programmeermethoden" bij kamer Gorlaeus BM.2.07.
   Overal duidelijk datum en namen van de (maximaal twee) makers vermelden, in het bijzonder als commentaar in de eerste regels van de C++-code.
   Het *verslag* (uiteraard weer in LaTeX, zie de eerdere opgaven) moet het volgende bevatten: een zeer korte beschrijving van het programma, een beschrijving van punten waarop het programma faalt (indien van toepassing), en een tabel met gewerkte uren, uitgesplitst per week en per persoon. En een referentie betreffende Minesweeper. En een grafiek (zie het **bijbehorende werkcollege** voor tips) waarin staat hoe lang een winnend of verliezend spelletje voor de random speler duurt — voor verschillende beginconfiguraties, met verwijzing en onderschrift.

Te gebruiken compiler: als hij maar C++ vertaalt; het programma moet in principe op een Linux-machine (met g++) draaien. Normering: layout 1; grafiek 1; verslag 1; commentaar 1; modulariteit (OOP, functies) 2; werking 4. Eventuele aanvullingen en verbeteringen: lees de huidige WWW-bladzijde:
`www.liacs.leidenuniv.nl/~kosterswa/pm/op4pm.php`.

www.liacs.leidenuniv.nl/~kosterswa/pm/op4pm.php

Vierde programmeeropgave — 2

Koffiesweeper programmeren we als volgt:

- week 1 (“10”): pointerpracticum, opgave lezen
- week 2 (“11”): klassen, pointerbord, meerdere files, ruw spelen
- week 3 (“12”): spel helemaal in orde maken, stapel
- week 4 (“13”): recursie, experiment (`gnuplot`), verslag

www.liacs.leidenuniv.nl/~kosterswa/pm/op4pm.php

- lees de vierde programmeeropgave, denk na over de klassen; de deadline is op maandag 9 december 2024
- lees Savitch Hoofdstuk 10

- lees dictaat Hoofdstuk 3.12

- maak opgaven 44/46, 52/56 uit het opgavendictaat

- doe het pointerpracticum: werkcollege 10 !

- `www.liacs.leidenuniv.nl/~kosterswa/pm/`