FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

HOOFDSTUK 2

NIEUWE FEATURES

SINDS C++11 (DEEL 1)

UNIVERSITEIT
GENT

Helga Naessens

Inhoud

- **functies als parameters**
- lambda functies
- `nullptr`
- smart pointers: `unique_ptr` en `shared_ptr`

Functies als parameters

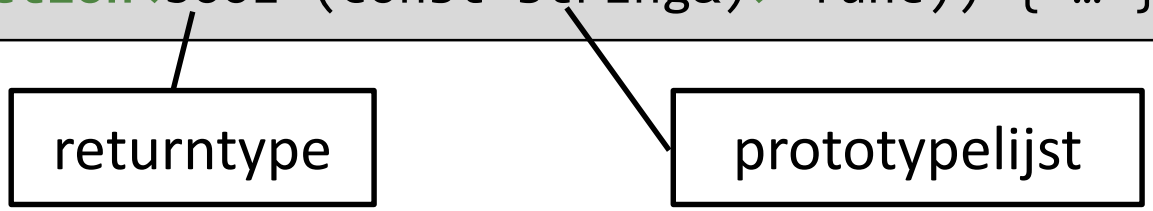
- (enkel) in C: met behulp van functie-pointers
- vóór C++11: met behulp van `template` => niet meer gebruiken!!!

```
template<typename Func>
bool zoek(const string t[], int n, Func func) { … }
```

- vanaf C++11: met behulp van `std::function`

```
#include <functional>
using namespace std;

bool zoek(const string t[], int n,
          function<bool (const string&)> func)) { … }
```



- Voorbeeld:

vóór C++11

```
template<typename Func>
bool zoek(const string t[], int n, Func func) {
   for (int i=0; i<n; i++)
      if (func(t[i]))
         return true;
   return false;
}
```

sinds C++11

```
bool zoek(const string t[], int n,
          function<bool (const string&)> func ) {
   … //zie hierboven
}
```

Zo is veel duidelijker welke signatuur de mee te geven functie moet hebben!!

```
bool geen_hoofdletters(const string& s) {
    …
}

int main() {
    string tab[10];
    … //tab opvullen
    if (zoek(tab, 10, geen_hoofdletters))
        cout << "string zonder hoofdletters gevonden ";
}
```

Vóór C++11: functie moest reeds ergens gedefinieerd zijn

Sinds C++11: er kunnen ook lambda functies gebruikt worden

Inhoud

- functies als parameters
- **lambda functies**
- `nullptr`
- smart pointers: `unique_ptr` en `shared_ptr`

lambda functies

- anonieme functies die toelaten om een functie lokaal (op de plaats van de oproep) te definiëren
- Definitie:

```
[ captures ] (parameters) -> returntype { statements; }
```

  - [*captures*] : [ ] duiden aan dat het om een lambda functie gaat; *captures* zijn optioneel (zie verder);
  - -> returntype: hoeft niet gespecifieerd te worden als het kan afgeleid worden uit de statements

- Voorbeeld: **vb_lambda1.cpp**

```
if (zoek(mail,4,[](const string& addr)
                {return addr.find(".be")!=string::npos;}))
```

capture-list

vb_lambda2.cpp

- bevat 0 of meer door een komma gescheiden captures
- Overzicht mogelijke captures:
  - `[]` captures nothing : de lambda functie kent enkel variabelen die meegegeven worden als parameter
  - `[a,&b]` a is captured by value, b is captured by reference.
  - `[&]` captures all variables in the body of the lambda by reference
  - `[&,b]` captures all variables in the body of the lambda by reference, but b is captured by value (⇒ b is read-only. Vb: ~~`b++;`~~)
  - `[=]` captures all variables in the body of the lambda by value
  - `[=, &b]` captures all variables in the body of the lambda by value, but b is captured by reference

Inhoud

- functies als parameters
- lambda functies
- **`nullptr`**
- smart pointers: `unique_ptr` en `shared_ptr`

nullptr

- sleutelwoord dat null-pointer constante voorstelt
- vervangt NULL en 0 en is sterk getypeerd
- voorbeeld

```
void proc(int);   //A
void proc(int *); //B
```

welke procedure zal opgeroepen worden?

```
proc(0);        //A
proc(NULL);     //compileerfout
proc(nullptr);  //B
```

vb_nullptr.cpp

Inhoud

- functies als parameters
- lambda functies
- `nullptr`
- **smart pointers: `unique_ptr` en `shared_ptr`**

Smart pointers: algemeen

- Smart pointer:
  - pointer wrapper klasse die naast de pointer zelf

    ook extra eigenschappen aanbiedt zoals

    automatisch vrijgeven van geheugen
  - Iets minder efficiënt dan traditionele pointers,

    maar géén `new` en `delete` meer nodig
- C++98: één smart pointer klasse (`auto_ptr`)
- C++11: twee smart pointer klassen: (`auto_ptr` is nu deprecated)
  - `unique_ptr`
  - `shared_ptr`

```
#include <memory>
using namespace std;
```

unique_ptr

uniquep.cpp

- slechts 1 eigenaar van de pointer mogelijk
- kan niet gekopieerd worden, mag wel moved of swapped worden

```
unique_ptr<int> p1 = make_unique<int>(); //sinds C++14
unique_ptr<int> p2 = make_unique<int>(202);
*p1 = 101; cout << *p1;
p2 = p1;   compilatie-fout: p1 mag niet gekopieerd worden!!
p1.swap(p2);   // wisselt pointers
p2 = move(p1); // transfereert eigenaar
p2.reset();    // geeft geheugen vrij
  // gebeurt automatisch bij out of scope
```

Oefening

Gegeven 2 *unieke pointers* `p1` en `p2` naar elementen van een zelfde onbepaald type.

Schrijf een procedure `kleinste_eerst(p1,p2)` die `p1` laat wijzen naar het kleinste element en `p2` naar het grootste element van de twee.

Je mag ervan uitgaan dat het type de <-operator ondersteunt.

shared_ptr

sharedp.cpp

- kan door meerdere eigenaars gedeeld worden
- geheugen waarnaar verwezen wordt, wordt slechts vrijgegeven als alle instanties van die `shared_ptr` vernietigd zijn

```
shared_ptr<int> p1, p2;
p1 = make_shared<int>(101); // sinds C++14
p2 = p1; // beide zijn nu eigenaar
p2.reset(); // geeft geheugen nog niet vrij (wegens p1)
p1.reset(); // geeft geheugen vrij (gebeurt automatisch
            // bij out of scope van alle eigenaars)
```

- gebruik `shared_ptr` enkel indien nodig!!
  (aantal referenties bijhouden zorgt voor overhead)

Inhoud

- functies als parameters
- lambda functies
- `nullptr`
- smart pointers: `unique_ptr` en `shared_ptr`