1 Leesvoer

In deze workshop gaan we de volgende concepten bestuderen:

1. linken,
2. headerbestanden,
3. bronbestanden,
4. de stack,
5. en heap.

Linken en het gerelateerde concept *linkage* wordt uitgelegd in hoofdstuk 15 van "The C++ Programming Language". Als je de voorbeeldcode niet snapt, is dat geen probleem. Concentreer je vooral op de concepten van linken, header en source files. Lees dit hoofdstuk **met uitzondering van** de volgende paragrafen:

1. 15.2.4 t/m 15.2.6
2. 15.4.2

2 Linke(r) boel

Onder vind je twee C++ bronbestanden. `file1.cpp` bevat een declaratie van de functie `hi`. `file2.cpp` bevat de definitie (implementatie) van `hi`. Beide bestanden kunnen we probleemloos compileren.

file1.cpp:

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

void hi();

int main(int argc, char **argv)
{
    hi();
    return 0;
}
```

file2.cpp:

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

void hi()
{
    cout << "Hello, world!" << endl;
}
```

Maak volgende opdrachten:

1. Gebruik de woorden “compiler” en “linker” om uit te leggen hoe je een werkend programma maakt van `file1.cpp` en `file2.cpp`.
2. Verwacht je een fout tijdens het compileren of linken wanneer we de declaratie op regel 4 van `file1.cpp` verwijderen? Leg uit.
3. Wat gebeurt er als we de definitie van `hi` uit `file2.cpp` verwijderen? Krijgen we dan een compilatie- of linkfout?
4. Plaats de declaratie van `hi` in een header bestand. Voeg een regel toe aan `file1.cpp` om deze header te gebruiken.

3 Volgorde

Bestudeer volgende code

volgorde.cpp:

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

int main(int argc, char **argv)
{
    hi();
    return 0;
}

void hi()
{
    cout << "Hello, world!" << endl;
}
```

Maak volgende opdrachten:

1. Compileer bovenstaande code.
2. Krijg je foutmeldingen? Zo ja, welke?
3. Wat betekenen deze foutmeldingen?
4. Pas het bestand aan zodat het wel te compileren is.

4 Declarations to the rescue!

Kijk eens naar volgend voorbeeld. Je hoeft de code nog niet volledig te begrijpen. Verderop wordt de werking uitgelegd.

forward_needed.cpp

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

```
void say_odd(int n)
{
    cout << n << " is odd." << endl;
    if (n > 0) {
        say_even(n - 1);
    }
}

void say_even(int n)
{
    cout << n << " is even." << endl;
    if (n > 0) {
        say_odd(n - 1);
    }
}

void say(int n)
{
    if (n < 0) {
        return;
    }

    if ((n % 2) == 0) {
        say_even(n);
    }
    else {
        say_odd(n);
    }
}

int main(int argc, char **argv)
{
    say(3);
}
```

In het volgende voorbeeld, wordt de functie `say` aangeroepen. Deze functie doet niets voor negatieve getallen. Voor een positief getal roept hij `say_odd` of `say_even` aan opdat `say(3)` volgende uitvoer geeft:

```
3 is odd
2 is even
1 is odd
0 is even
```

Om dit te laten werken, roept `say_odd` de functie `say_even` aan en visa versa.

Maak volgende opdrachten:

1. Welke foutmelding(en) krijg je als je deze code wilt compileren?
2. Wat betekent/betekenen deze?

Deze foutmelding ontstaat omdat `say_odd` de functie `say_even` wilt aanroepen voordat deze is gedeclareerd. Een C++-compiler kijkt niet vooruit om te zien dat de functie `say_even` later wel wordt gedefinieerd.

Maak volgende opdracht:

3. Leg uit waarom we in tegenstelling tot het vorige voorbeeld we dit probleem niet kunnen oplossen door de functies `say_odd` en `say_even` om te draaien.
4. Zorg dat de code wel compileert door op de juiste plaats de functie `say_even` te declareren.

5 Geheugenbeheer in C++ begrijpen

In het hoorcollege heb je geleerd dat lokale variabelen op de *stack* worden geplaatst. Daarnaast heb je gezien dat geheugen dat je met `new` aanmaakt op *heap* wordt geplaatst.

In deze opdracht gaan we onderzoeken hoeveel data je op de stack en heap mag plaatsen. Hiervoor gebruiken we onderstaand programma:

heapstack.cpp:

```cpp
#include <cstddef> // for std::size_t
#include <memory> // for std::uninitialized_fill

static const std::size_t stack_size = 1024 * 1024; // 1 MB!
static const std::size_t heap_size = 1024 * 1024 * 1024; // 1 gigabyte

void fill_stack() {
    char stack_object[stack_size];
    std::uninitialized_fill_n(stack_object, stack_size, 'a');
}

void fill_heap() {
    char *heap_object = new char[heap_size];
    std::uninitialized_fill_n(heap_object, heap_size, 0);
    delete[] heap_object;
}

int main() {
    fill_stack();
    return 0;
}
```

Op regel vier en vijf worden twee constanten gedefinieerd. Deze bepalen de grootte van de objecten (in bytes) die we op de stack en heap gaan aanmaken.

De functie `fill_stack` maakt een lokale array met een omvang van `stack_size` bytes. Omdat de array een lokale variabele is, komt de hele array op de stack te staan. De functie `uninitialized_fill_n` zorgt ervoor dat elke byte ook daadwerkelijk door je programma gebruikt wordt.

De functie `fill_heap` maakt een array aan op de heap. Dit kun je zien aan het sleutelwoord `new`. De grootte van dit object wordt bepaald door de constante `heap_size`.

In de functie `main` wordt één van deze twee functies aangeroepen.

Maak nu volgende opdrachten:

1. Pas de waarde van `stack_size` aan tot je programma crasht. Hoeveel geheugen mag je van jouw C++-implementatie op de stack gebruiken? **Let op!** *de grootte van de stack kan variëren per besturingssysteem en (compiler)*
2. Laat `main` de functie `fill_heap` aanroepen. Bij welke waarde van `heap_size` crasht je programma?
3. Kun je op basis van de vorige twee opdrachten een conclusie trekken over het verschil in grootte van de heap en stack?