

Inleiding Programmeren

Pointers, Dynamic memory

Tom Hofkens

Wat zagen we vorige week?

Call by Value

- Default bij functie aanroepen in C++
 - Parameter = variabele in de functie
 - Parameter wordt geïnitieerd met een kopij van het argument

Call by Reference

- Voeg & toe voor de parameter naam
 - Parameter = alias voor het argument
 - Argument moet een lvalue (= object geassocieerd met een geheugenplaats) zijn!

Wat zagen we vorige week?

```
void deling(int deelta1, int deler, int& quotient, int& rest)
    quotient = deelta1 / deler;
    rest = deelta1 % deler;
}
```

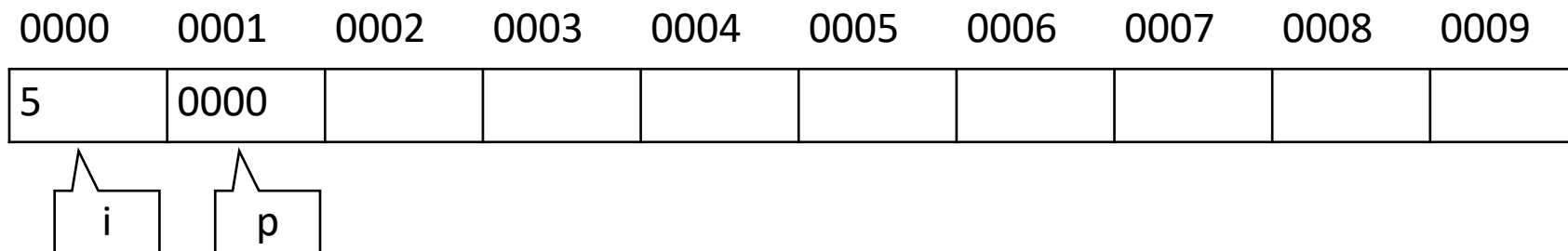
```
int main() {
    int dt = 100;
    int dr = 13;
    int q;
    int r;
    deling(dt, dr, q, r);
    cout << q << " " << r << endl;
    return 0;
}
```



Wat zagen we vorige keer?

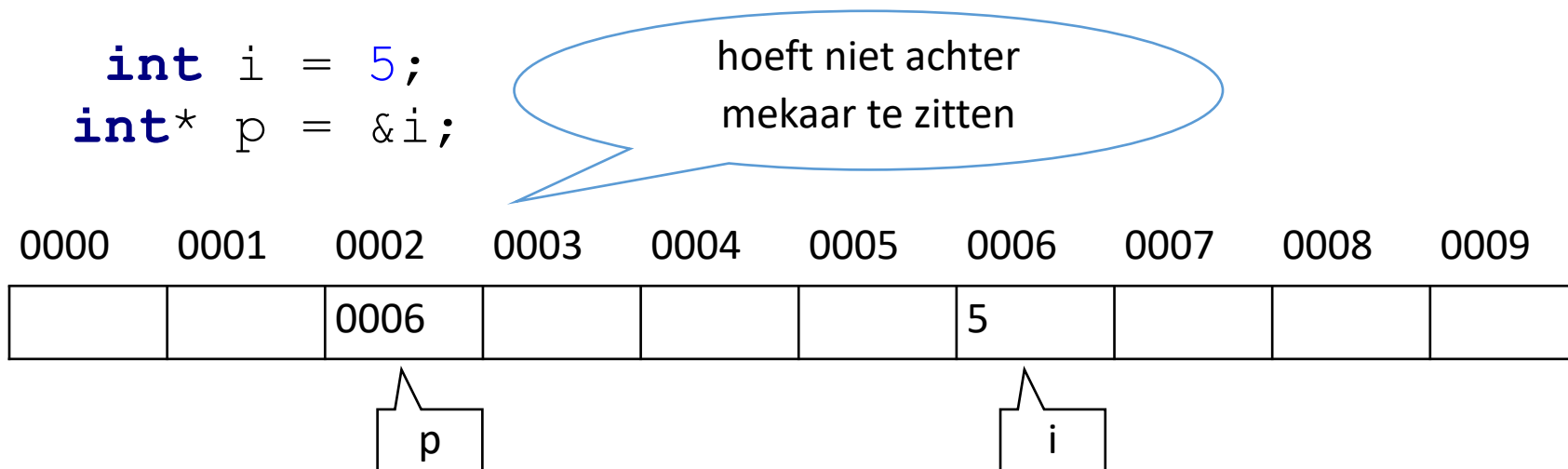
- Variabele = naam van een geheugenplaats
- Pointer = verwijzing naar een plaats in het geheugen
 - Pointer variabele = naam van een plaats in het geheugen die een geheugenadres bevat
 - Pointers in C++ zijn getypeerd

```
int i = 5;  
int* p = &i;
```



Wat zagen we vorige keer?

- Variabele = naam van een geheugenplaats
- Pointer = verwijzing naar een plaats in het geheugen
 - Pointer variabele = naam van een plaats in het geheugen die een geheugenadres bevat
 - Pointers in C++ zijn getypeerd



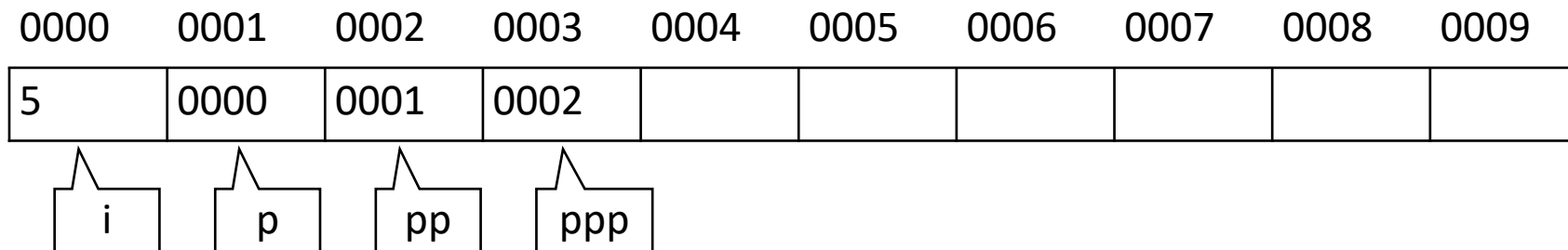
Wat zagen we vorige keer?

- We kunnen ook een pointer naar een pointer naar een integer maken

```

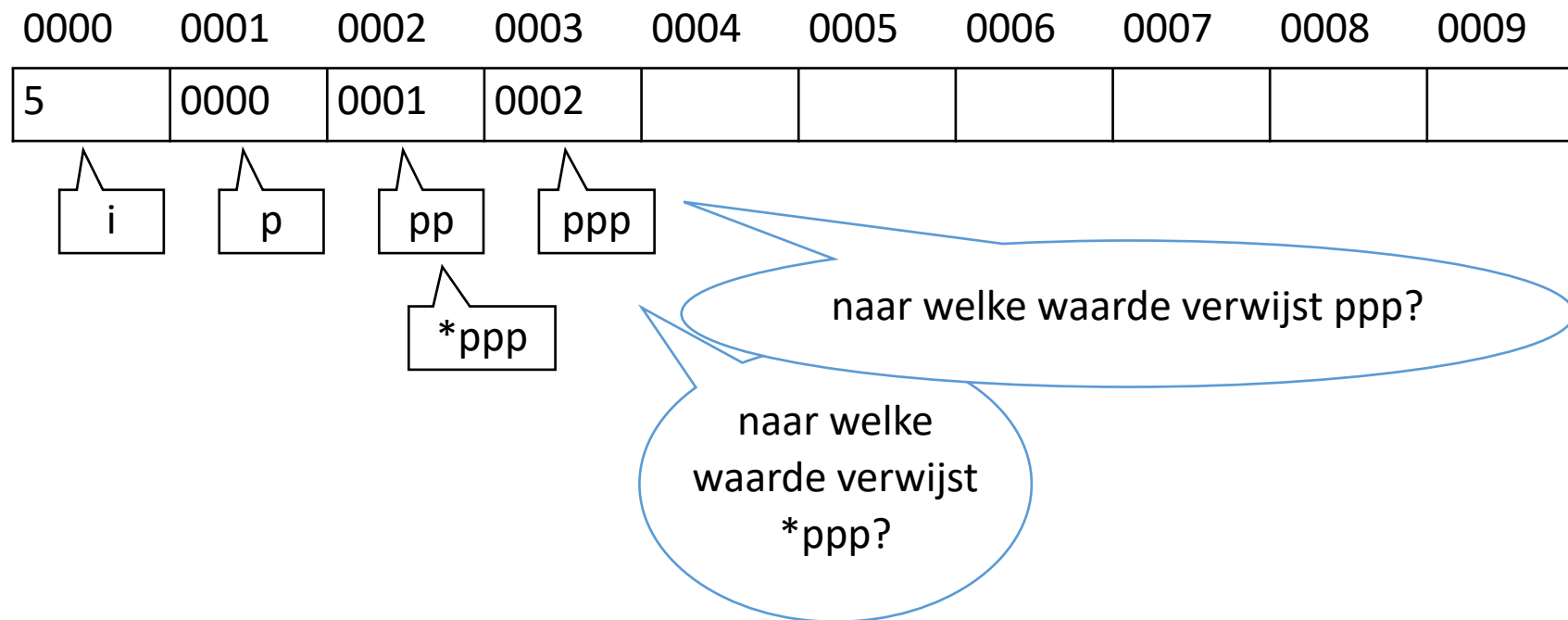
    int i = 5;
    int* p = &i;
    int** pp = &p;
    int*** ppp = &pp;

```



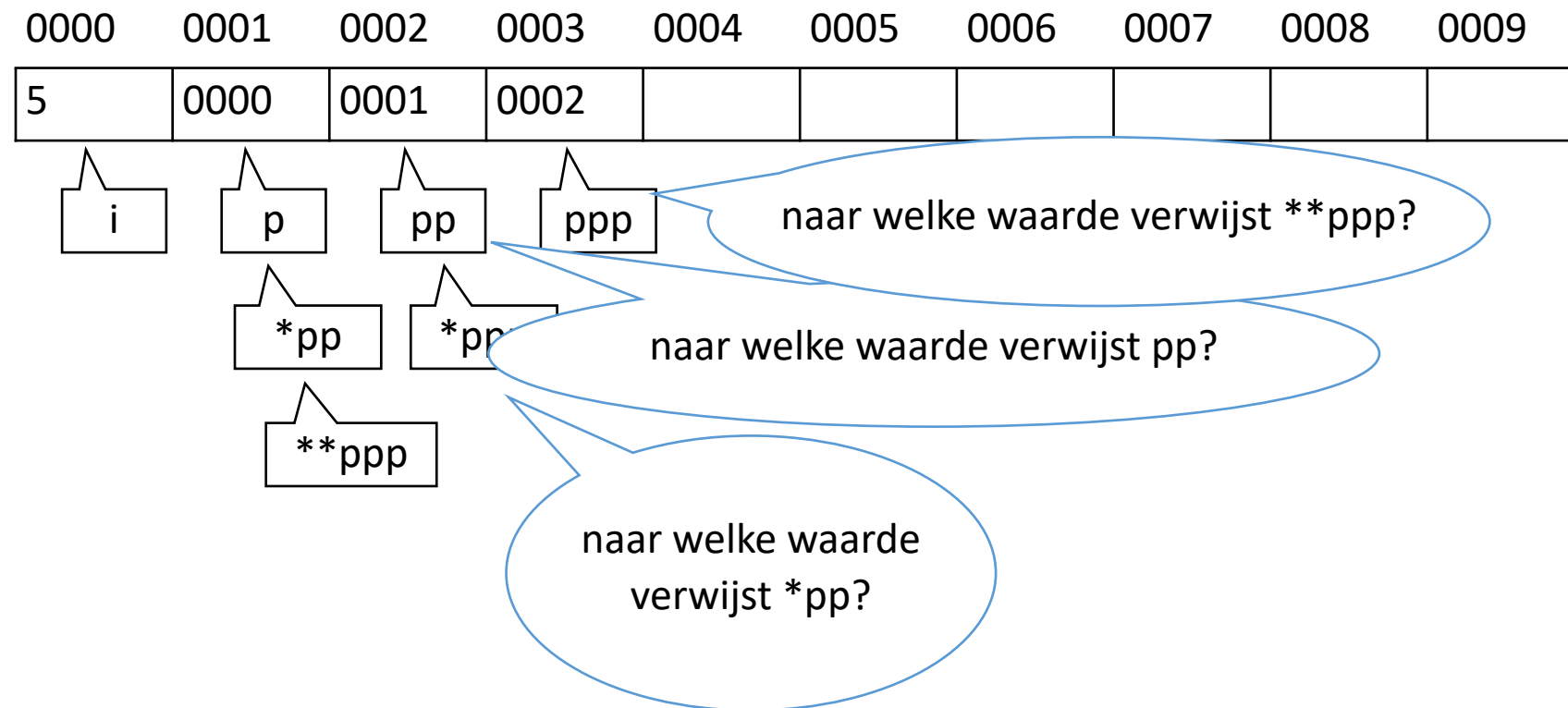
Wat zagen we vorige keer?

- Dereferentie operator *
- $*p$: object geassocieerd met de geheugenplaats waar p naar wijst



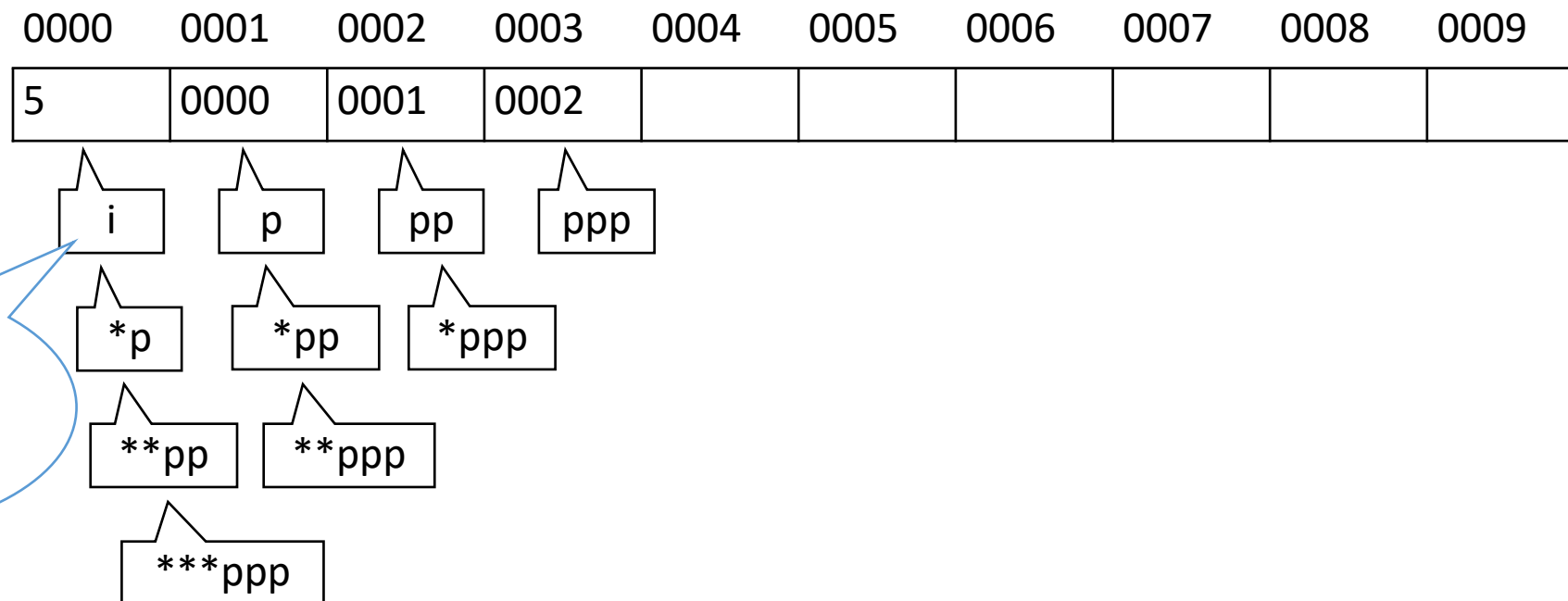
Wat zagen we vorige keer?

- Dereferentie operator *
- $*p$: object geassocieerd met de geheugenplaats waar p naar wijst



Wat zagen we vorige keer?

- Dereferentie operator *
- $*p$: object geassocieerd met de geheugenplaats waar p naar wijst



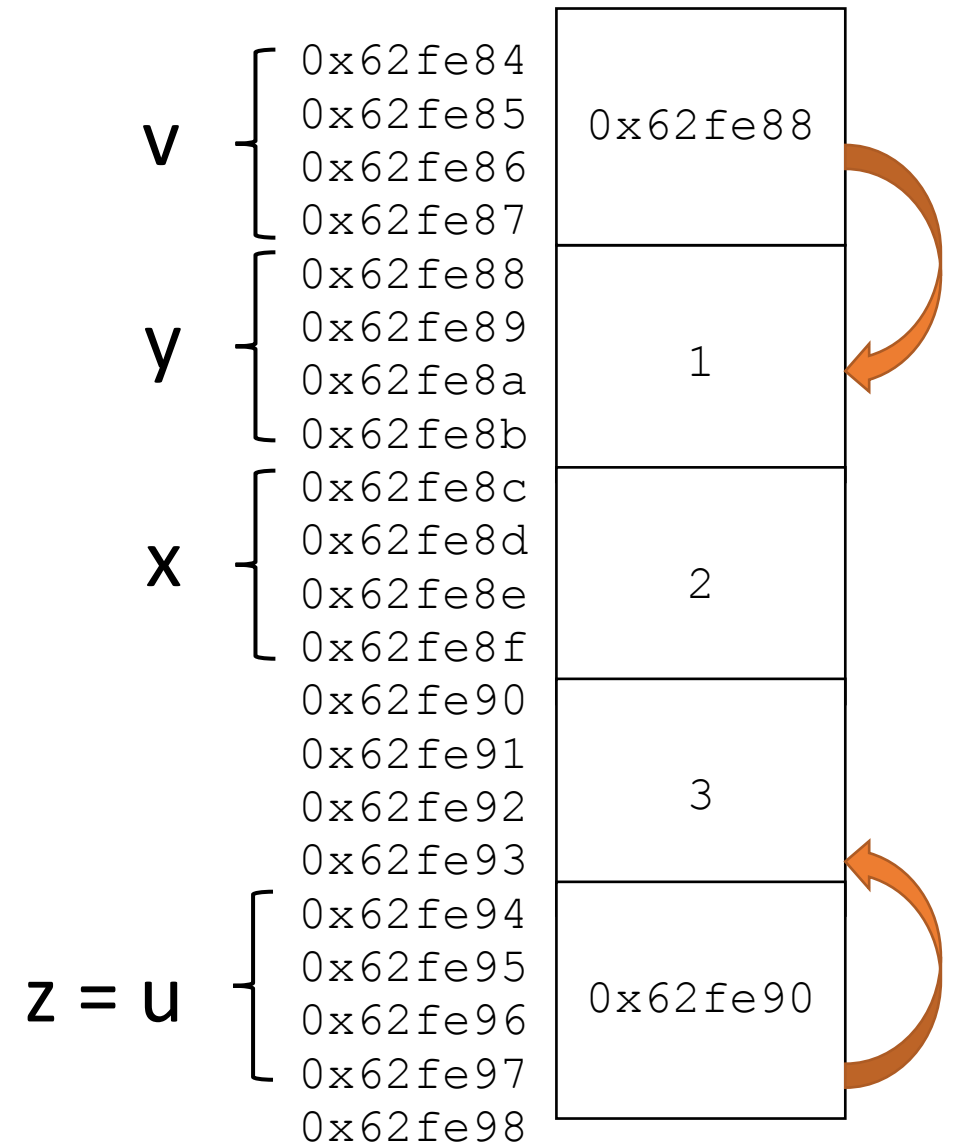
waaraan is dit
allemaal gelijk?

Vlugge vraag

- Bekijk volgend stukje code:
Is volgende illustratie van de geheugenconfiguratie consistent met dit stukje code ?

```

int    v[3]  = {1, 2, 3};
int&   x     = v[1];
int    y     = v[0];
int*   z     = v + 2;
int&   u     = *z;
  
```

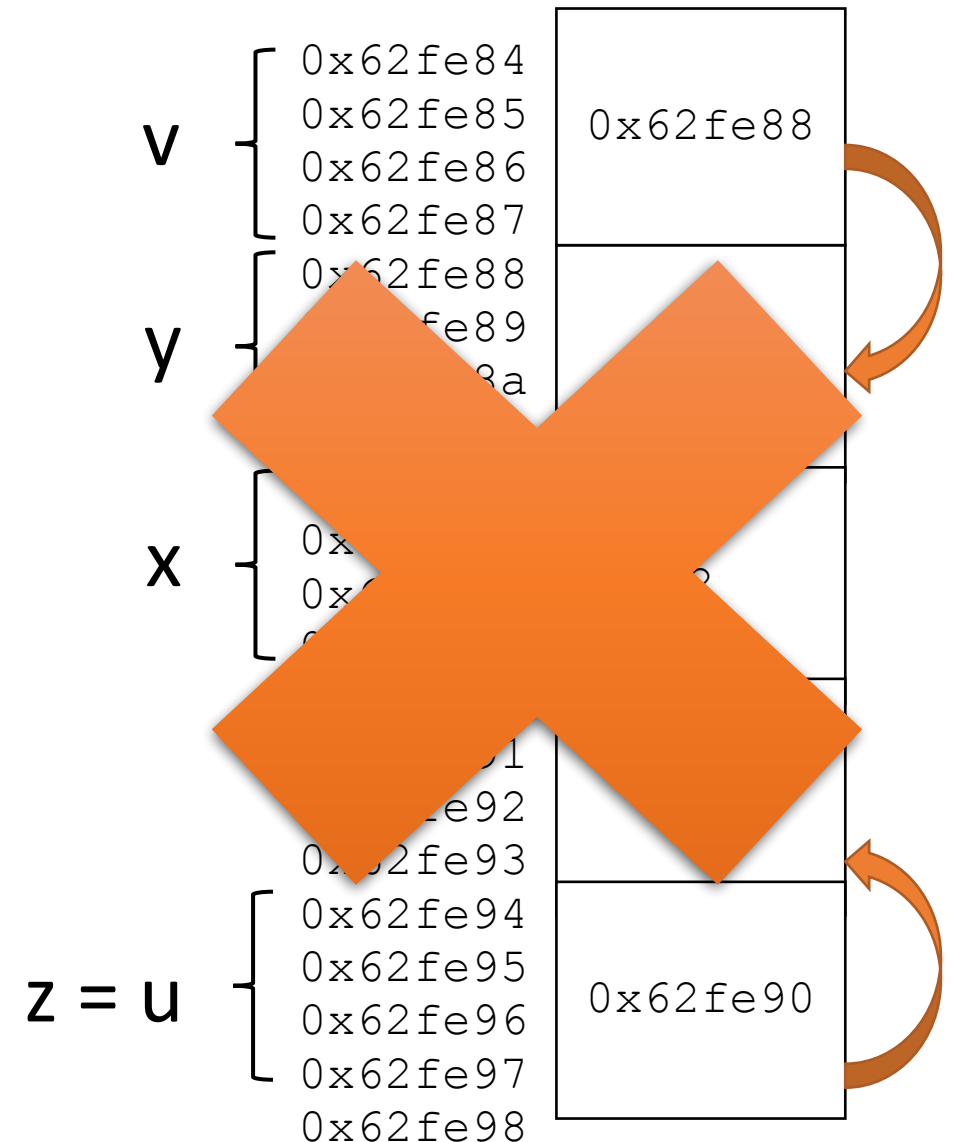


Vlugge vraag

- Bekijk volgend stukje code:
Is volgende illustratie van de geheugenconfiguratie consistent met dit stukje code ?

```

int    v[3]  = {1, 2, 3};
int&   x     = v[1];
int    y     = v[0];
int*   z     = v + 2;
int&   u     = *z;
  
```

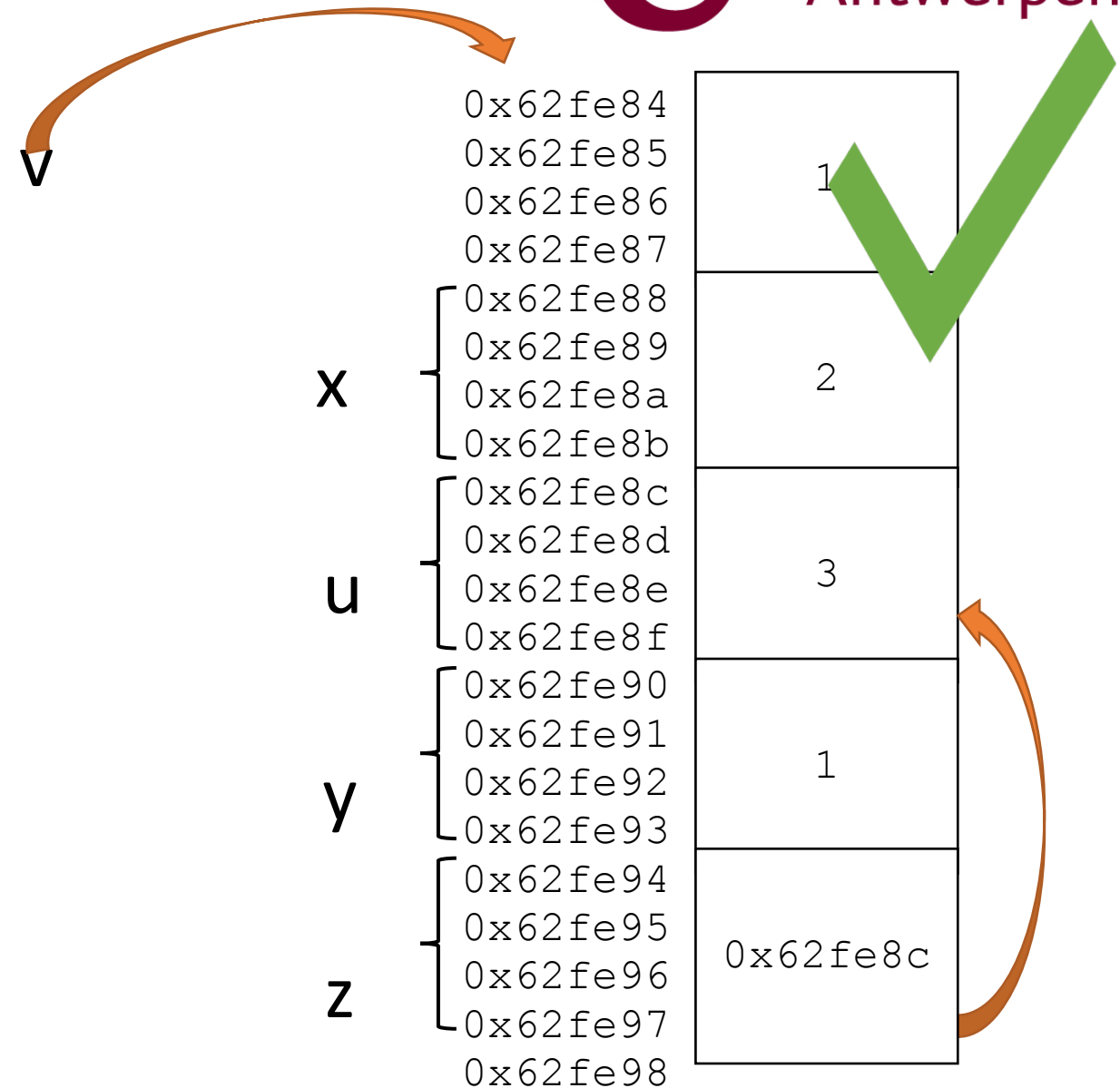


Vlugge vraag

- Bekijk volgend stukje code:
Is volgende illustratie van de geheugenconfiguratie consistent met dit stukje code ?

```

int    v[3]  = {1, 2, 3};
int&   x      = v[1];
int    y      = v[0];
int*   z      = v + 2;
int&   u      = *z;
  
```



Vlugge vraag

- Welke van de volgende uitdrukkingen zijn l-values?

```
int    v[3] = {1, 2, 3};  
int& x    = v[1];  
int    y    = v[0];  
int* z     = v + 2;  
int& u     = *z;
```

- v[0]
- *v
- v+2
- x
- y
- z
- &x
- *z
- &(*z)

Vlugge vraag

- Welke van de volgende uitdrukkingen zijn l-values?

```
int    v[3] = {1, 2, 3};  
int& x    = v[1];  
int    y    = v[0];  
int* z    = v + 2;  
int& u    = *z;
```

- `v[0]`



- `*v`



- `v+2`



- `x`



- `y`



- `z`



- `&x`



- `*z`



- `&>(*z)`



Call by reference voor grote stl containers

- Gebruik bij voorkeur *call by reference* om grote STL containers door te geven.

```
// Zoek element i via binary search in de geordende lijst v
// start en einde duiden begin en eind zoekpositie aan
bool zoek_van_tot(int i, vector<int> &v, int start, int einde) {
    if (start > einde) {
        return false;
    }
    int mid = (start + einde) / 2;
    if (v[mid] > i) {
        return zoek_van_tot(i, v, start, mid-1);
    } else if (v[mid] < i) {
        return zoek_van_tot(i, v, mid+1, einde);
    } else return true;
}
```

goed voor geheugen,
maar is dat niet gevaarlijk?

- Op deze manier vermijden we onnodige kopieën.



Benchmarking

```
// Zoek element i via binary search in de geordende lijst v
// start en einde duiden begin en eind zoekpositie aan
bool zoek_van_tot(int i, vector<int> &v, int start, int einde) {
    if (start>einde) {
        return false;
    }
    int mid=(start+einde)/2;
    if (v[mid]>i) {
        return zoek_van_tot(i,v,start,mid-1);
    } else if (v[mid]<i) {
        return zoek_van_tot(i,v,mid+1,einde);
    } else return true;
}
```



```
// Zoek element i via binary search in de geordende lijst v
// start en einde duiden begin en eind zoekpositie aan
bool zoek_van_tot_2(int i, vector<int> v, int start, int einde) {
    if (start>einde) {
        return false;
    }
    int mid=(start+einde)/2;
    if (v[mid]>i) {
        return zoek_van_tot(i,v,start,mid-1);
    } else if (v[mid]<i) {
        return zoek_van_tot(i,v,mid+1,einde);
    } else return true;
}
```



```
int main() {
    int N = 10 000 000;

    vector<int> v;

    for (int i = 0; i < N;i++) {
        v.push_back(i);
    }

    clock_t start = clock();

    for (int j = 0; j < 1000;j++) {
        zoek_van_tot(rand() % 2 * N, v, 0, v.size()-1);
    }

    clock_t einde = clock();
    cout << "Duur van het programma : " << einde - start << endl;

    start = clock();
    for (int j = 0; j < 1000;j++) {
        zoek_van_tot_2(rand() % 2 * N, v, 0, v.size()-1);
    }

    einde = clock();
    cout << "Duur van het programma : " << einde - start << endl;

    return 0;
}
```

#include <time.h>

The results

N	By Reference	By Value
10 000	0	3
100 000	0	18
1 000 000	1	1 179
10 000 000	1	13 464

Time in milliseconds; 0 indicates less than 1 millisecond

And the winner is:



Waarom niet altijd by ref?

- kan, maar ...
 - soms WIL je een kopie
 - je wint geen geheugen bij basistypes:
 - `int i; 4 bytes`
 - ref (implementatie via pointers dus `int* pi = &i`) 4 bytes
 - je VERLIEST performantie!
 - je moet `*pi` gebruiken (een extra operatie)

Wat zagen we vorige week?

```
void deling2(int deeltal, int deler, int* quotient, int* rest) {  
    *quotient = x / y;  
    *rest = x % y;  
}
```

```
int main() {  
    int dt = 100;  
    int dr = 13;  
    int q;  
    int r;  
    deling2(dt, dr, &q, &r);  
    cout << q << " " << r << endl;  
    return 0;  
}
```

zo wordt het geïmplementeerd in C++

Let op! Problemen met pointers

- Probleem 1: niet-geïnitieerde pointer ('wild')

```
int* p;  
*p = 5;
```

*Gaat op een willekeurige positie in het geheugen "5" schrijven.
Kan at runtime een segmentation fault geven.*

Let op: want dit kan SOMS
een fout geven

dus heel moeilijk te vinden,
daarom valgrind!

Altijd variabelen initialiseren!
Bv `p = nullptr` (of `p = 0`)

Let op! Problemen met pointers

- Probleem 2: pointer naar een object dat niet meer bestaat ('dangling')

```
int* f() {  
    int i;  
    return &i;  
}
```

Geeft een pointer naar een plaats op de stack die na de functie niet langer gereserveerd is en mogelijk herbruikt wordt.

Let op! Dangling pointer

```
int* max_of_two(int x, int y) {  
    if (x>y) {  
        return &x;  
    } else {  
        return &y;  
    }  
}
```


Zo wel goed:

```
int* grootste(int &a, int &b) {  
    if(a>b) return &a;  
    return &b;  
}  
  
int main() {  
    int a=3;  
    int b=5;  
    int* gr=grootste(a,b);  
    *gr=10;  
  
    cout << a << " " << b << endl;  
  
    return 0;  
}
```

3 10

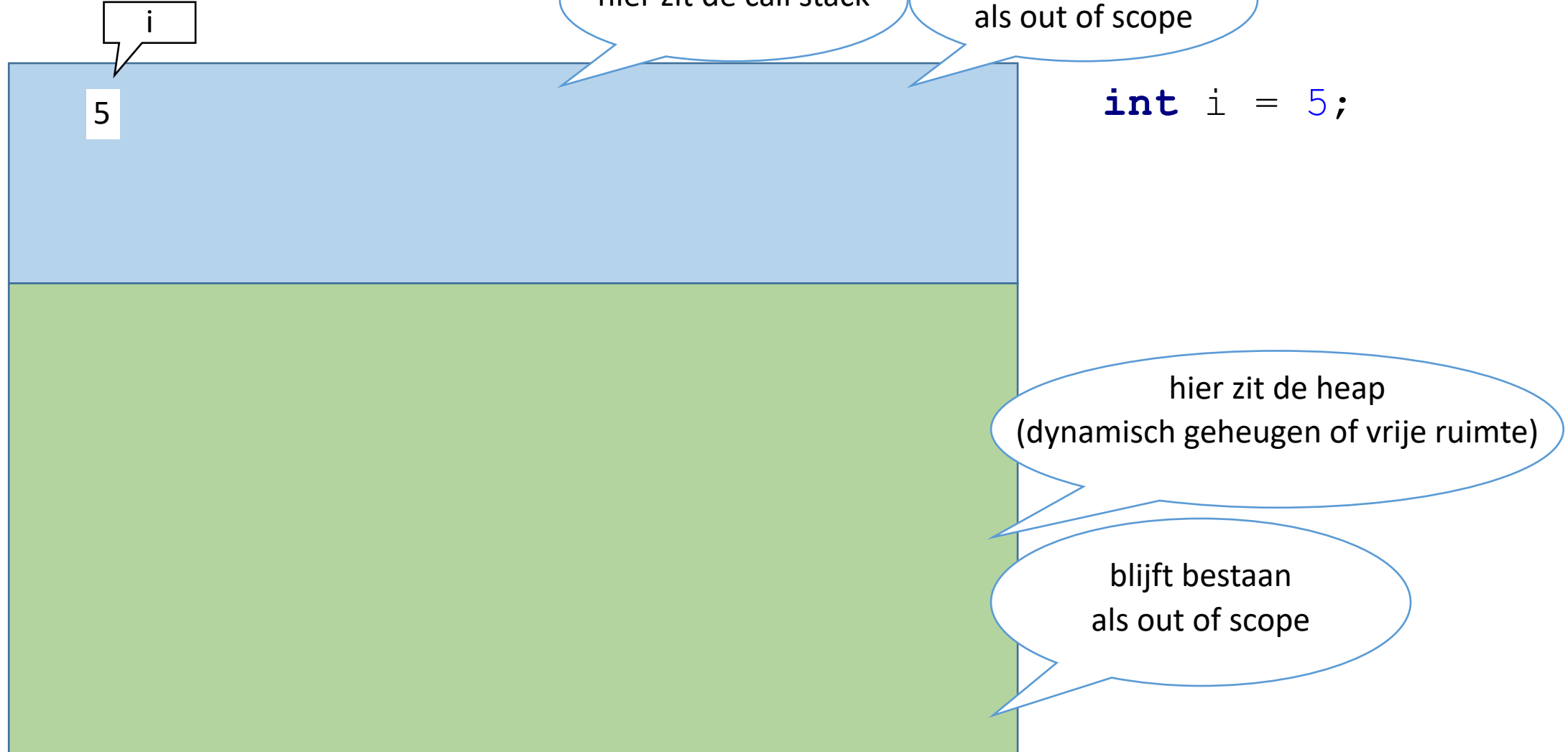
Dynamic memory allocation

- Door de scope van variabelen hebben we weinig controle over de levensduur van variabelen.
- Via dynamic memory allocation kunnen we toch controle krijgen.
- Maar ...
 - “With great power comes great responsibility”
 - zelf geheugen aanmaken betekent ook zelf geheugen vrijgeven
- Hoe kunnen we objecten aanmaken tijdens de duur van het programma?

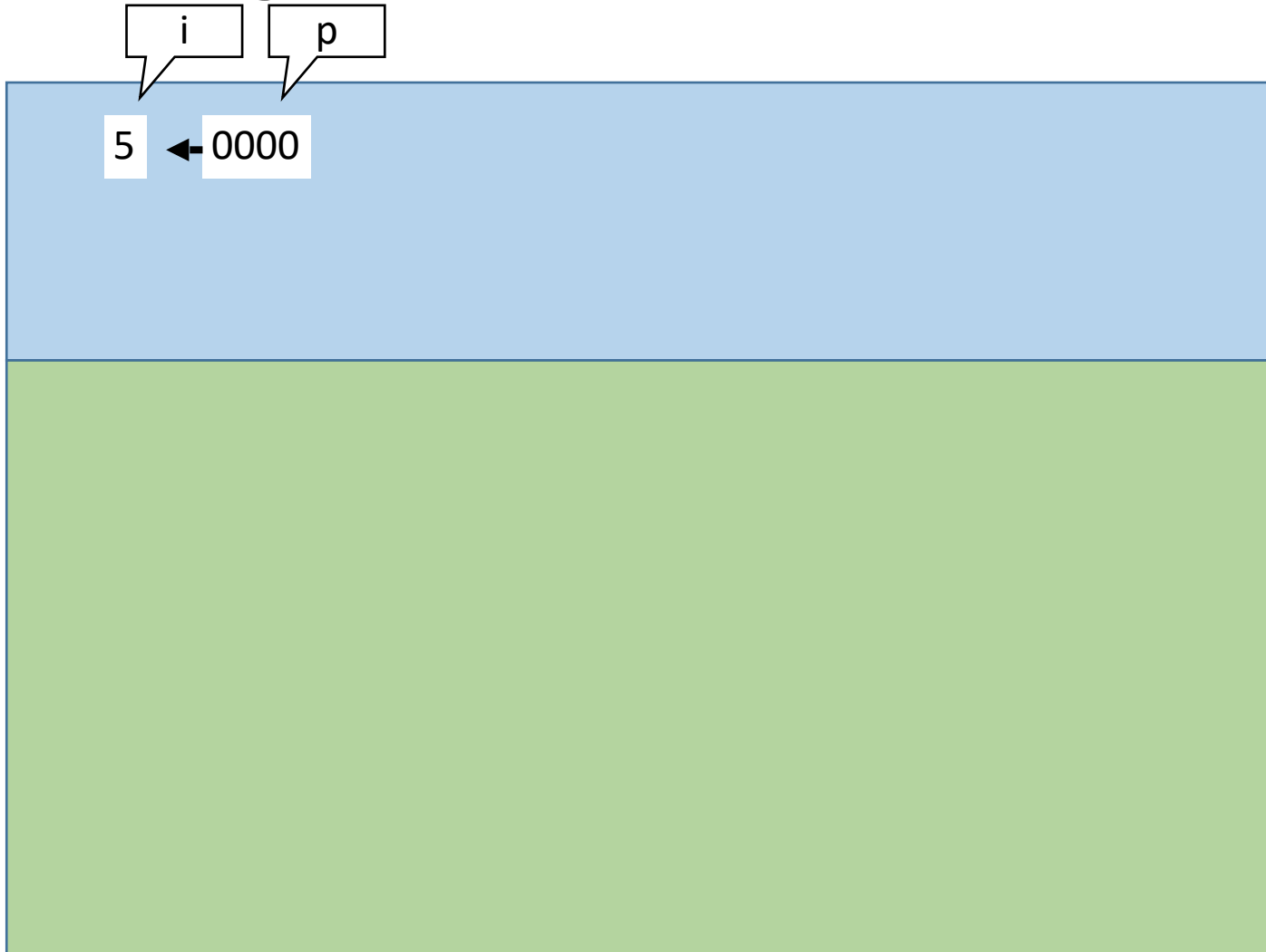
Dynamic memory allocation

- Via functie “new” kunnen we dynamisch geheugen reserveren
 - new geeft ons een pointer naar het gereserveerde geheugen terug
- Via “delete” kunnen we het terug vrijgeven
- Dit geheugen **blijft bestaan tot we het expliciet zelf verwijderen**, ook als we de functie waarin het geheugen gereserveerd werd, verlaten.

Geheugenbeheer

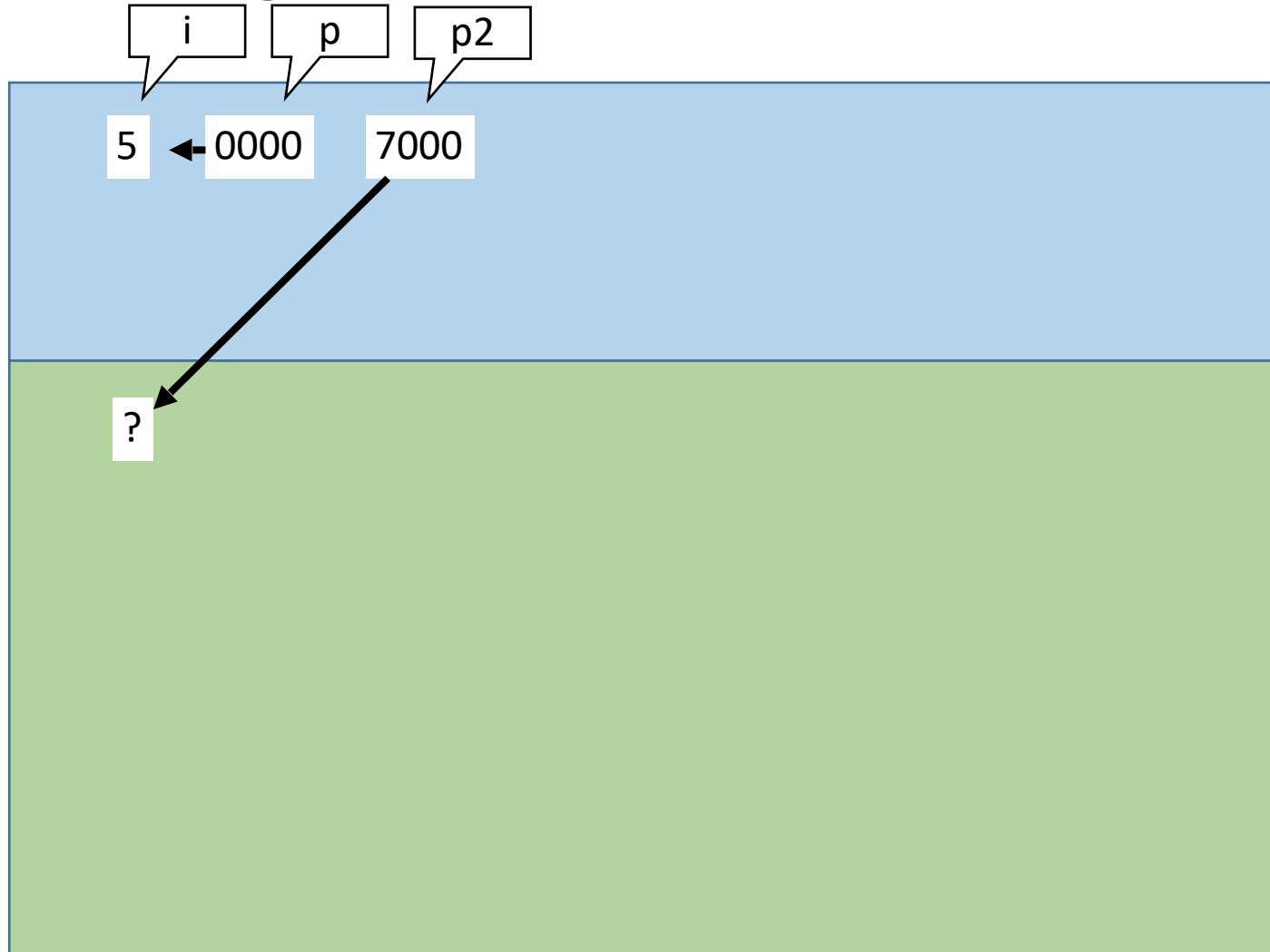


Geheugenbeheer



```
int i = 5;  
int* p = &i;
```

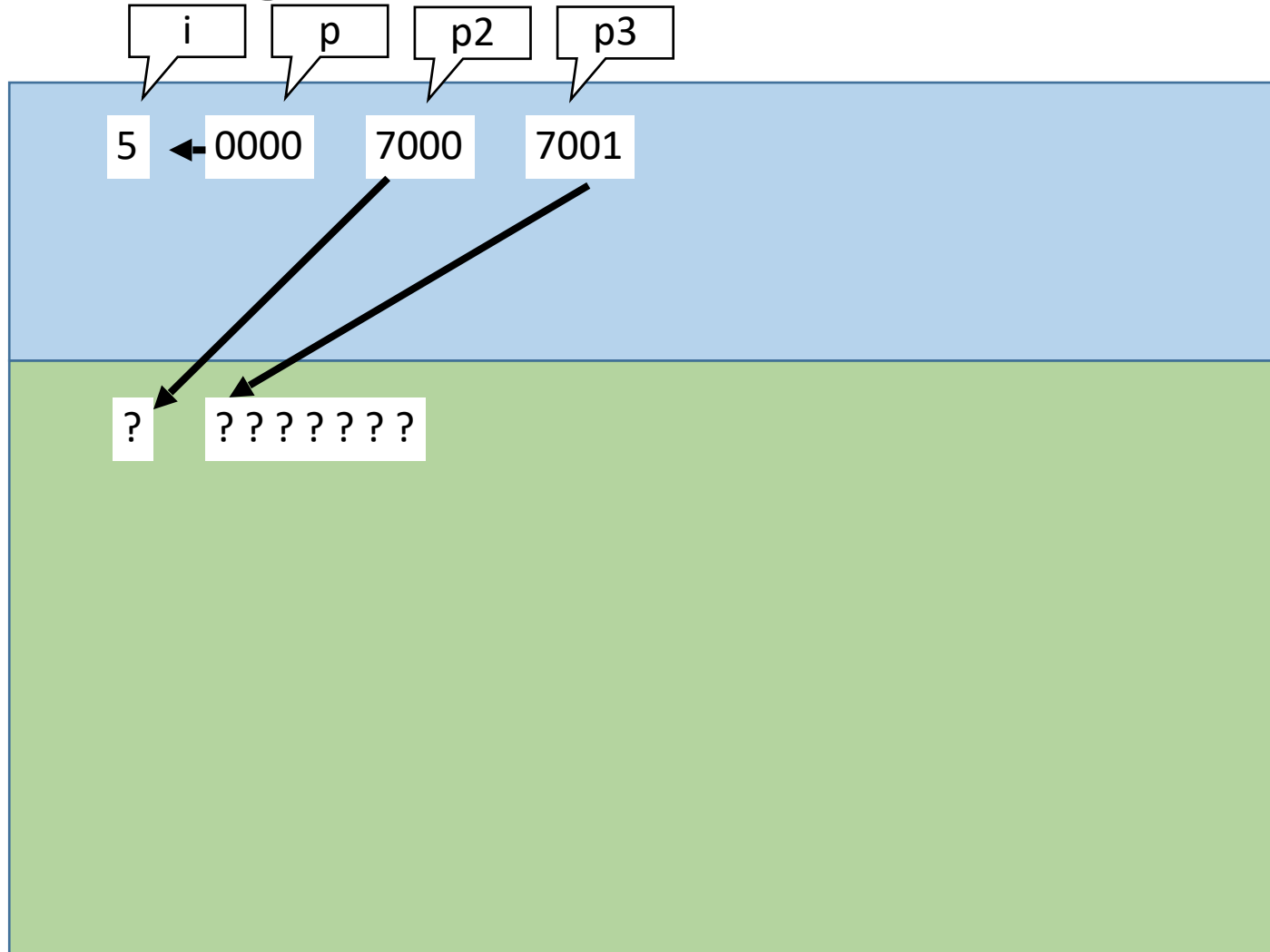
Geheugenbeheer



```
int i = 5;  
int* p = &i;
```

```
int* p2 = new int;
```

Geheugenbeheer



```
int i = 5;
int* p = &i;
```

```
int* p2 = new int;
int* p3 = new int[7];
```

Dynamic memory allocation

- Via “new type” krijgen we een pointer naar een geheugenplaats waarin we een waarde van dat type kunnen opslaan
- Via “new type[int expressie]” krijgen we een pointer naar een array terug.

```
cout << "Geef n : ";  
int n;  
cin >> n;  
int* da=new int[n];  
for (int i=0;i<n;i++) {  
    da[i]=i;  
}  
cout << da[0] << " " << da[n-1] << endl;
```

da = int[n] mag niet!

bekend at
compile time

da = int[3] komt op call stack
da = new int[3] komt op heap/free mem

bekend at
run time

```
Geef n : 7  
0 6
```


Nut van new en new[]?

- Via new kunnen we complexe objecten aanmaken die *blijven bestaan*
- Deze structuur kunnen we in een functie aanmaken in de vrije ruimte, en dan een pointer naar dit complexe object teruggeven

Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}

```

0x62fe84
 0x62fe85
 0x62fe86
 0x62fe87
 0x62fe88
 0x62fe89
 0x62fe8a
 0x62fe8b
 0x62fe8c
 0x62fe8d
 0x62fe8e
 0x62fe8f
 0x62fe90
 0x62fe91
 0x62fe92
 0x62fe93
 0x62fe94

stack

0xf316d8
 0xf316d9
 0xf316da
 0xf316db
 0xf316dc
 0xf316dd
 0xf316de
 0xf316df
 0xf316e0
 0xf316e1
 0xf316e2
 0xf316e3
 0xf316e4
 0xf316e5
 0xf316e6
 0xf316e7
 0xf316e8

Free
mem.

Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

0x62fe84
 0x62fe85
 0x62fe86
 0x62fe87
 0x62fe88
 0x62fe89
 0x62fe8a
 0x62fe8b
 0x62fe8c
 0x62fe8d
 0x62fe8e
 0x62fe8f
 0x62fe90
 0x62fe91
 0x62fe92
 0x62fe93
 0x62fe94

stack

0xf316d8
 0xf316d9
 0xf316da
 0xf316db
 0xf316dc
 0xf316dd
 0xf316de
 0xf316df
 0xf316e0
 0xf316e1
 0xf316e2
 0xf316e3
 0xf316e4
 0xf316e5
 0xf316e6
 0xf316e7
 0xf316e8

Free
mem.

Voorbeeld: new in functie

```
int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```

0x62fe84
0x62fe85
0x62fe86
0x62fe87
0x62fe88
0x62fe89
0x62fe8a
0x62fe8b
0x62fe8c
0x62fe8d
0x62fe8e
0x62fe8f
0x62fe90
0x62fe91
0x62fe92
0x62fe93
0x62fe94

int* v

stack

0xf316d8
0xf316d9
0xf316da
0xf316db
0xf316dc
0xf316dd
0xf316de
0xf316df
0xf316e0
0xf316e1
0xf316e2
0xf316e3
0xf316e4
0xf316e5
0xf316e6
0xf316e7
0xf316e8

Free
mem.

...

...

Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

0x62fe84
 0x62fe85
 0x62fe86
 0x62fe87
 0x62fe88
 0x62fe89
 0x62fe8a
 0x62fe8b
 0x62fe8c
 0x62fe8d
 0x62fe8e
 0x62fe8f
 0x62fe90
 0x62fe91
 0x62fe92
 0x62fe93
 0x62fe94

int* v

stack

...

0xf316d8
 0xf316d9
 0xf316da
 0xf316db
 0xf316dc
 0xf316dd
 0xf316de
 0xf316df
 0xf316e0
 0xf316e1
 0xf316e2
 0xf316e3
 0xf316e4
 0xf316e5
 0xf316e6
 0xf316e7
 0xf316e8

Free
mem.

...

Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

0x62fe84
 0x62fe85
 0x62fe86
 0x62fe87
 0x62fe88
 0x62fe89
 0x62fe8a
 0x62fe8b
 0x62fe8c
 0x62fe8d
 0x62fe8e
 0x62fe8f
 0x62fe90
 0x62fe91
 0x62fe92
 0x62fe93
 0x62fe94

int* v

int n = 3

stack

0xf316d8
 0xf316d9
 0xf316da
 0xf316db
 0xf316dc
 0xf316dd
 0xf316de
 0xf316df
 0xf316e0
 0xf316e1
 0xf316e2
 0xf316e3
 0xf316e4
 0xf316e5
 0xf316e6
 0xf316e7
 0xf316e8

Free
mem.

Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

0x62fe84
 0x62fe85
 0x62fe86
 0x62fe87
 0x62fe88
 0x62fe89
 0x62fe8a
 0x62fe8b
 0x62fe8c
 0x62fe8d
 0x62fe8e
 0x62fe8f
 0x62fe90
 0x62fe91
 0x62fe92
 0x62fe93
 0x62fe94

int* v

int n = 3

stack

0xf316d8
 0xf316d9
 0xf316da
 0xf316db
 0xf316dc
 0xf316dd
 0xf316de
 0xf316df
 0xf316e0
 0xf316e1
 0xf316e2
 0xf316e3
 0xf316e4
 0xf316e5
 0xf316e6
 0xf316e7
 0xf316e8

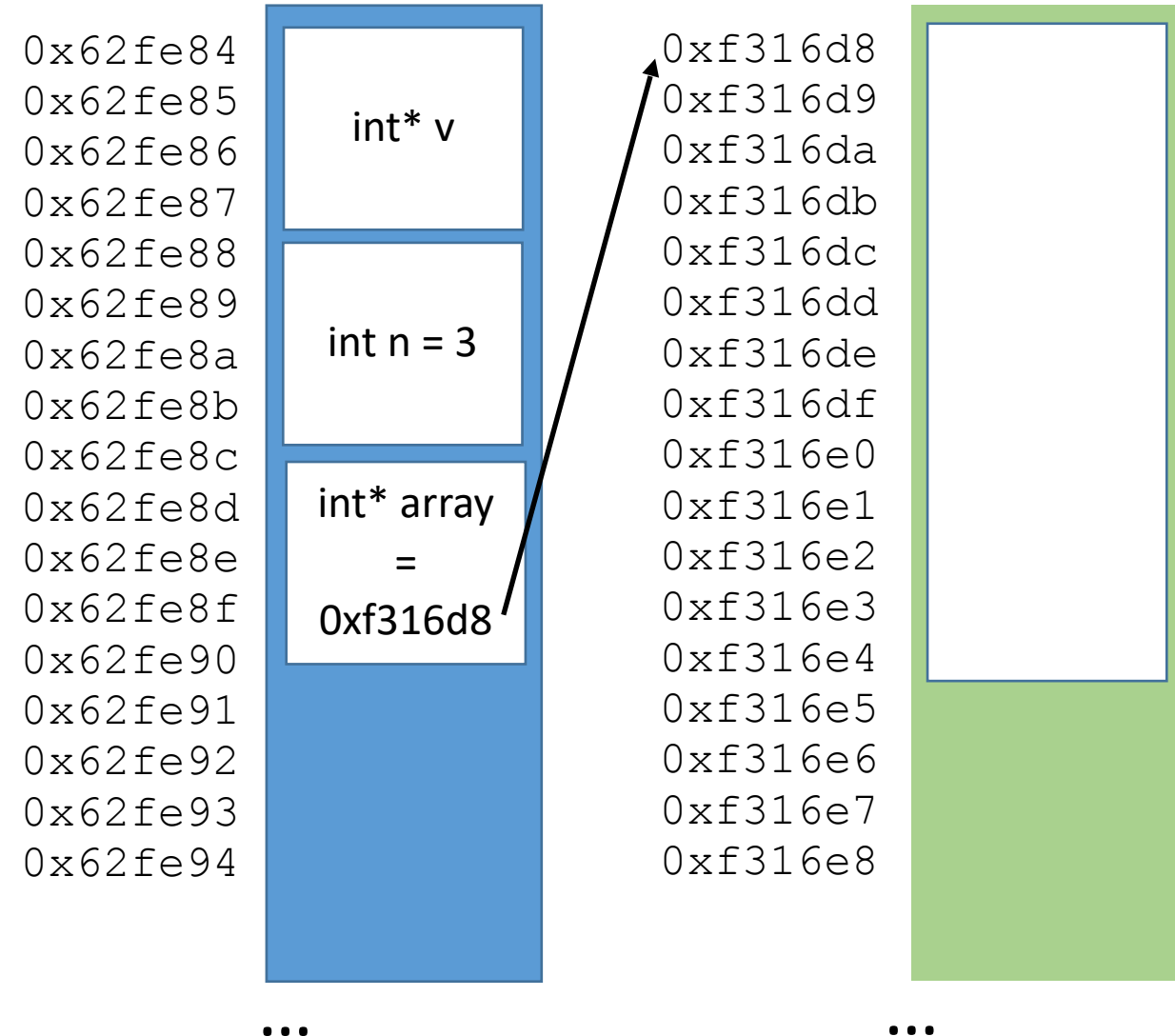
Free
mem.

Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

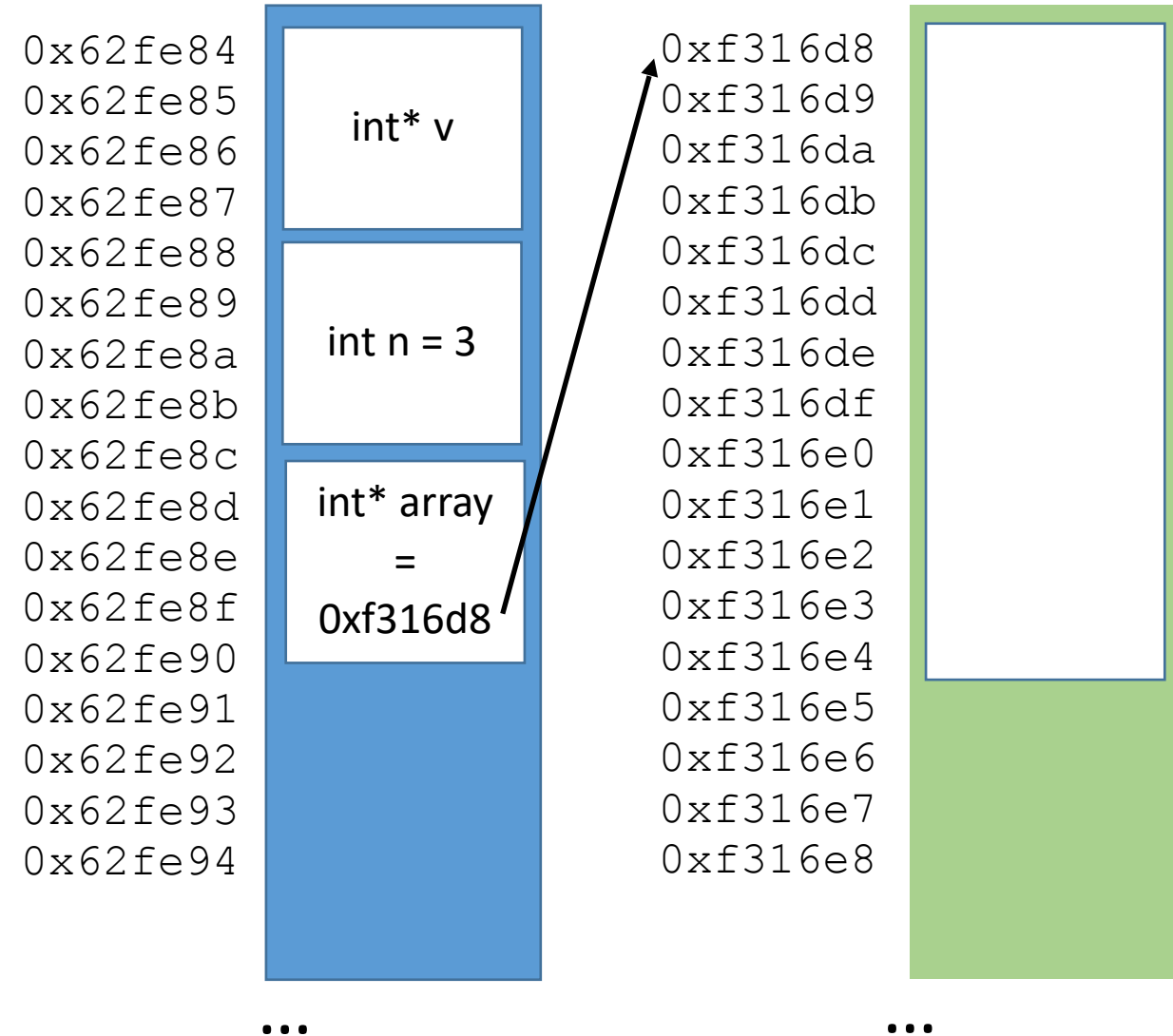


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

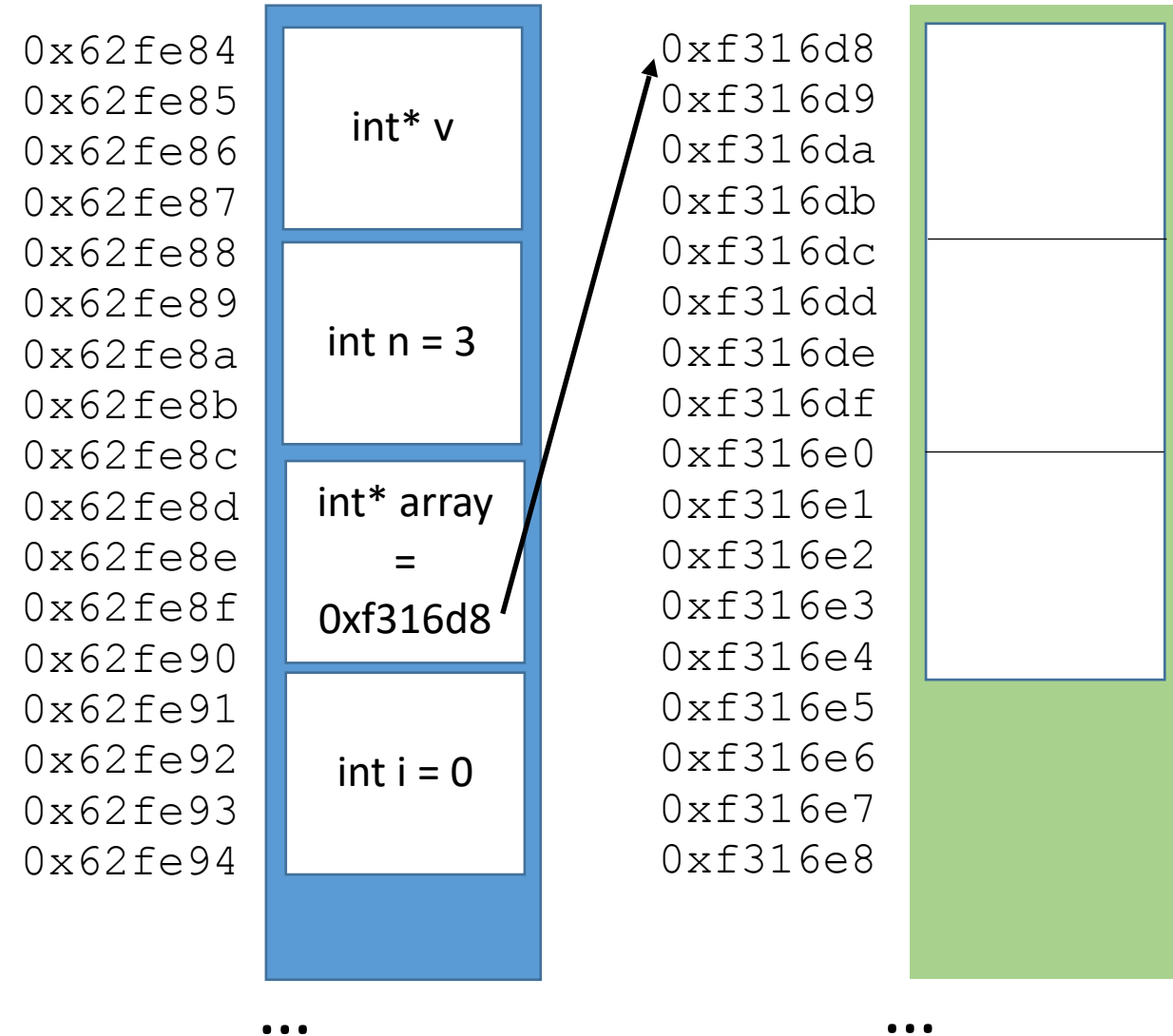


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

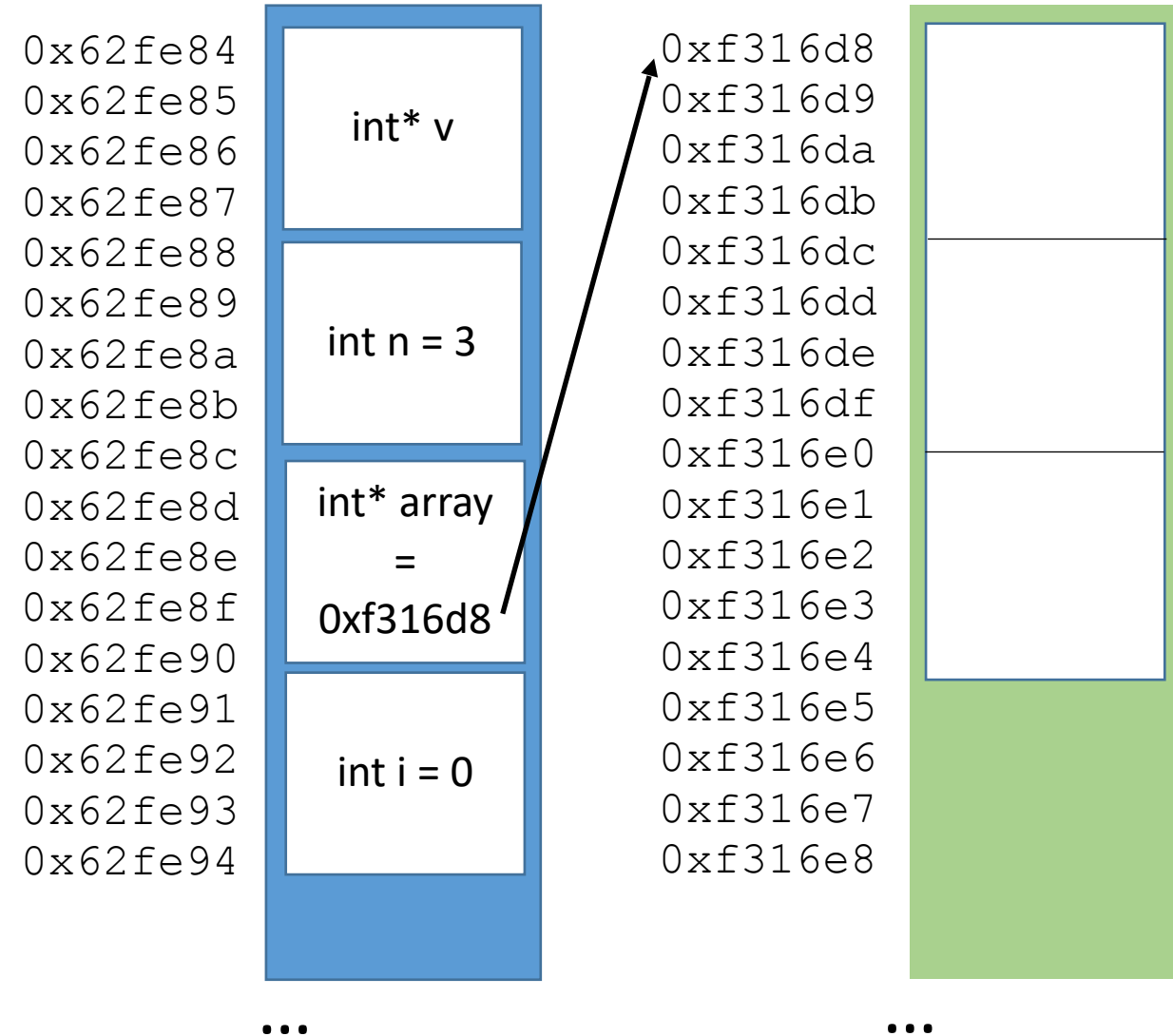


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

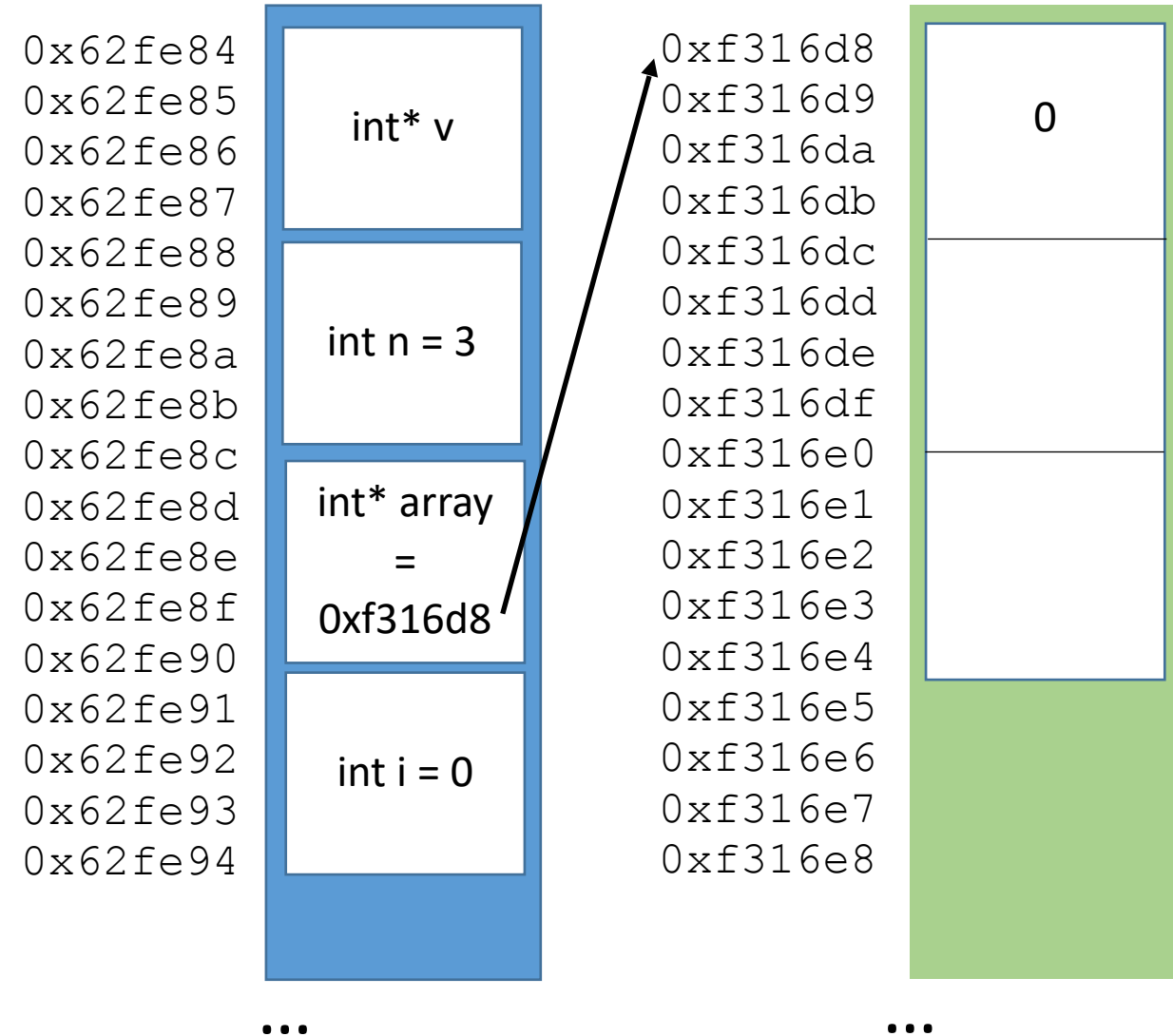


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

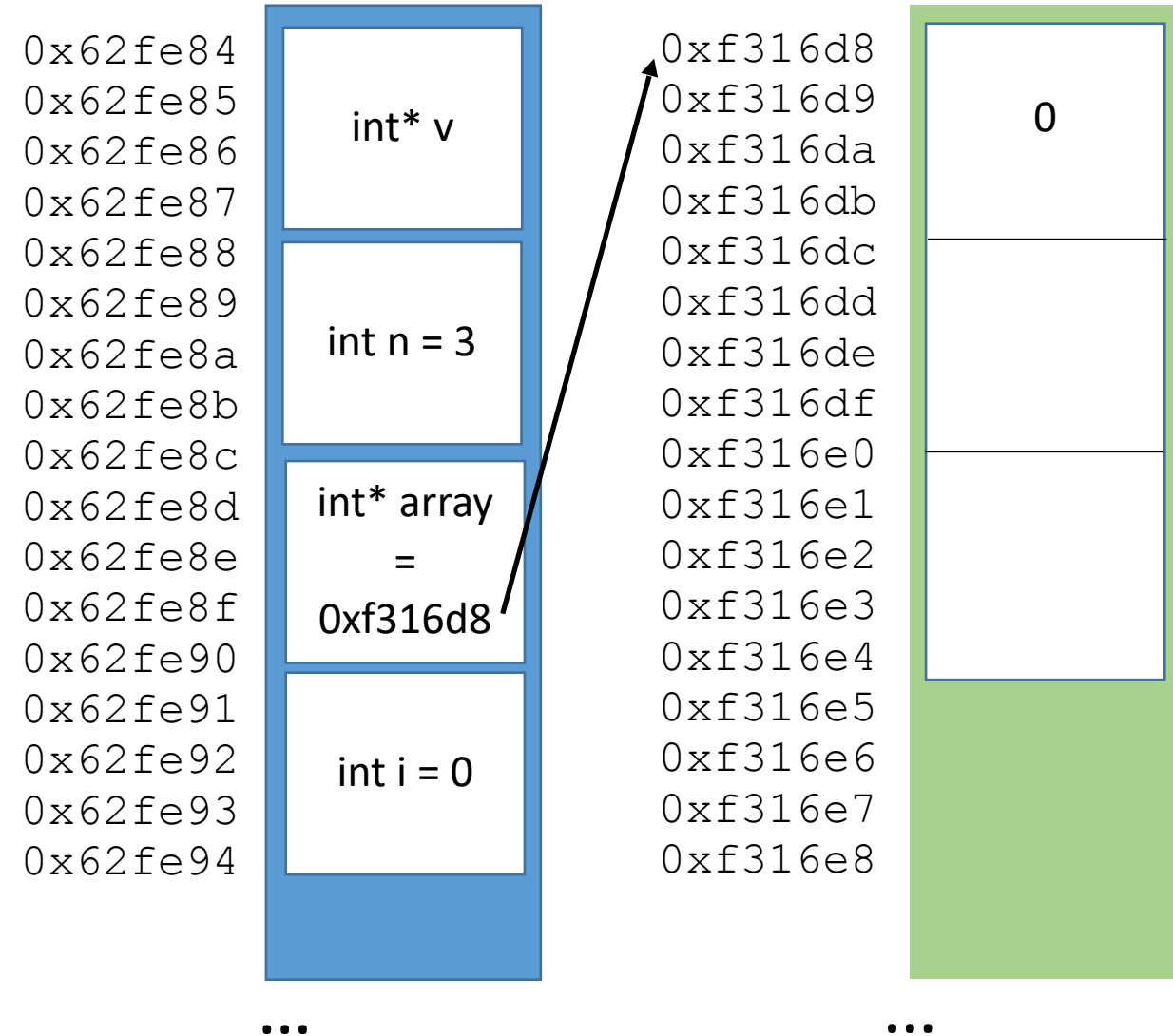


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

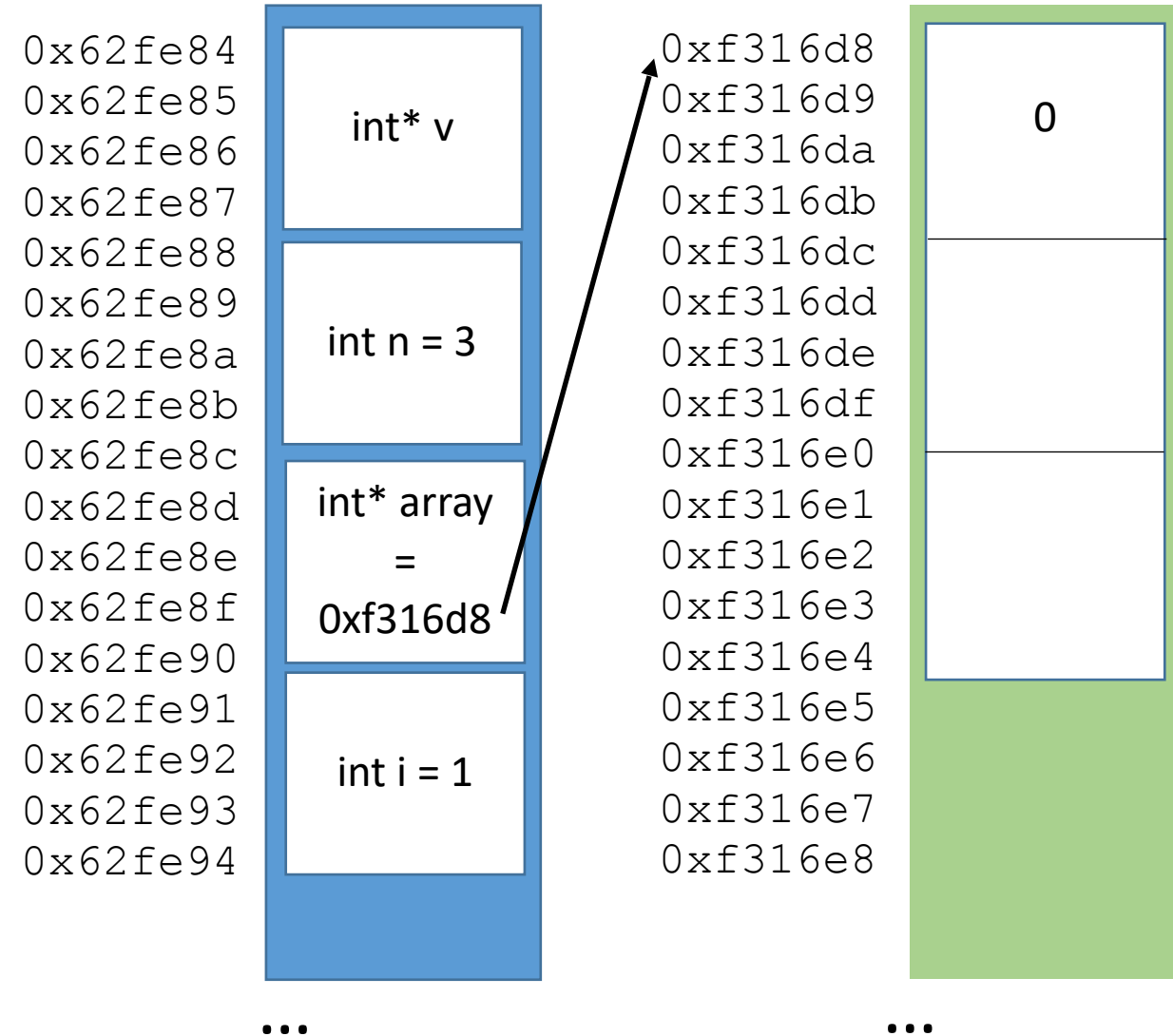


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```



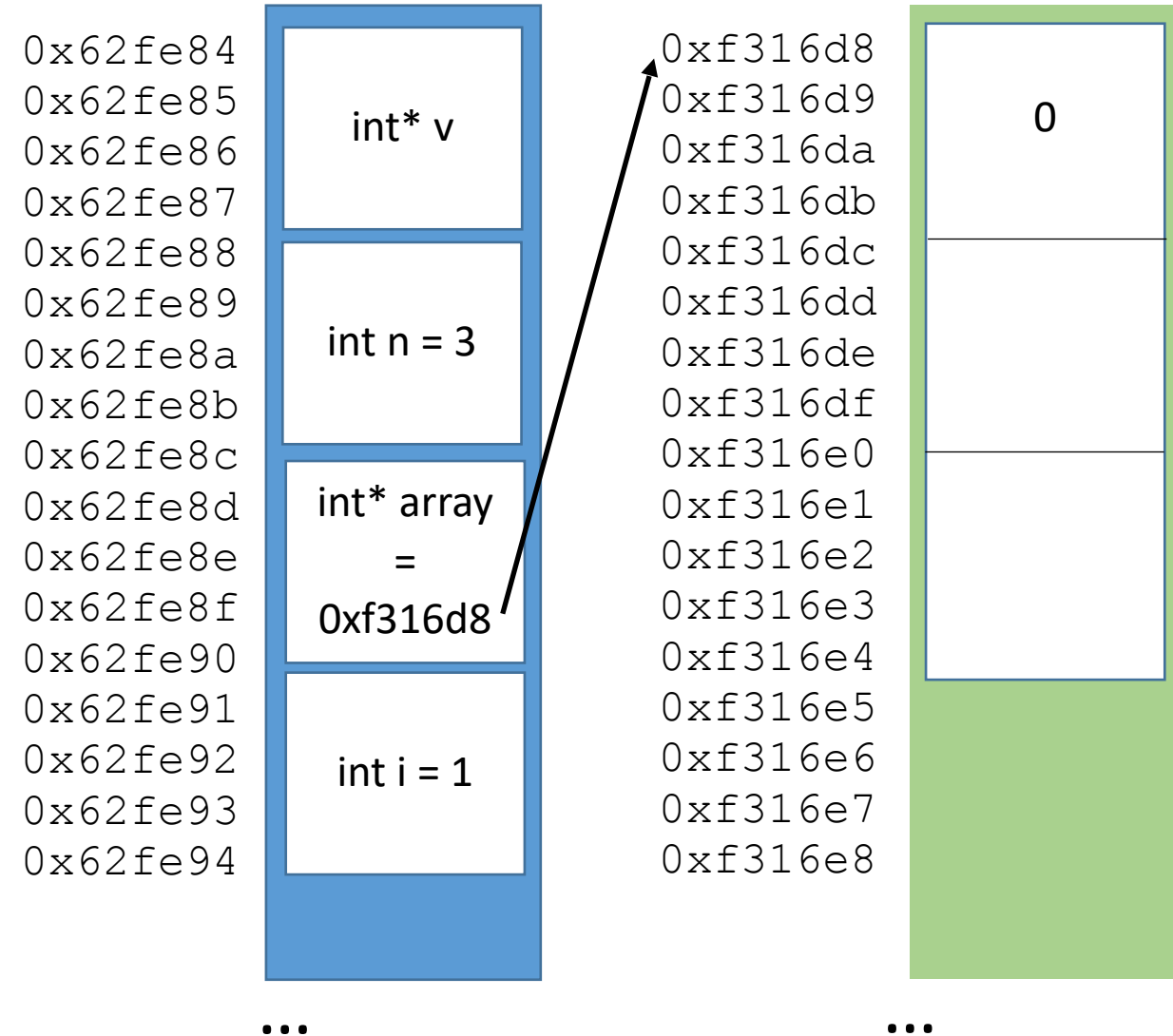
Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}

```

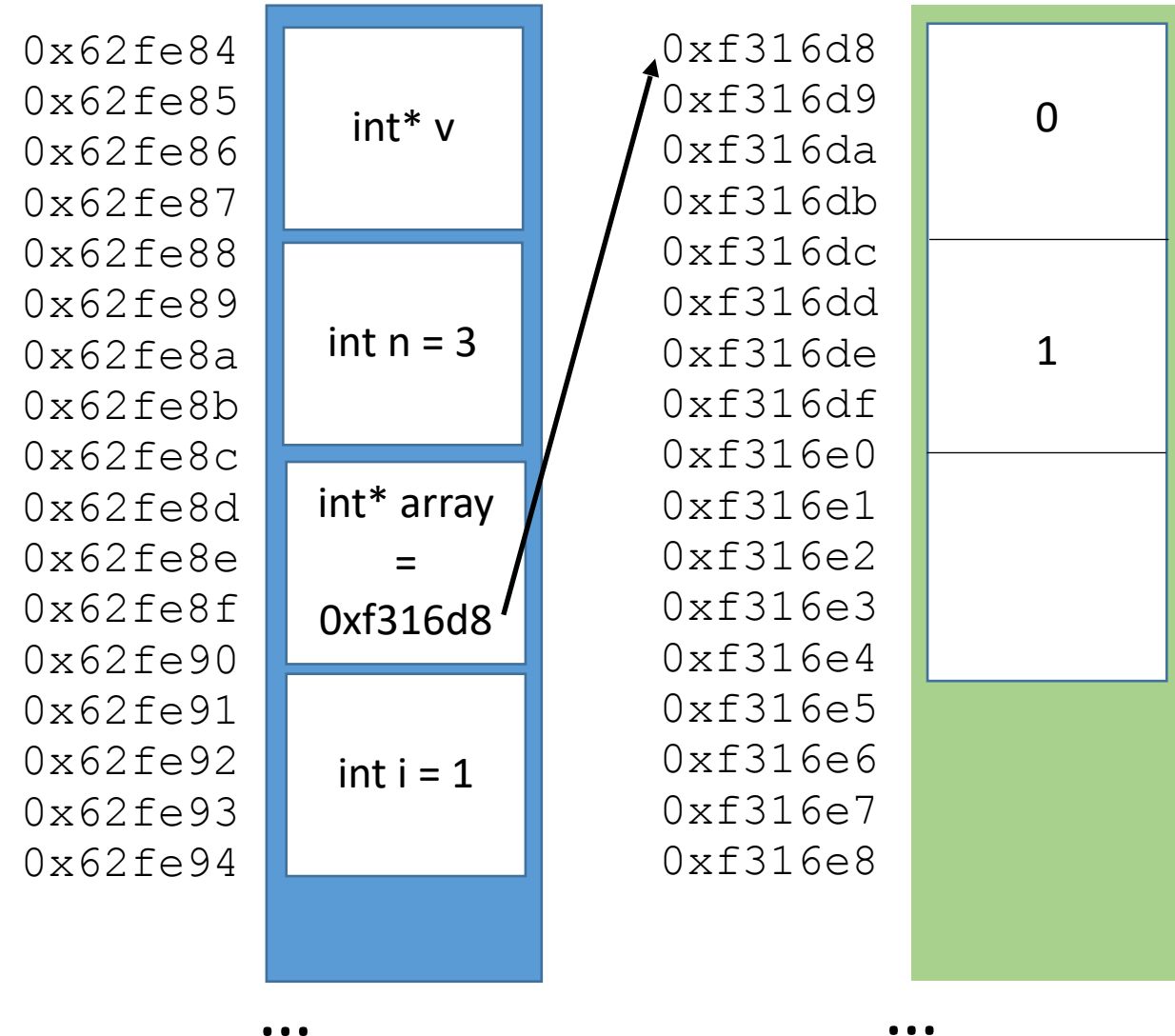


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

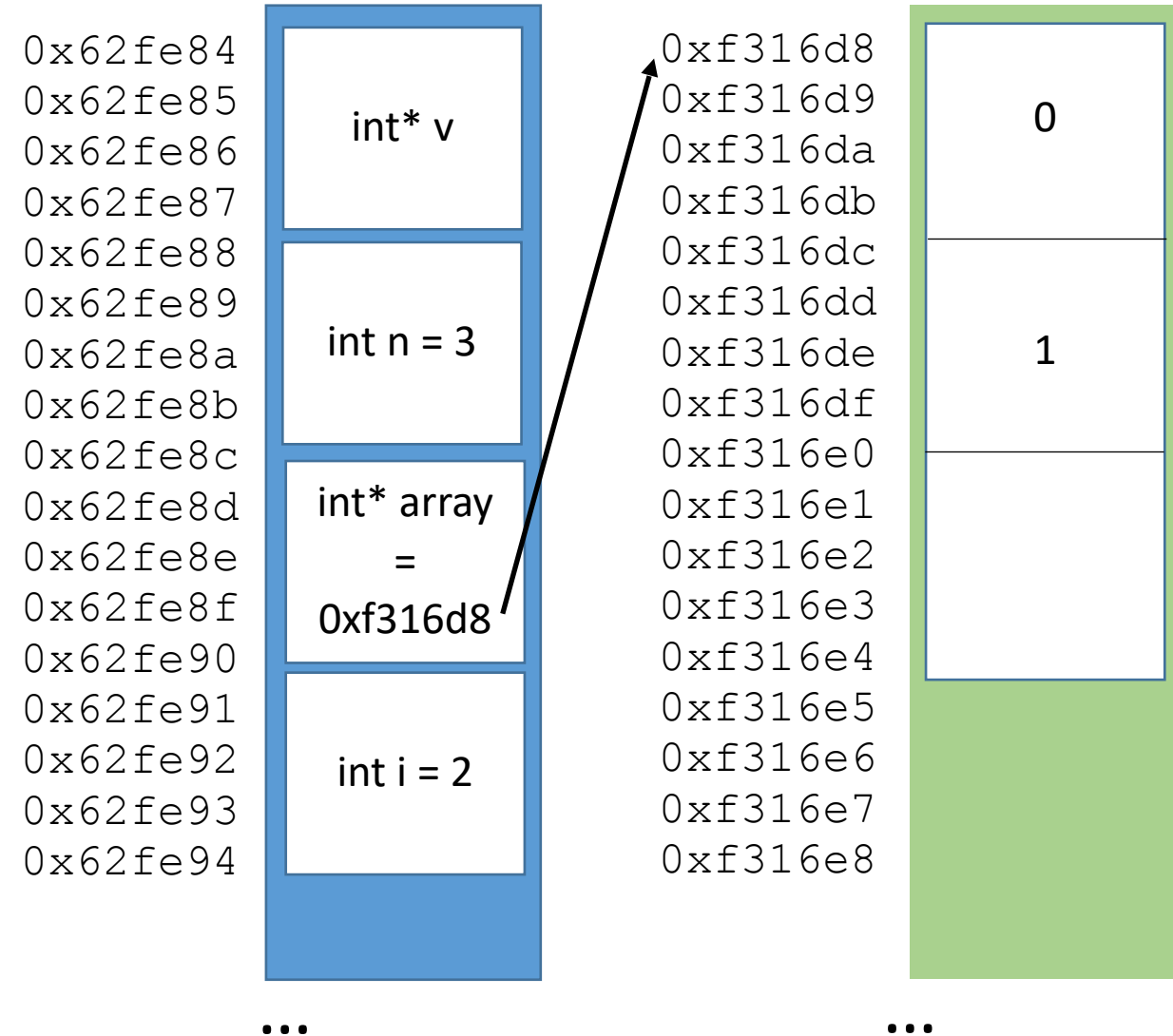


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

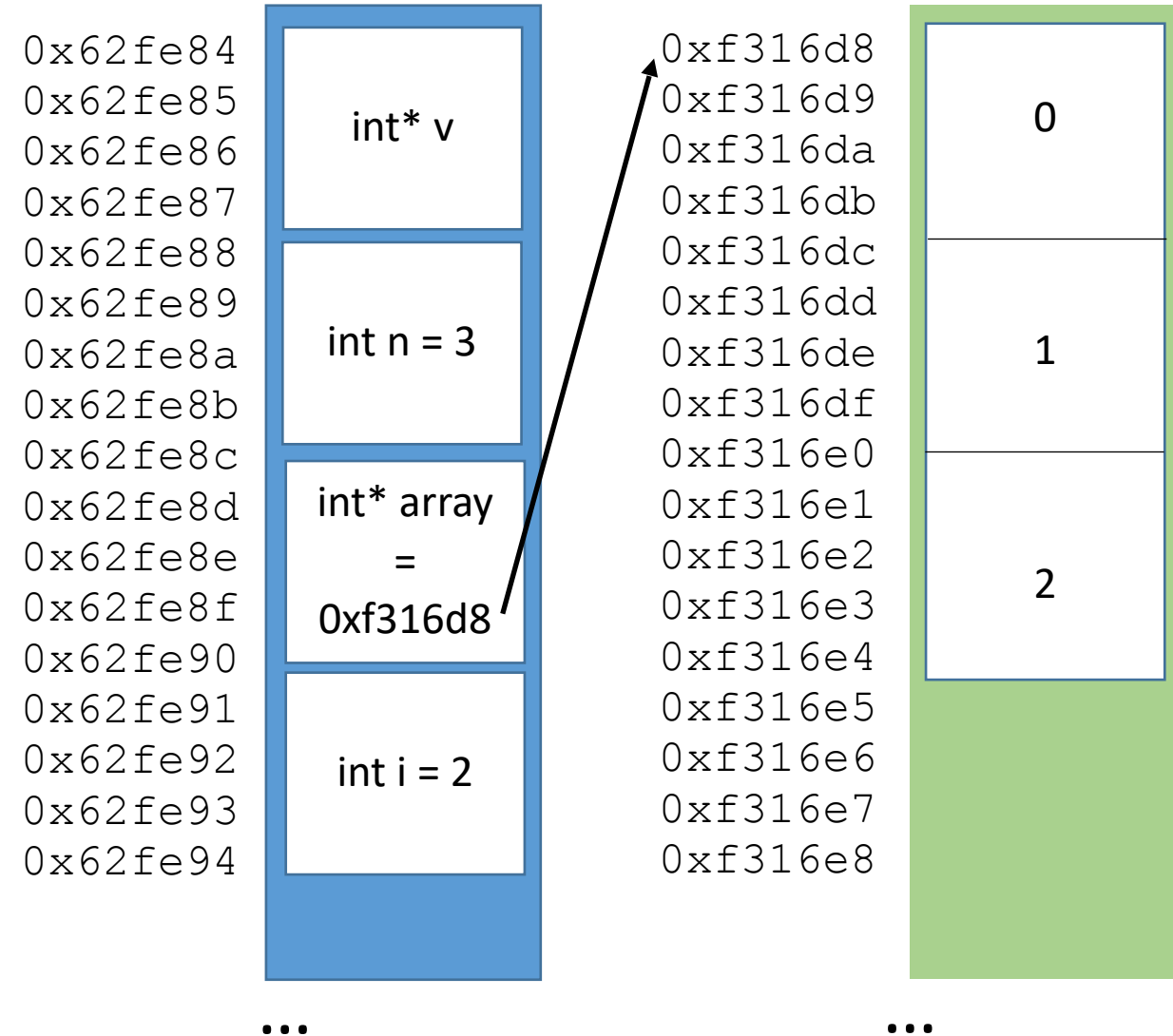


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

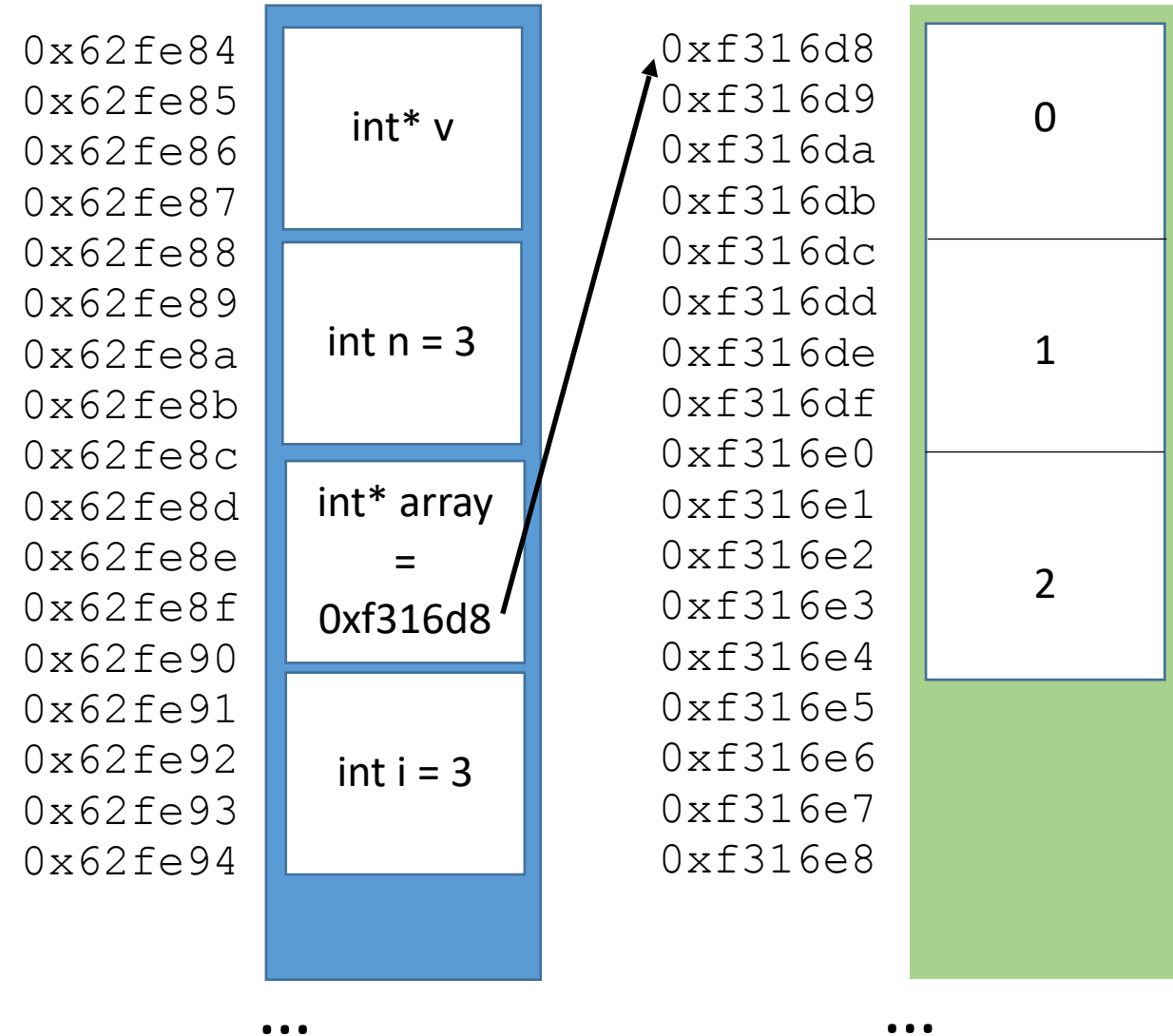


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```



Voorbeeld: new in functie

```
int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}
```

wat gaat er nu
verdwijnen?

```
int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```

0x62fe84
0x62fe85
0x62fe86
0x62fe87
0x62fe88
0x62fe89
0x62fe8a
0x62fe8b
0x62fe8c
0x62fe8d
0x62fe8e
0x62fe8f
0x62fe90
0x62fe91
0x62fe92
0x62fe93
0x62fe94

int* v

int n = 3

int* array
=
0xf316d8

int i = 3

0xf316d8
0xf316d9
0xf316da
0xf316db
0xf316dc
0xf316dd
0xf316de
0xf316df
0xf316e0
0xf316e1
0xf316e2
0xf316e3
0xf316e4
0xf316e5
0xf316e6
0xf316e7
0xf316e8

0

1

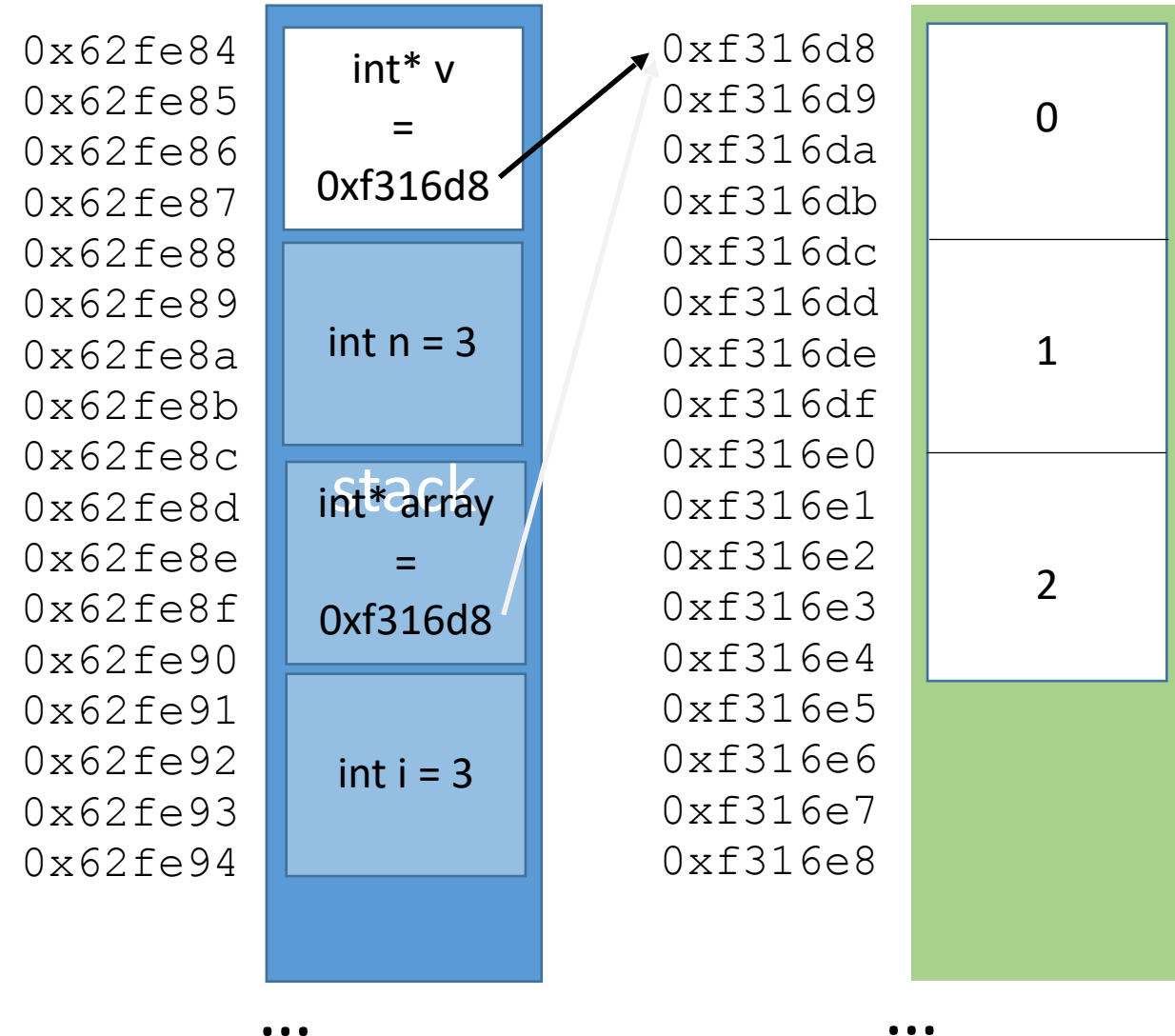
2

Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

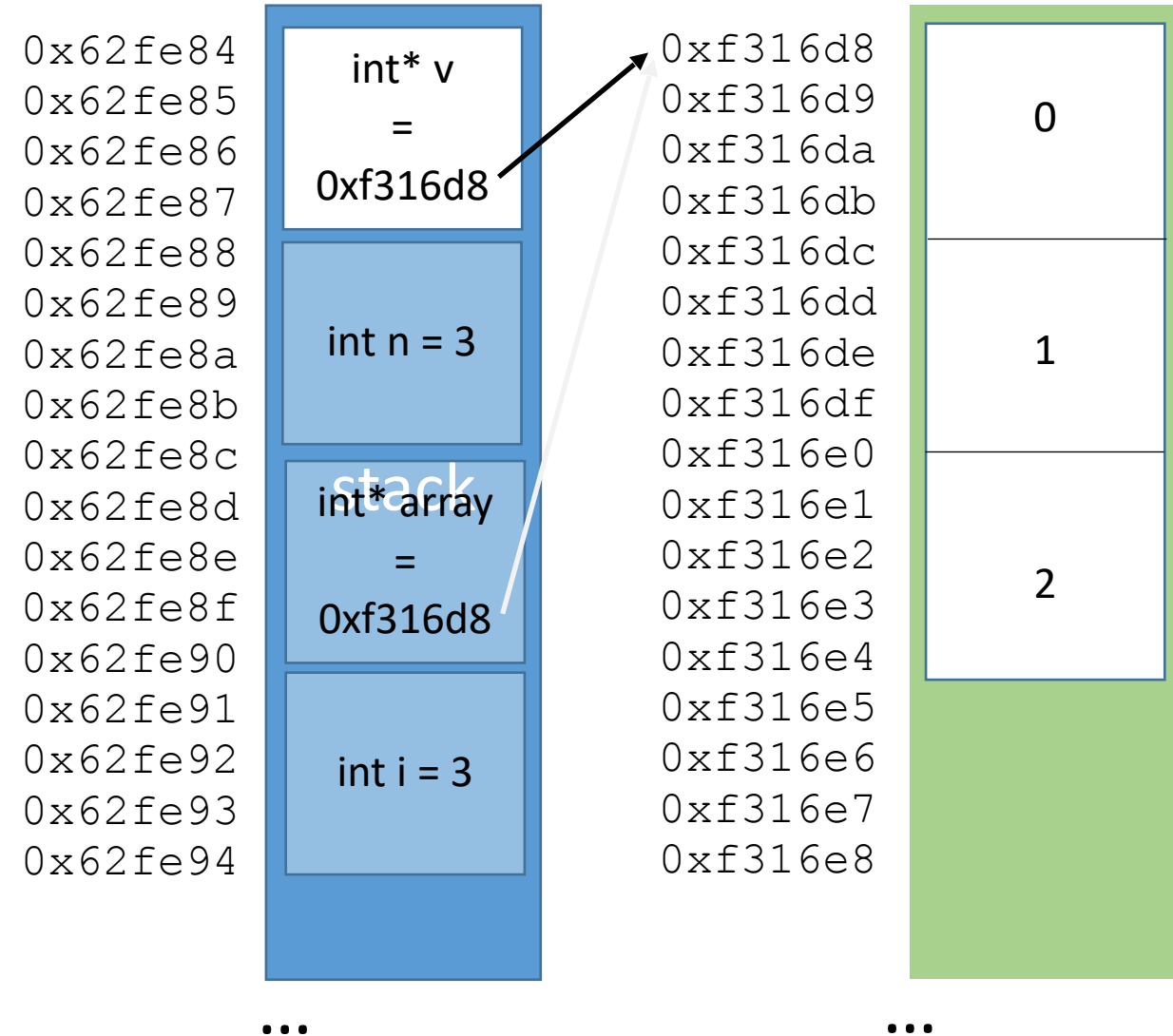


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

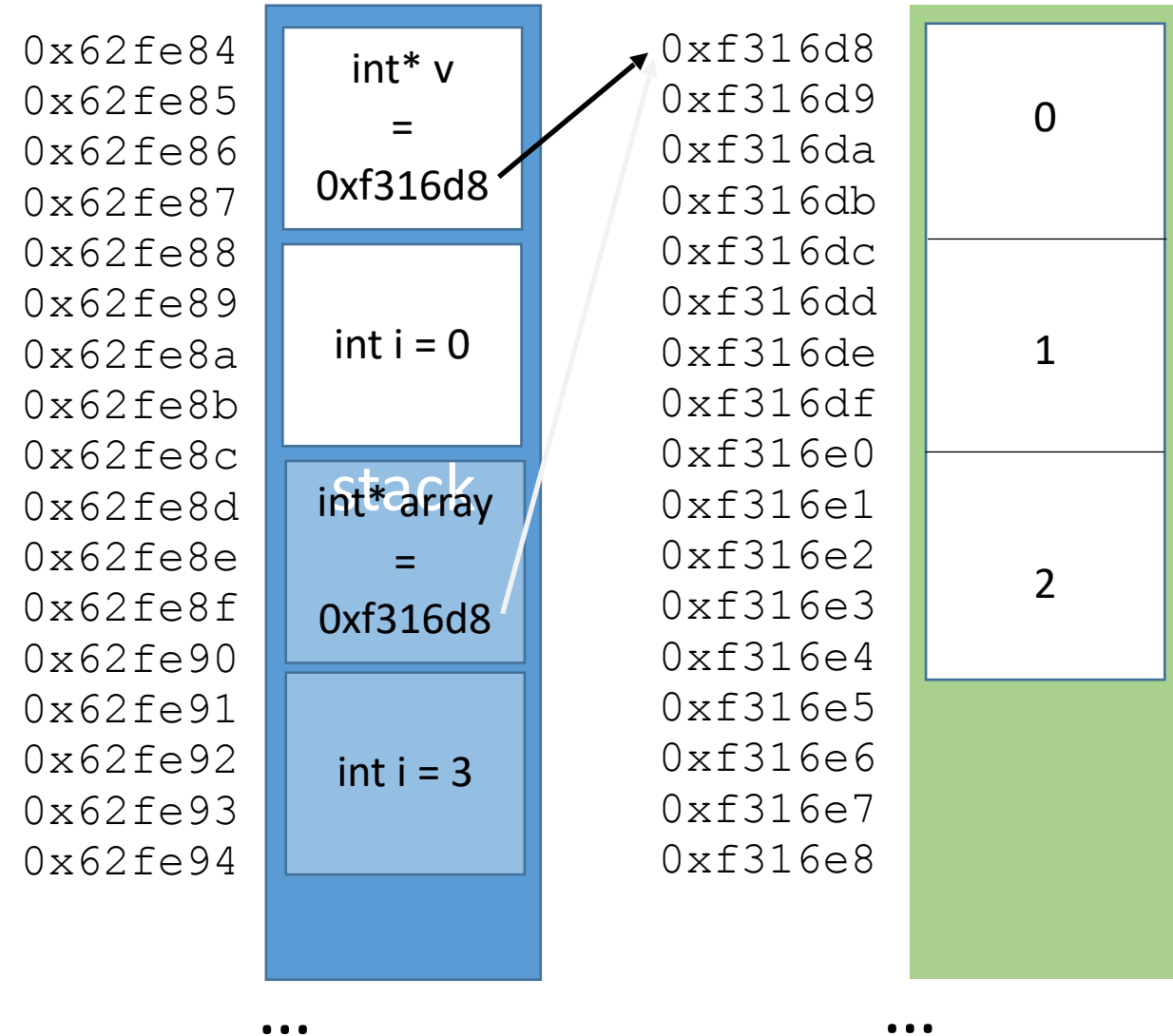
int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```



Voorbeeld: new in functie

```
int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

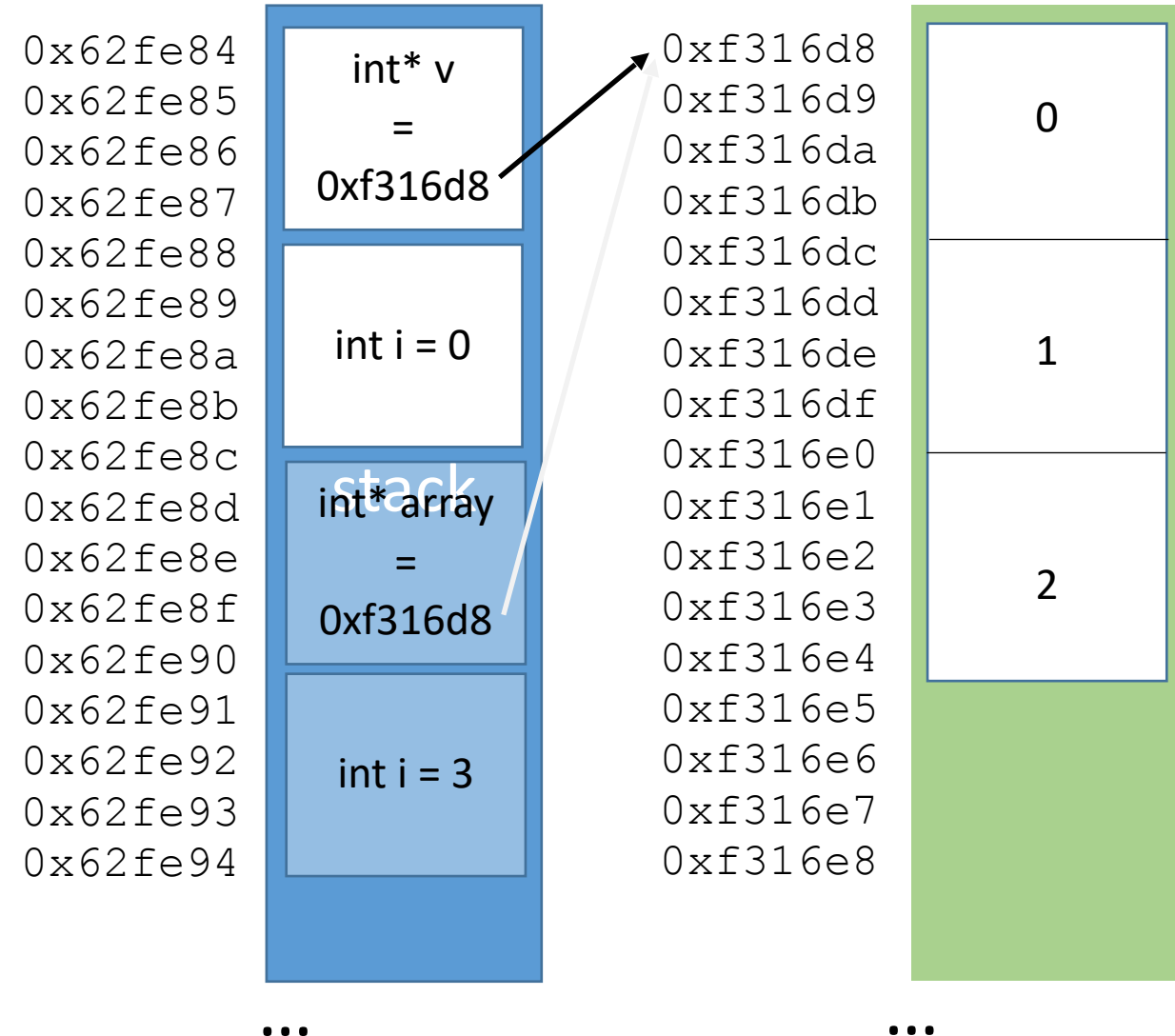
int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```



Voorbeeld: new in functie

```
int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

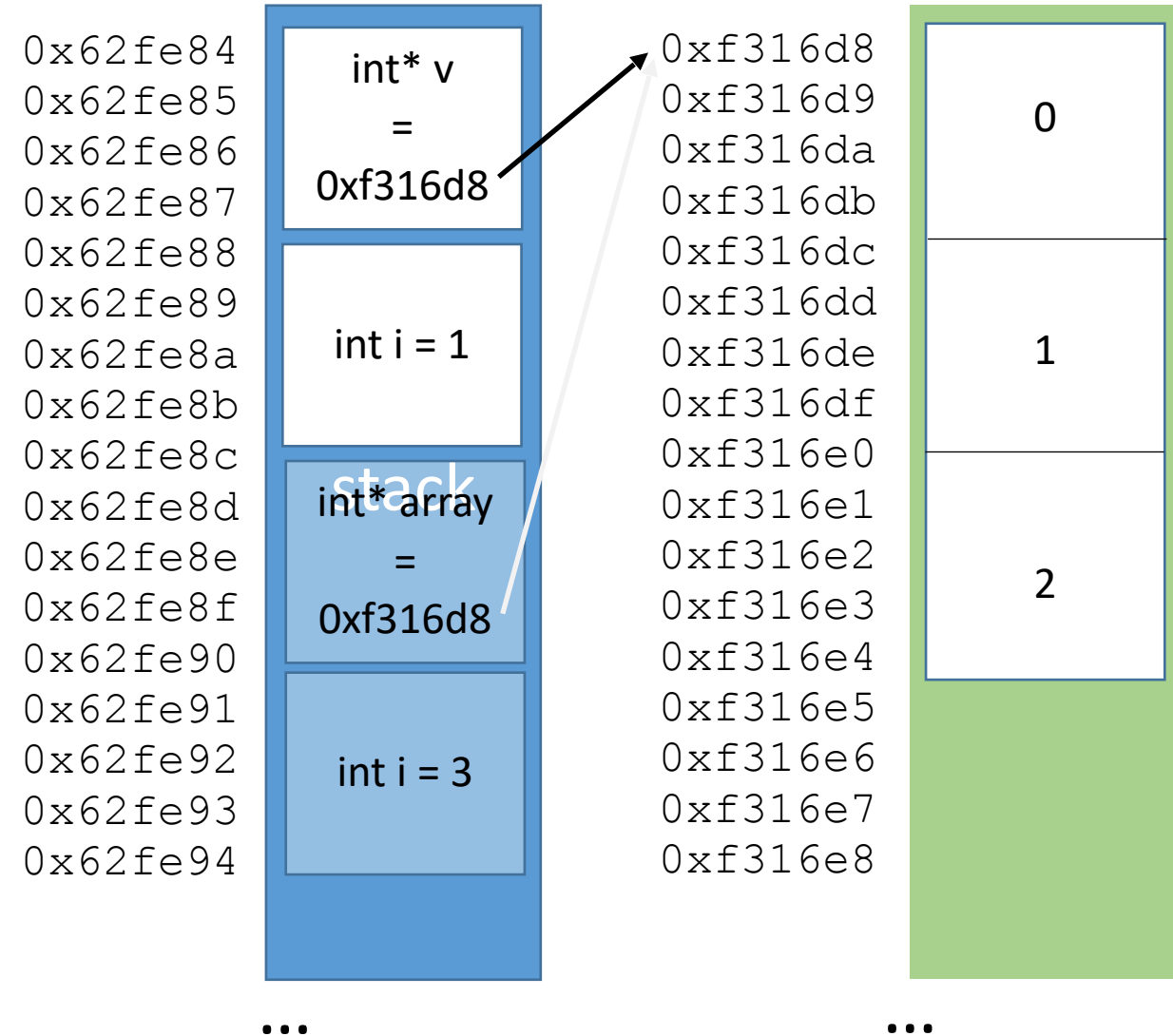
int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```



Voorbeeld: new in functie

```
int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

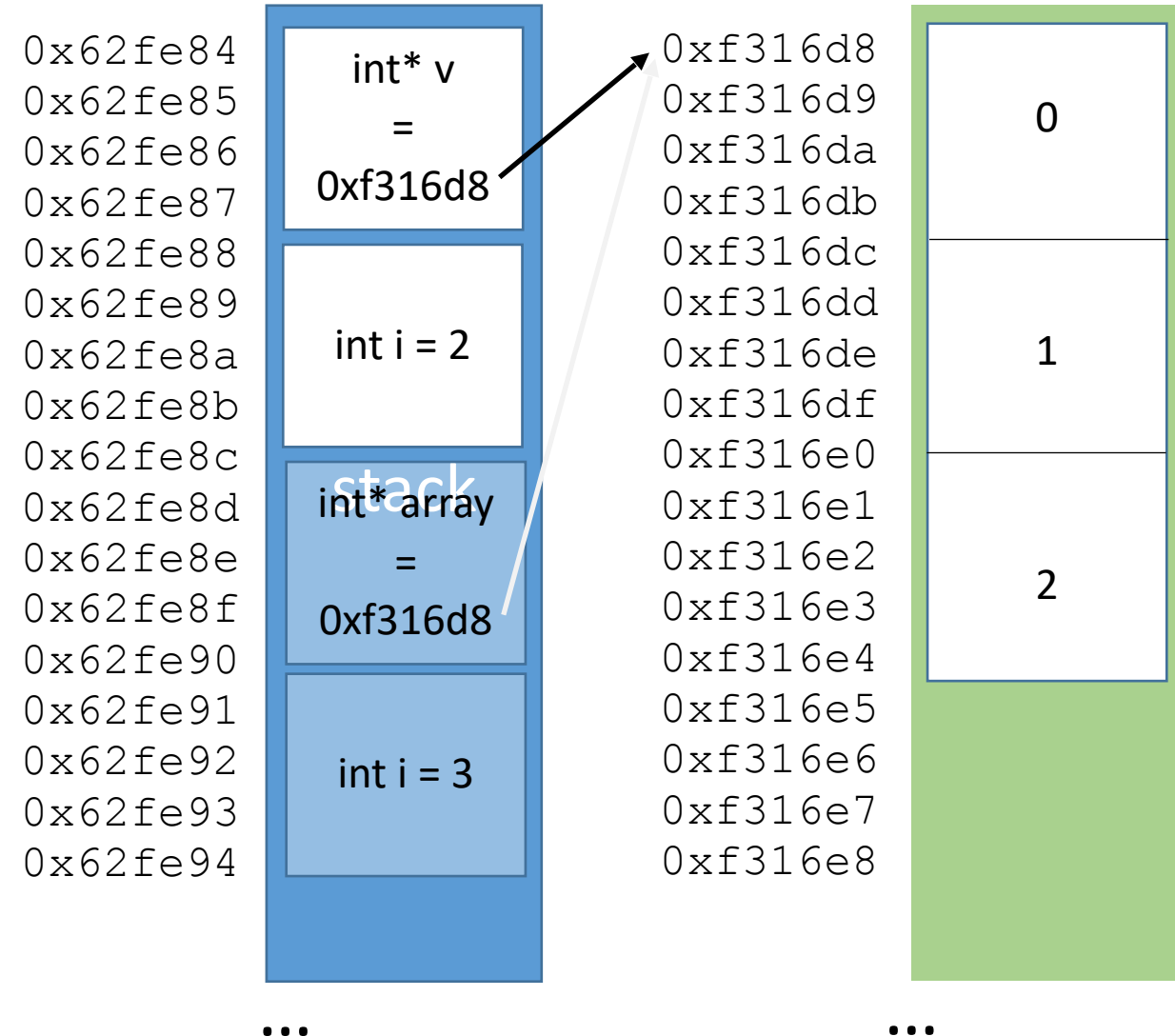
int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```



Voorbeeld: new in functie

```
int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

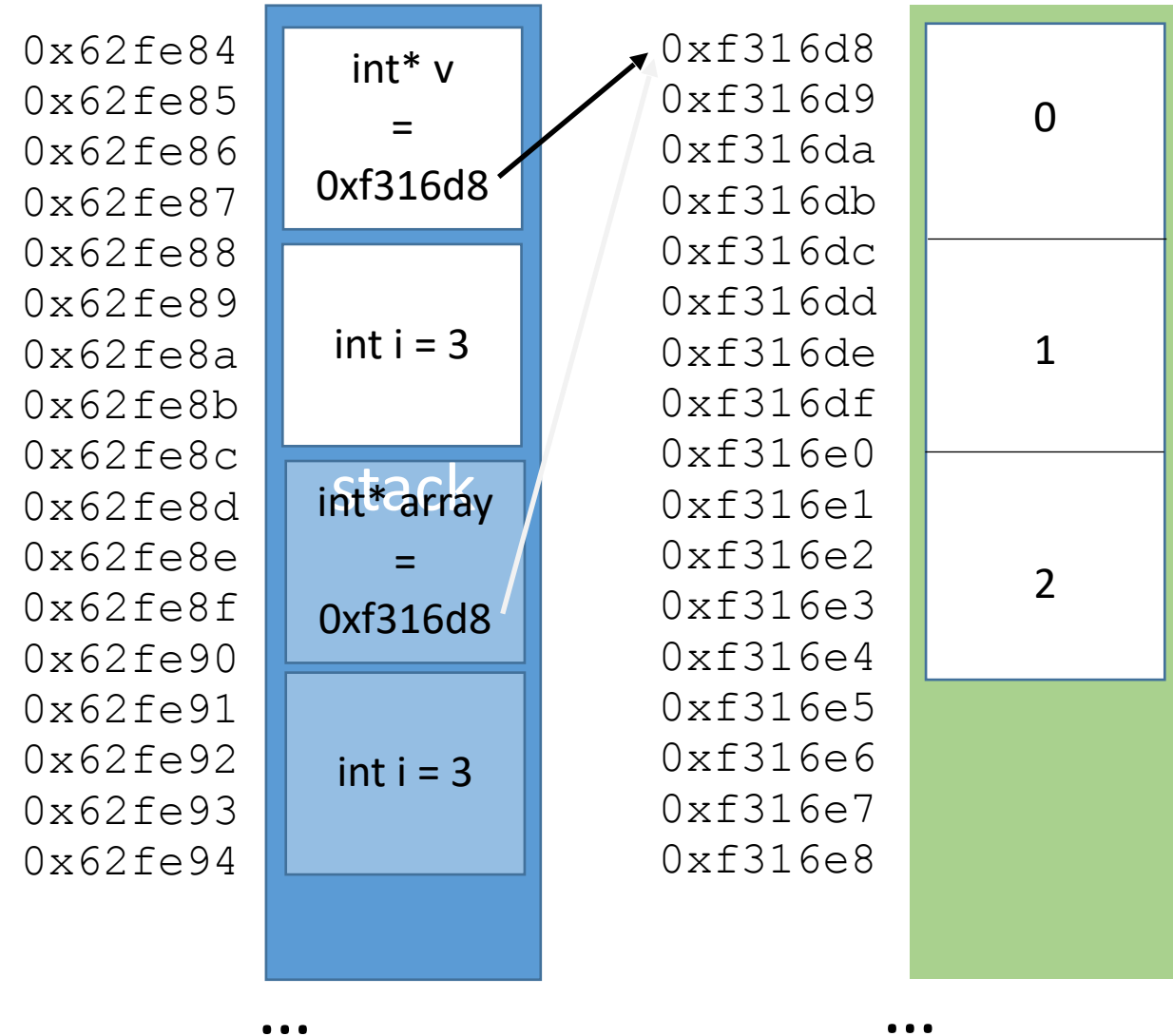
int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```



Voorbeeld: new in functie

```
int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

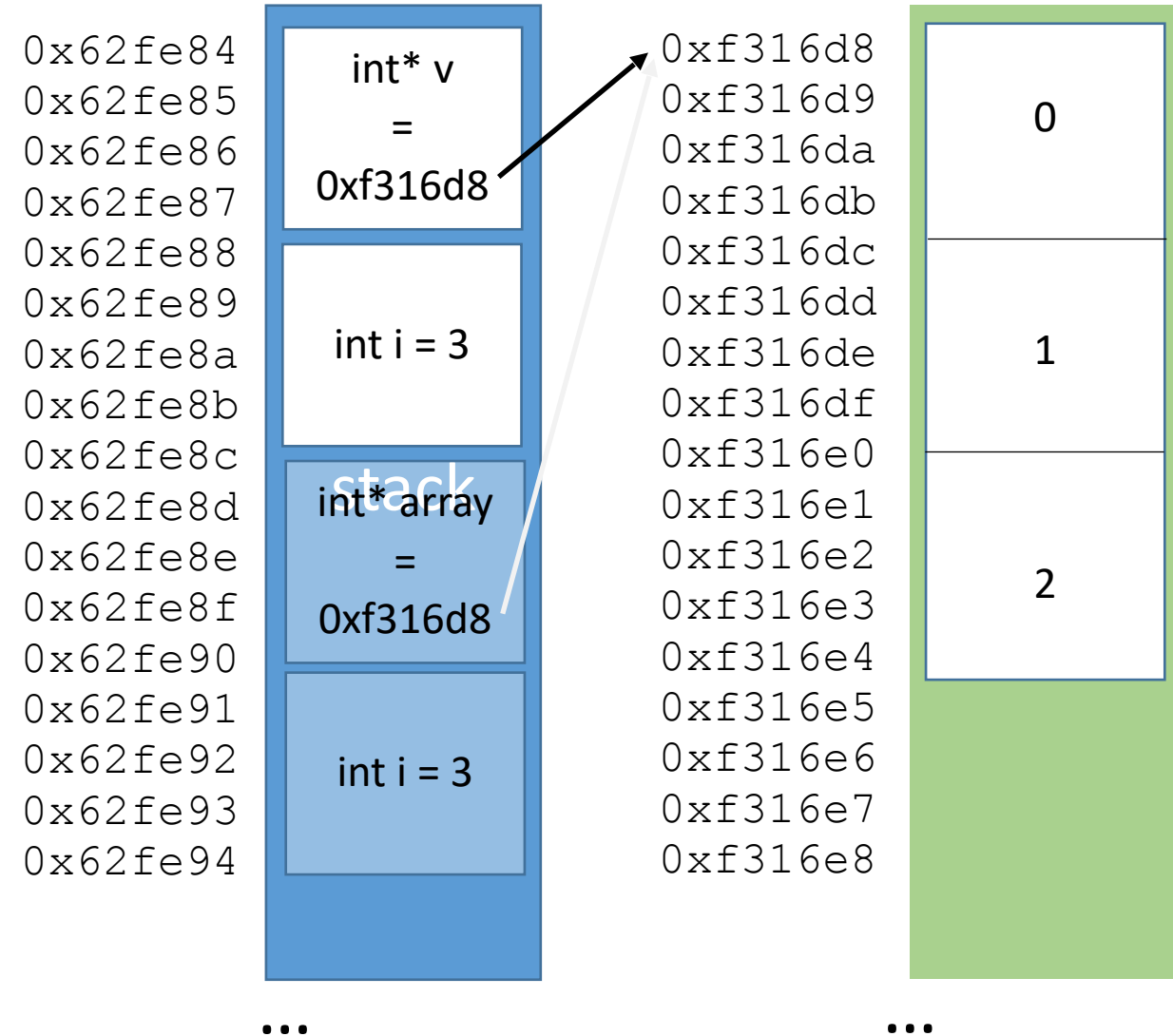
int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```



Voorbeeld: new in functie

```
int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```

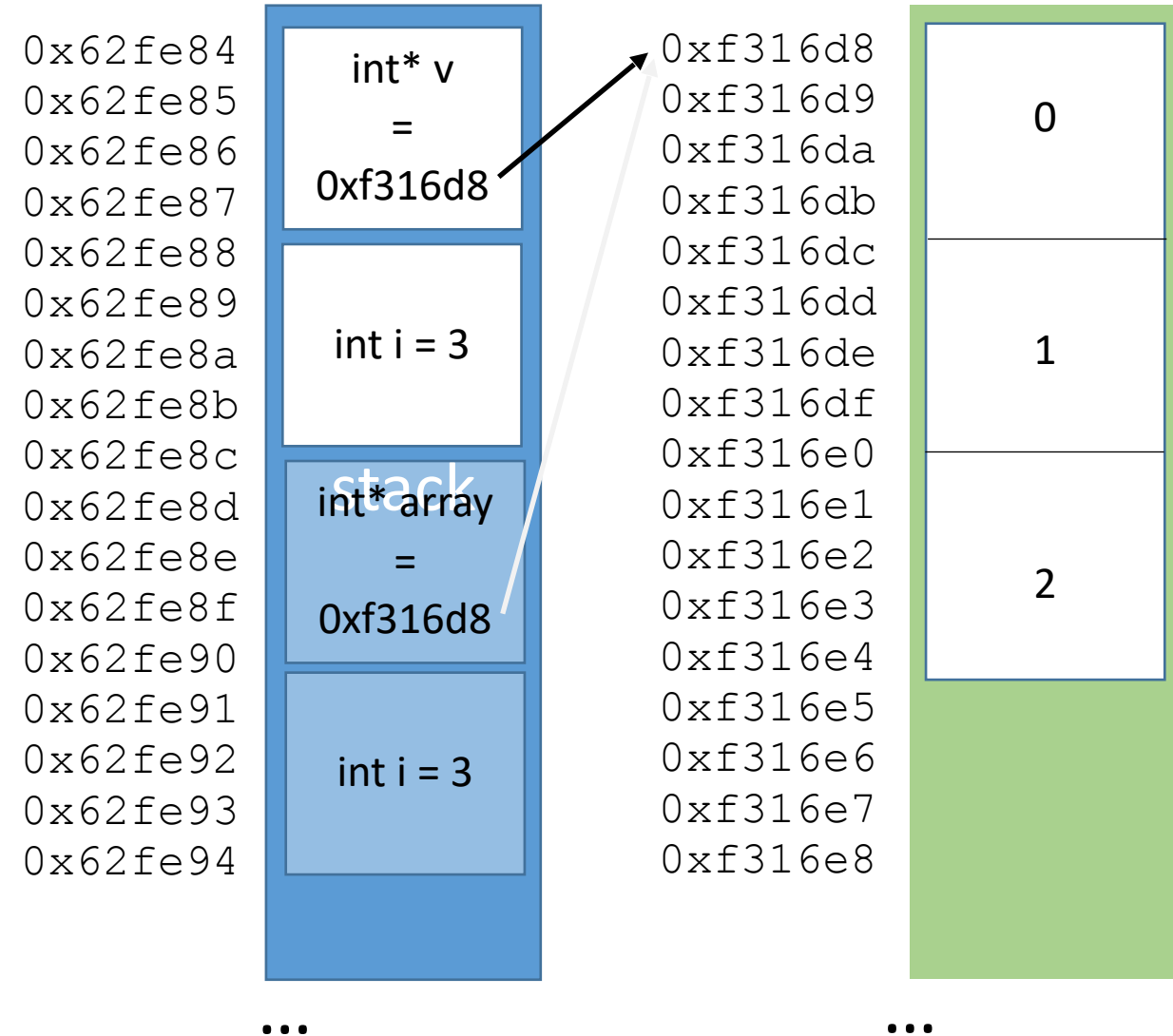


Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```



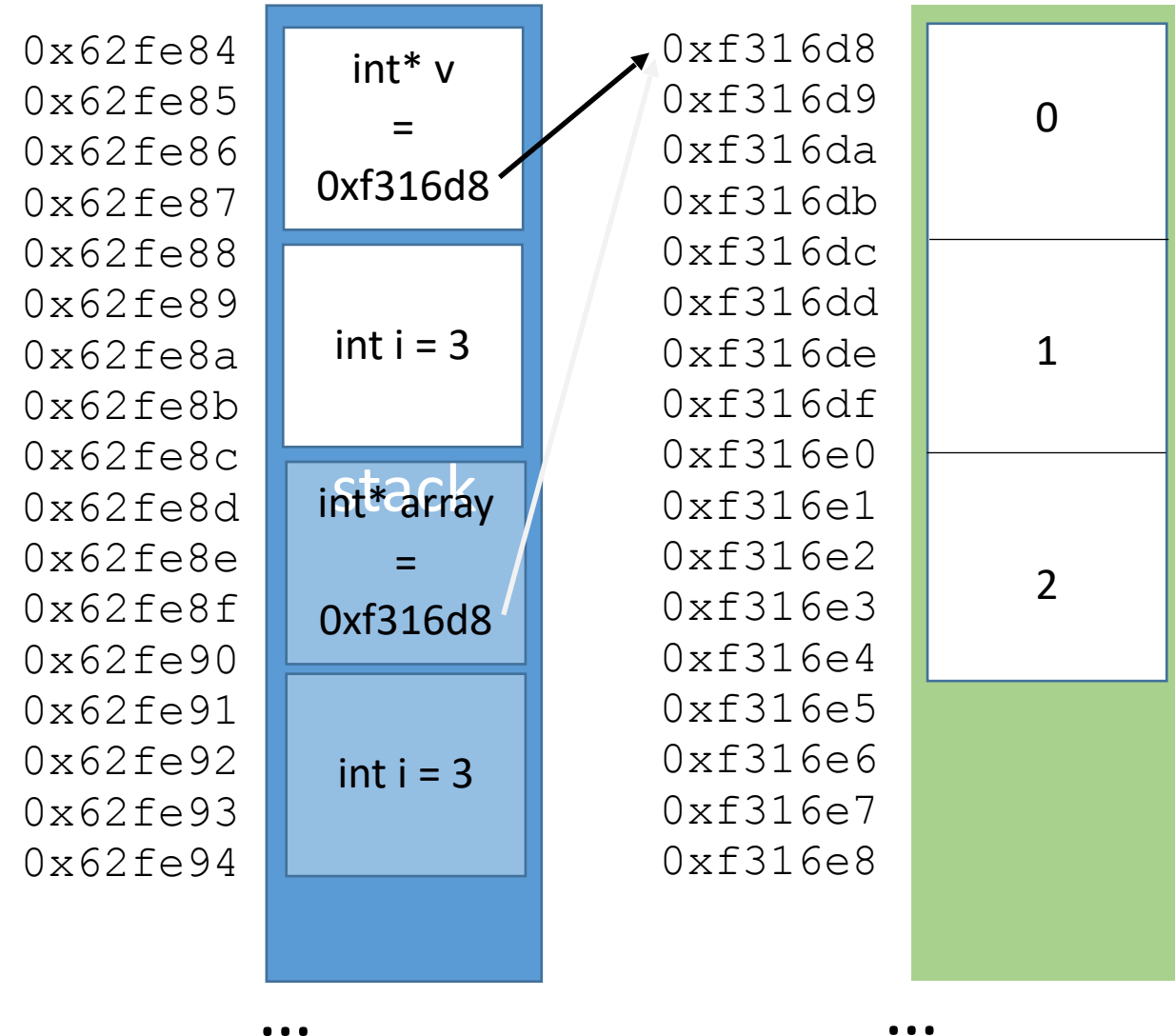
Voorbeeld: new in functie

```

int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
  
```

wat gaat er nu
verdwijnen?



Voorbeeld: new in functie

```
int* range(int n) {
    int* array=new int[n];
    for (int i=0;i<n;i++) {
        array[i]=i;
    }
    return array;
}

int main() {
    int* v=range(3);
    for (int i=0;i<3;i++) {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```

```
0x62fe84
0x62fe85
0x62fe86
0x62fe87
0x62fe88
0x62fe89
0x62fe8a
0x62fe8b
0x62fe8c
0x62fe8d
0x62fe8e
0x62fe8f
0x62fe90
0x62fe91
0x62fe92
0x62fe93
0x62fe94
```

```
int* v
=
0xf316d8
```

```
int i = 3
```

```
int* array
    =
    0xf316d8
```

```
int i = 3
```

• • •

0xf316d8
0xf316d9
0xf316da
0xf316db
0xf316dc
0xf316dd
0xf316de
0xf316df
0xf316e0
0xf316e1
0xf316e2
0xf316e3
0xf316e4
0xf316e5

0

1

2

memory
leak

Verboden!

...

Voorbeeld: new in functie

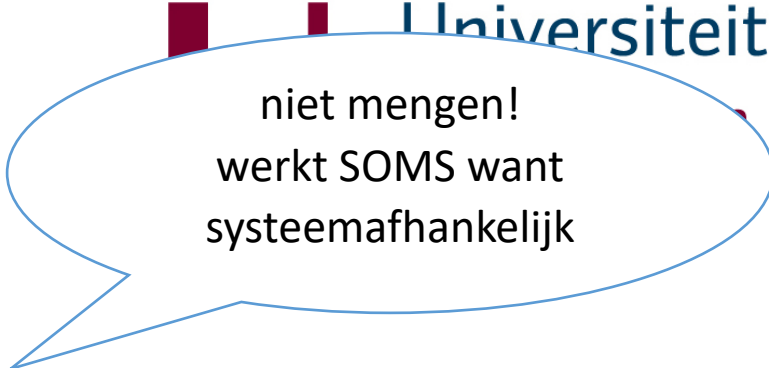
```
int* range(int n) {  
    int* array=new int[n];  
    for (int i=0;i<n;i++) {  
        array[i]=i;  
    }  
    return array;  
}  
  
int main() {  
    int* v=range(3);  
    for (int i=0;i<3;i++) {  
        cout << v[i] << " ";  
    }  
    cout << endl;  
    return 0;  
}
```

- Value die wordt teruggegeven = de *waarde* van de pointer ; een geheugenadres
- Pointer is een “lijn” waarmee de data kan “opgevist” worden
- *Enkel data op de stack* verdwijnt als de functie afgelopen is.

Nadeel van new en new[]?

- Via new kunnen we complexe objecten aanmaken die *blijven bestaan*
- Objecten die we via new aanmaken blijven bestaan, ook als we ze niet meer nodig hebben!
 - Geheugen loopt vol
- Gelukkig kunnen we expliciet aangeven dat geheugen terug vrijgegeven moet worden

Dynamic Memory Allocation



niet mengen!
werkt SOMS want
systeemafhankelijk

- Het geheugen gereserveerd met `new` blijft gereserveerd tot `delete`
- Het geheugen gereserveerd met `new[]` blijft gereserveerd tot `delete[]`
- Als we `delete` of `delete[]` vergeten, krijgen we een “memory leak” = gereserveerd geheugen dat niet gebruikt wordt
 - Memory leaks manifesteren zich pas na lange tijd; geheugen vult zich langzaam met niet-vrijgegeven gereserveerde blokken.
 - Het programma wordt langzaam en gebruikt veel geheugen
 - Op een bepaald moment is het geheugen vol → crash!
 - Memory leaks zijn dan ook erg moeilijk te vinden.

Voorbeeld delete en delete []

```
int i = 5;  
int* p = &i;
```

```
int* p2 = new int;  
int* p3 = new int[7];
```

```
delete p2;  
delete[] p3;
```

Zoek de fout

```
int* range(int n) {  
    int* array = new int[n];  
    for (int i = 0; i < n; i++) {  
        array[i] = i;  
    }  
    return array;  
}
```

```
int main() {  
    int* v = range(3);  
    for (int i = 0; i < 3; i++) {  
        cout << v[i] << " ";  
    }  
    cout << endl;  
    delete v;  
    return 0;  
}
```

Wat werd er dan eigenlijk
wel verwijderd?

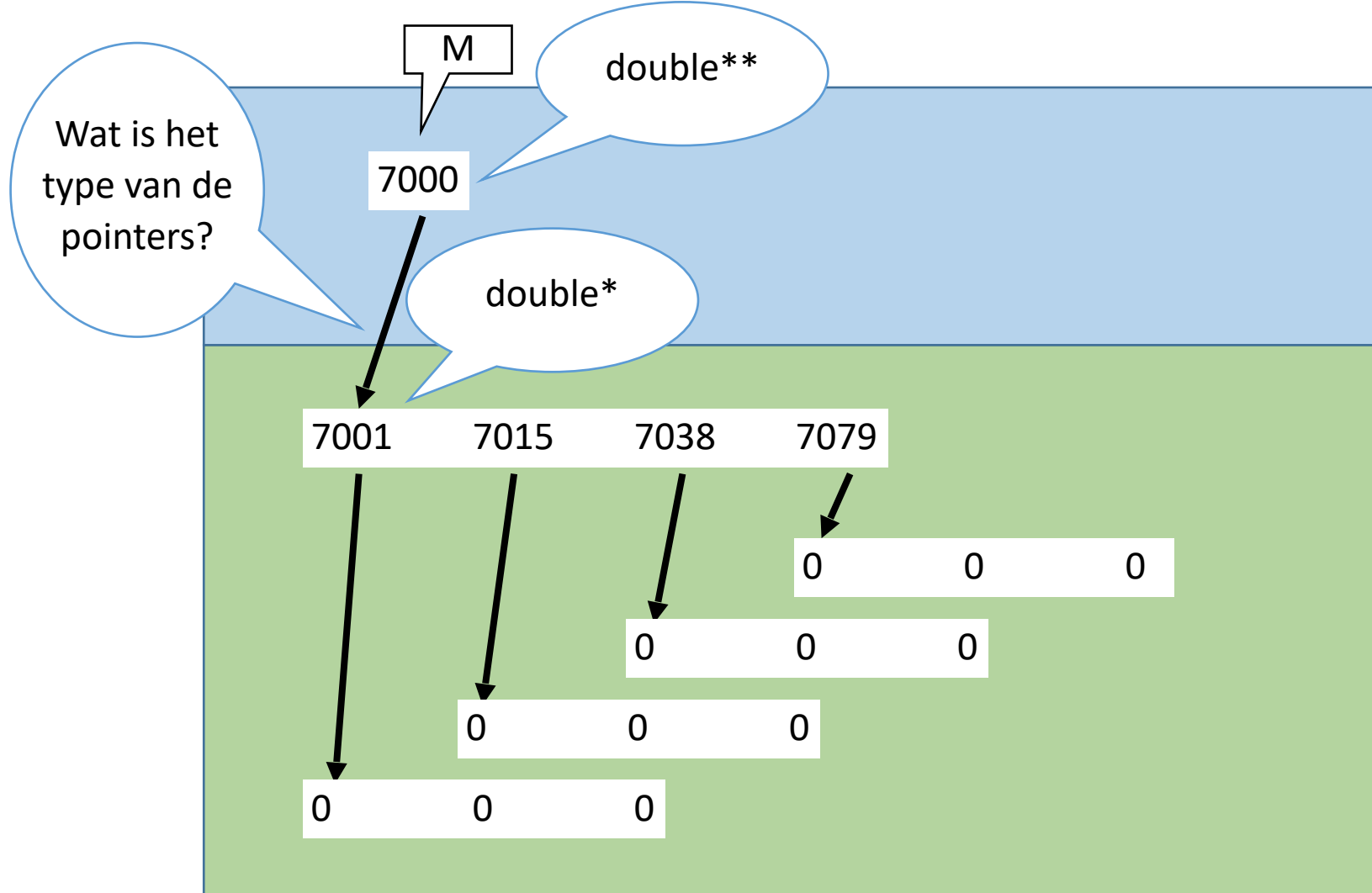
Voorbeeld: Matrix aanmaken

- Via new kunnen we complexe objecten aanmaken die
- Bijvoorbeeld een matrix:
 - Een cel in een matrix is een double
 - Een rij is een array van doubles (lengte = #cols)
 - Een matrix is een array van rijen (lengte = #rows)
- Deze structuur kunnen we in een functie aanmaken in de vrije ruimte, en dan een pointer naar de matrix teruggeven

denk eraan: een array is hetzelfde als een pointer naar het eerste element

```
double  
double* rij = double[cols]  
double** matrix = double*[rows]
```

Matrix maken

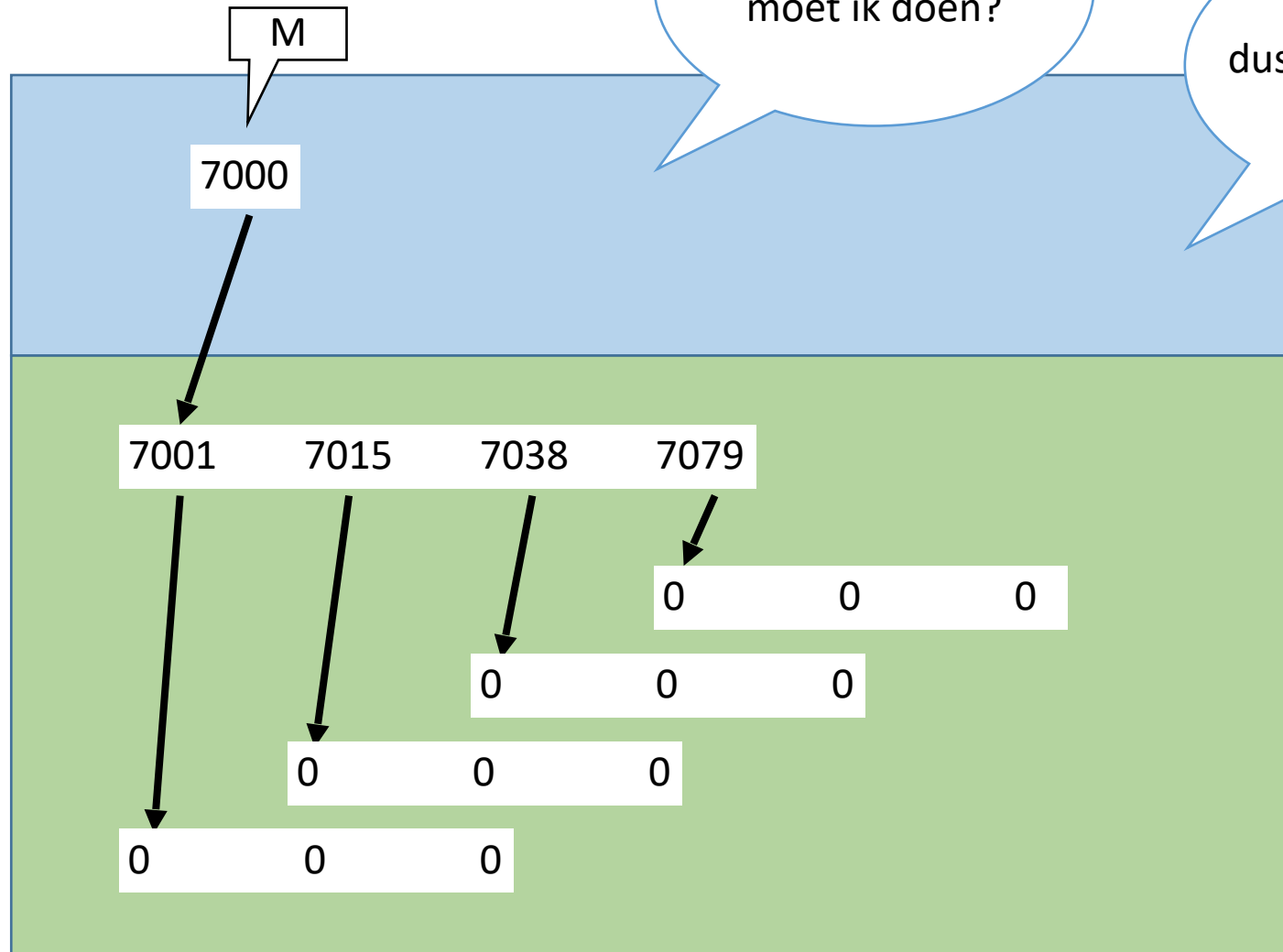


Nut van new en new[]?

- Via new kunnen we complexe objecten aanmaken die *blijven bestaan*

```
double** maakmatrix(int rows, int cols) {  
    double** M = new double*[rows];  
    for (int i = 0; i < rows; i++) {  
        M[i] = new double[cols];  
        for (int j = 0; j < cols; j++) {  
            M[i][j] = 0.0;  
        }  
    }  
    return M;  
}
```

Matrix deleten



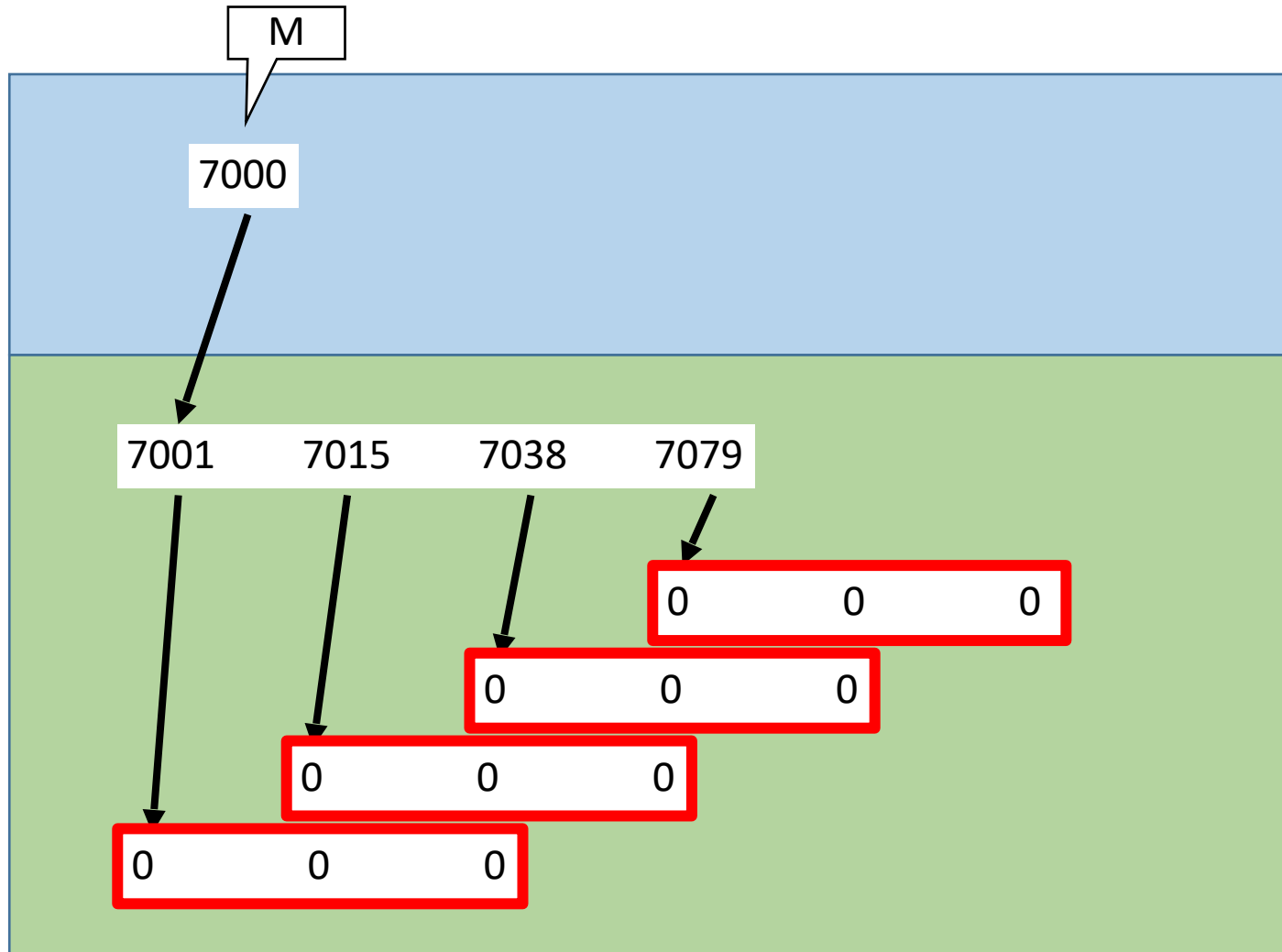
hoeveel deletes
moet ik doen?

dus eerst delete M?

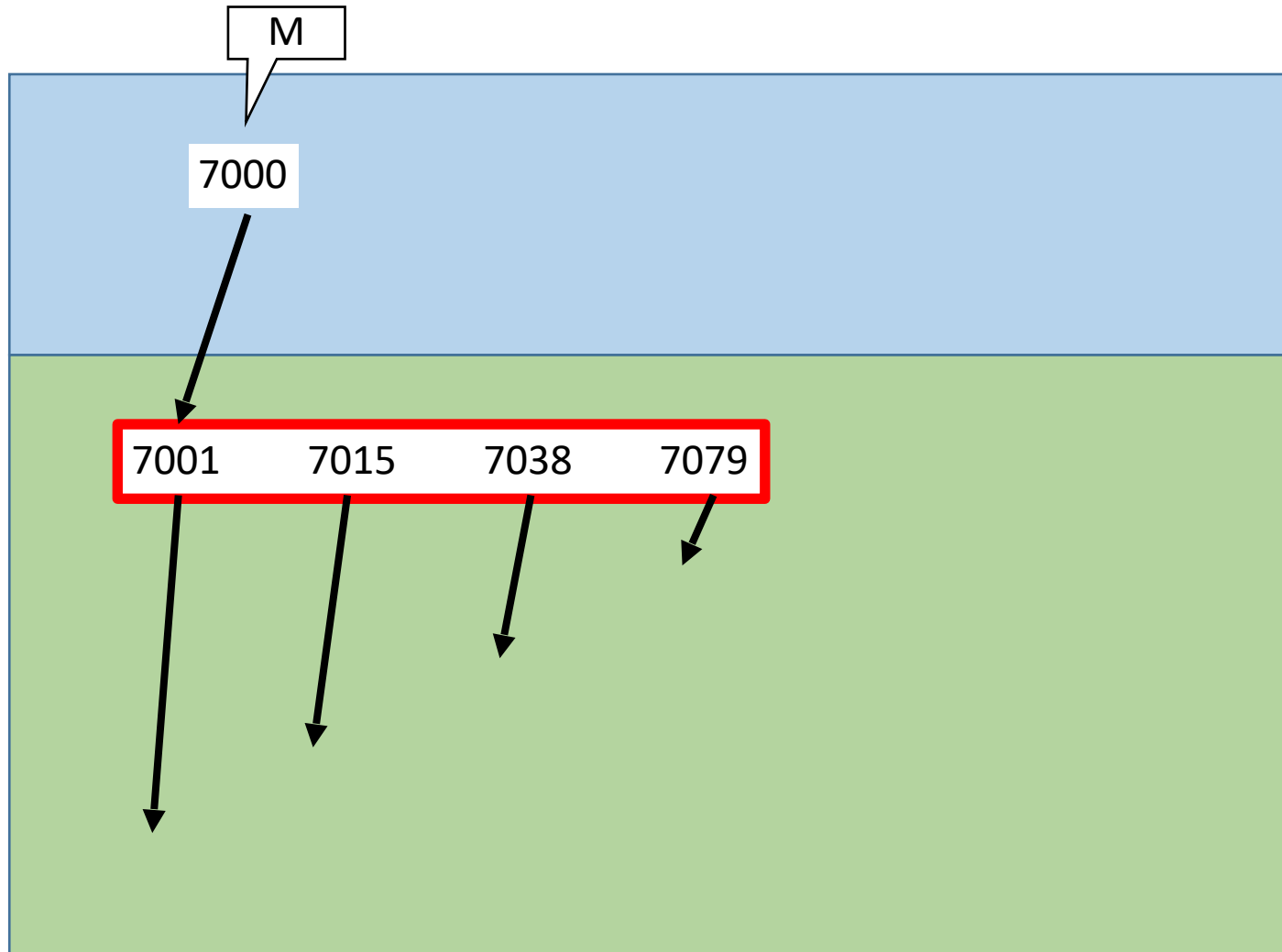
oh nee, dat is
delete[] M!

Nope! Dan kan ik
de andere niet meer
deleten!

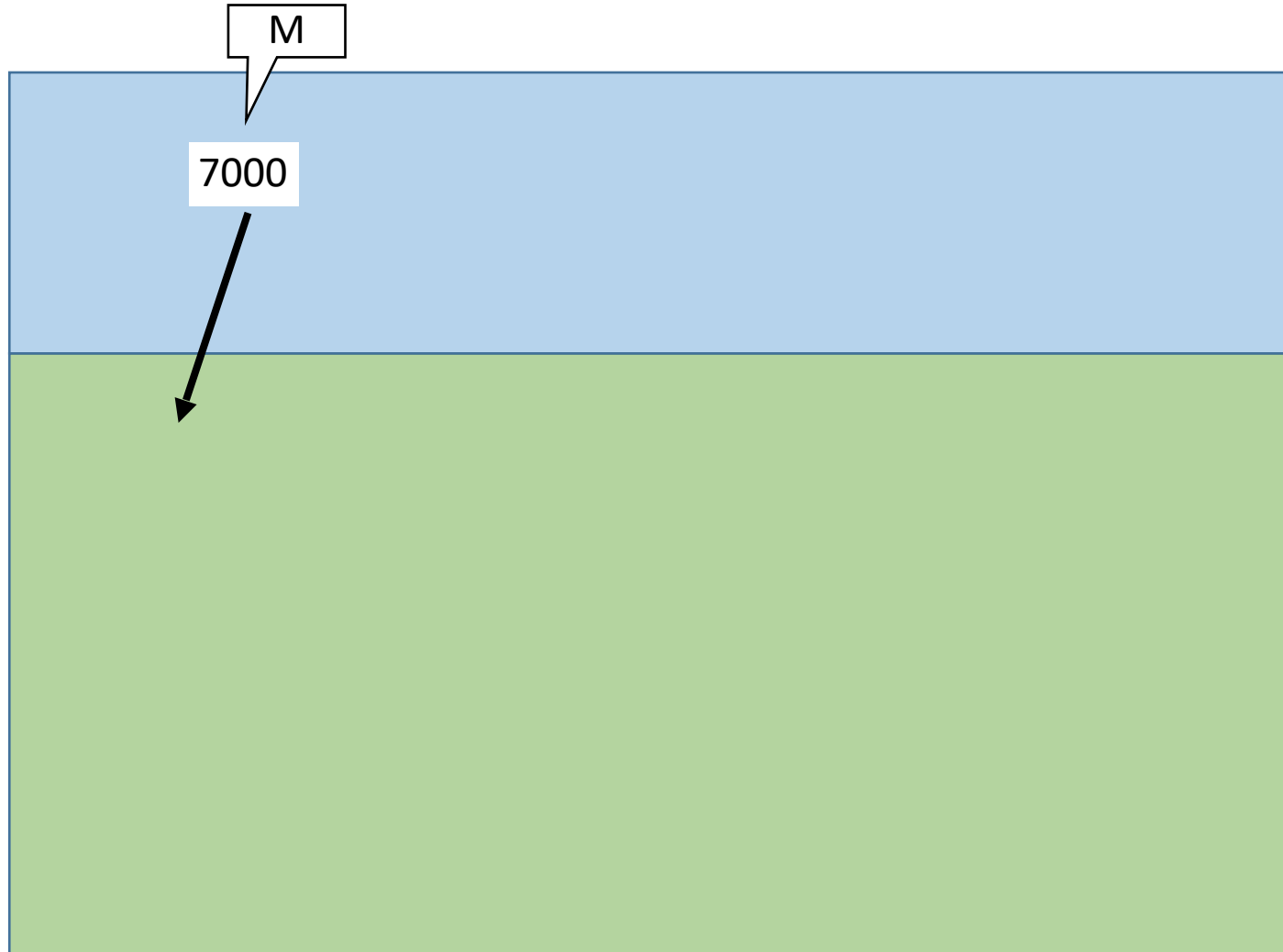
Matrix deleten



Matrix deleten



Matrix deleten



Matrix deleten

```
void delete_matrix(double** M, int rows) {  
    for (int i = 0; i < rows; i++) {  
        delete[] M[i];  
    }  
    delete[] M;  
}
```

Struct

- Groeperen van data

zoals class in Python

```
struct Kaart {  
    int kleur;  
    int nummer;
```

geen return type!

```
    Kaart(int k, int n) {  
        kleur = k;  
        nummer = n;  
    }
```

geen self nodig

```
};
```

```
int main() {  
    Kaart k(5, 3);  
    cout << k.kleur << endl;  
}
```

net zoals in Python

Struct en new

```
struct Speler {  
    string naam;  
    int enkel;  
    int dubbel;  
    int gemengd;
```

```
    Speler(string n, int ke, int kd, int kg) {  
        naam = n;  
        enkel = ke;  
        dubbel = kd;  
        gemengd = kg;  
    }  
};
```

pointers zodat als naam
aangepast van een speler dat in
de spelers vector ook
aangepast is

```
struct Ploeg {  
    vector<Speler*> spelers;  
    string naam;  
    bool gemengd;
```

```
    Ploeg(string pnaam, bool g) {  
        naam = pnaam;  
        gemengd = g;  
    }  
};
```

```
Speler* speler1 = new Speler("Tom", 2, 2, 3);
Speler* speler2 = new Speler("Ayman", 2, 3, 3);
Speler* speler3 = new Speler("Marie", 2, 2, 2);
Speler* speler4 = new Speler("Thomas", 2, 2, 2);
Speler* speler5 = new Speler("Mo", 2, 2, 2);
Speler* speler6 = new Speler("Noran", 2, 3, 3);
Ploeg p1("2H", false);
Ploeg p2("3G", true);
```

```
p1.spelers.push_back(speler1);
p1.spelers.push_back(speler2);
p1.spelers.push_back(speler3);
p1.spelers.push_back(speler4);
```

```
p2.spelers.push_back(speler1);
p2.spelers.push_back(speler4);
p2.spelers.push_back(speler5);
p2.spelers.push_back(speler6);
```

spelers worden maar
1 keer aangemaakt en
bijgehouden in geheugen

Zowel in ploeg p1 als in ploeg
p2 staat deze speler nu met
de juiste gegevens

79 speler1->enkel = 4;

Samenvattend

- new <type> = pointer naar geheugenplaats die <type> kan bevatten
 - Niet op stack, maar in “vrije ruimte”
- wordt enkel verwijderd door delete / delete[]
- delete vergeten? = *memory leak*
- struct om variabelen te groeperen (zoals “class” in Python)
 - eigen samengesteld datatype genereren
- structs worden *shallow* gekopieerd bij = of call-by-value

Extra

- structs en call by value
- sizeof van containers

Struct en Call-By-Value

- Met struct kunnen we nieuwe datatypes maken door bestaande datatypes te combineren

```
struct staticList {  
    int* v;  
    int size;  
};
```

```
staticList(int n) {  
    v=new int[n];  
    size=n;  
}
```

Constructor; wordt uitgevoerd
telkens een nieuwe staticList wordt
gemaakt.

Struct en Call-By-Value

```
struct staticList {  
    int* v;  
    int size;  
  
    staticList(int n) {  
        v=new int[n];  
        size=n;  
    }  
};
```

```
staticList reverse(staticList s) {  
    staticList result(s.size);  
  
    for (int i=0;i<s.size;i++) {  
        result.v[i]=s.v[s.size-1-i];  
    }  
  
    return result;  
}  
  
int main() {  
    staticList s=staticList(2);  
    s.v[0]=0;s.v[1]=1;  
    staticList s2=reverse(s);  
    return 0;  
}
```

Struct en Call-By-Value

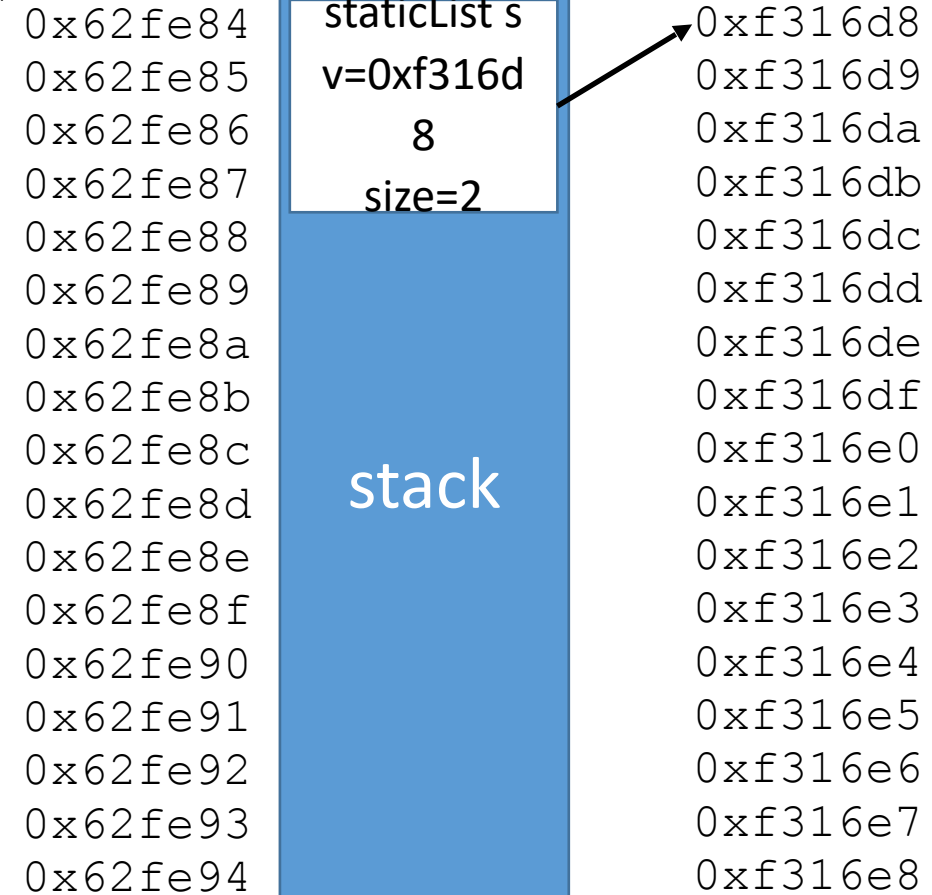
```
staticList reverse(staticList s) {
    staticList result(s.size);

    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

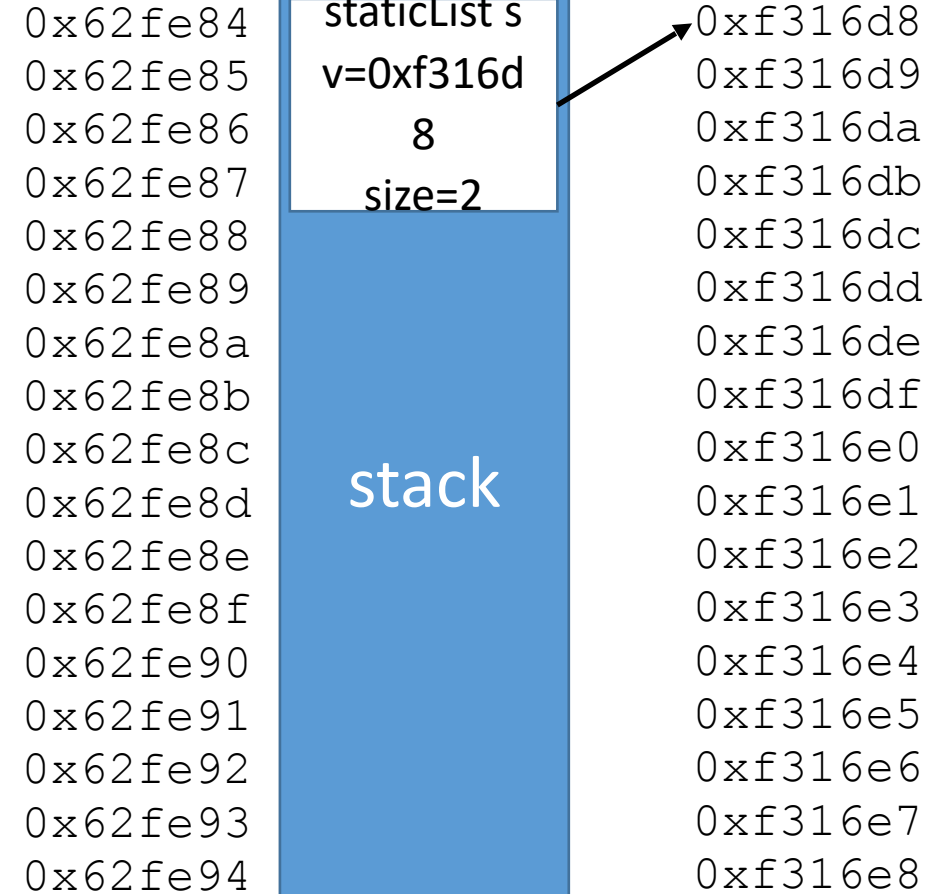
```
staticList reverse(staticList s) {
    staticList result(s.size);

    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

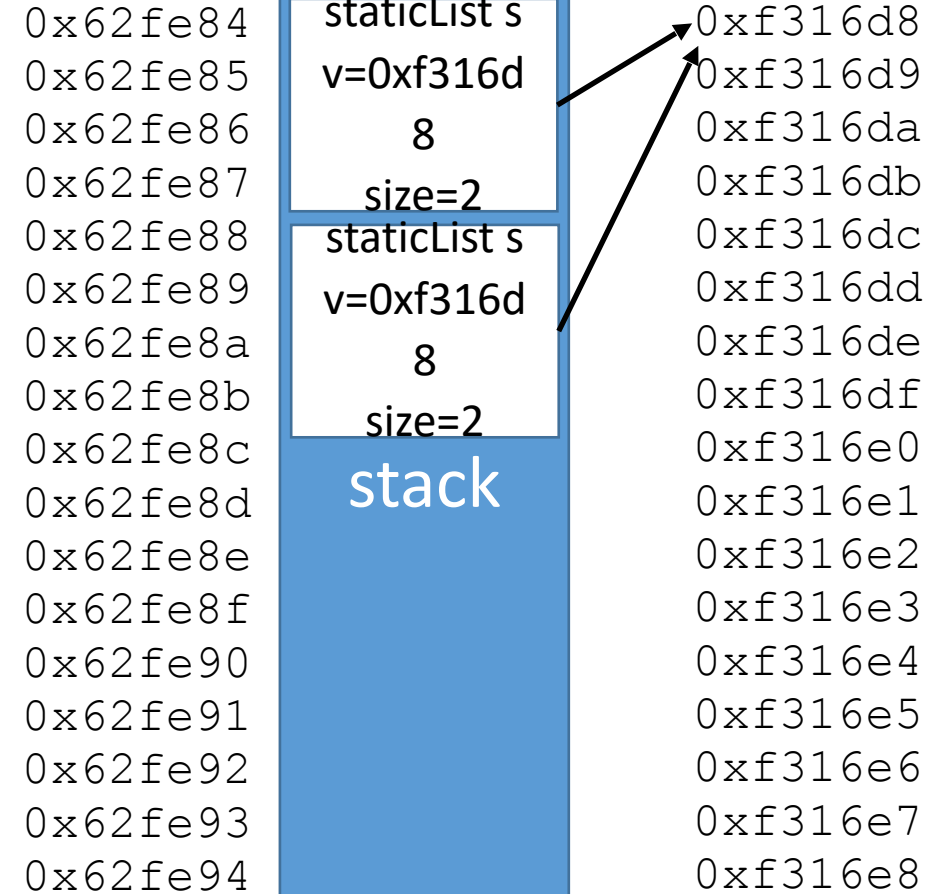
```
staticList reverse(staticList s) {
    staticList result(s.size);

    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

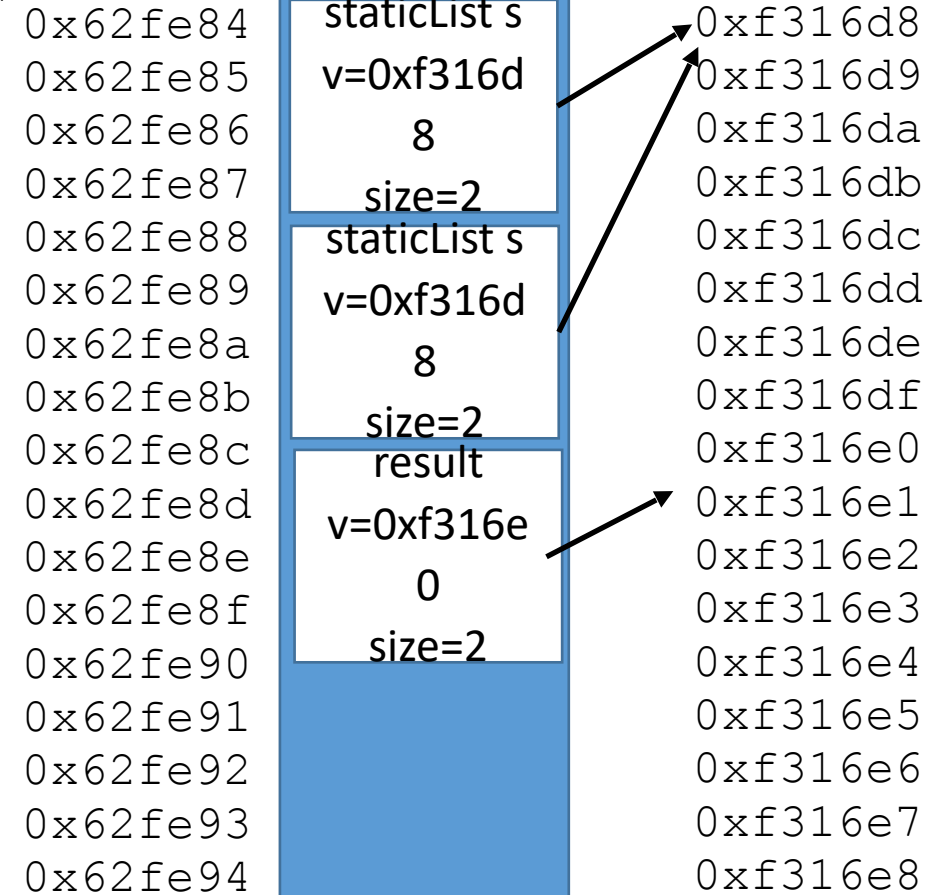
```
staticList reverse(staticList s) {
    staticList result(s.size);

    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

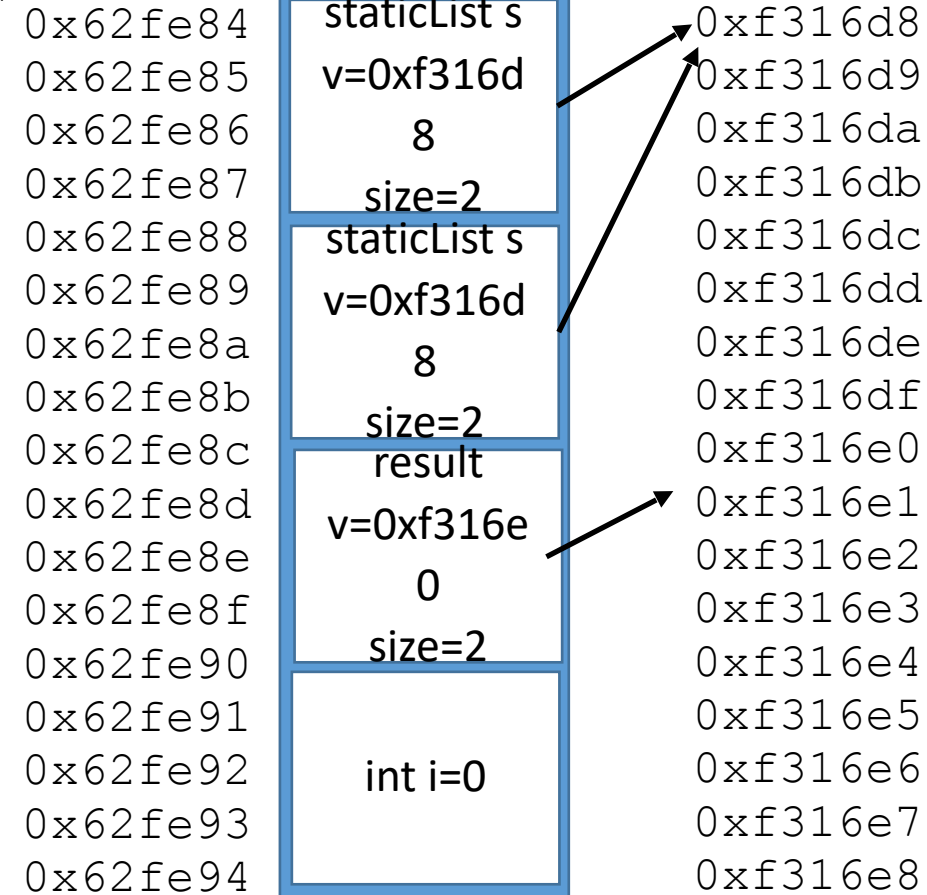
```
staticList reverse(staticList s) {
    staticList result(s.size);

    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

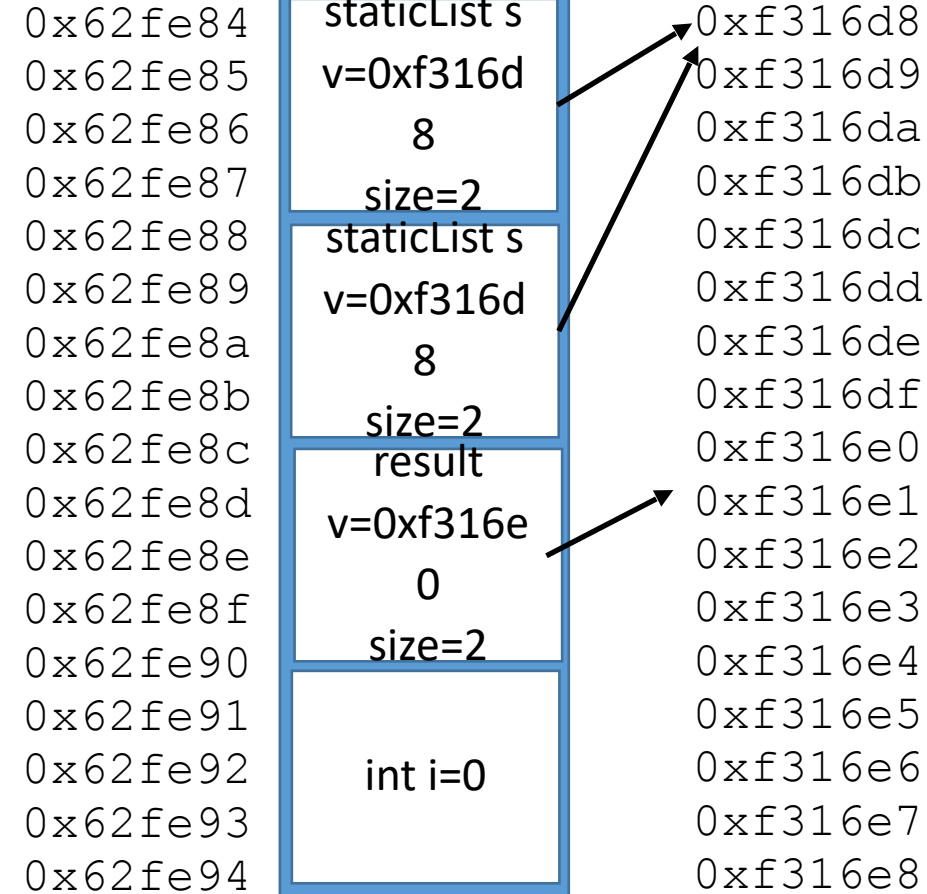
```
staticList reverse(staticList s) {
    staticList result(s.size);

    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

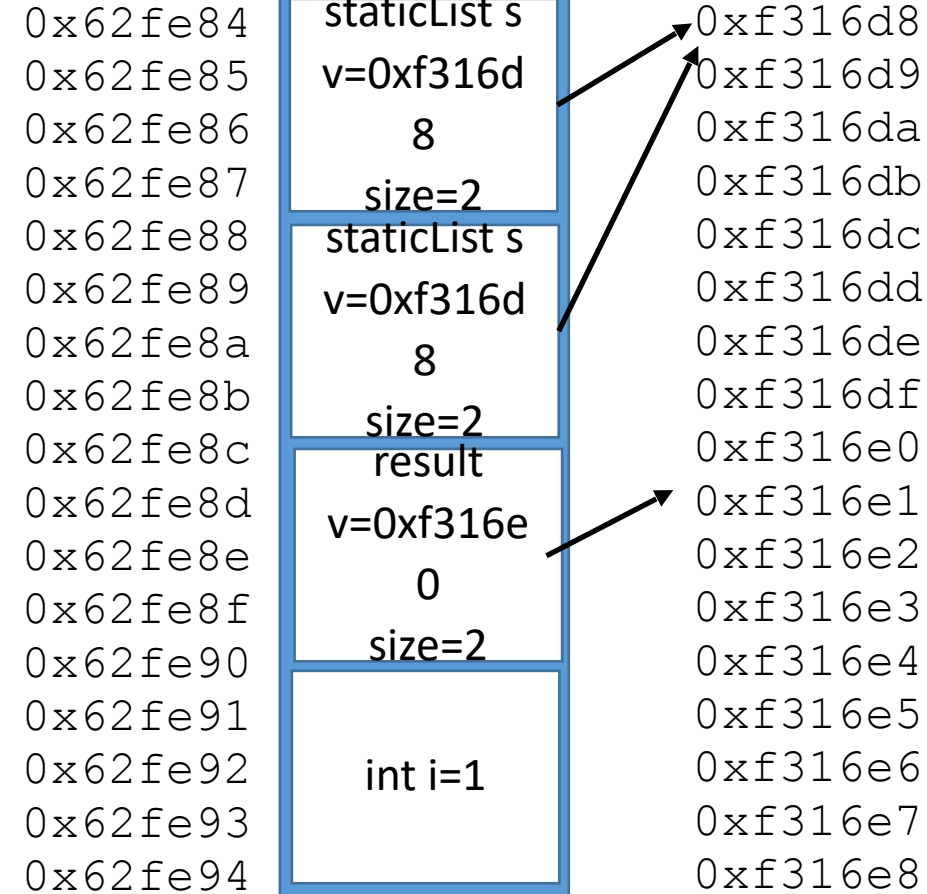
```
staticList reverse(staticList s) {
    staticList result(s.size);

    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

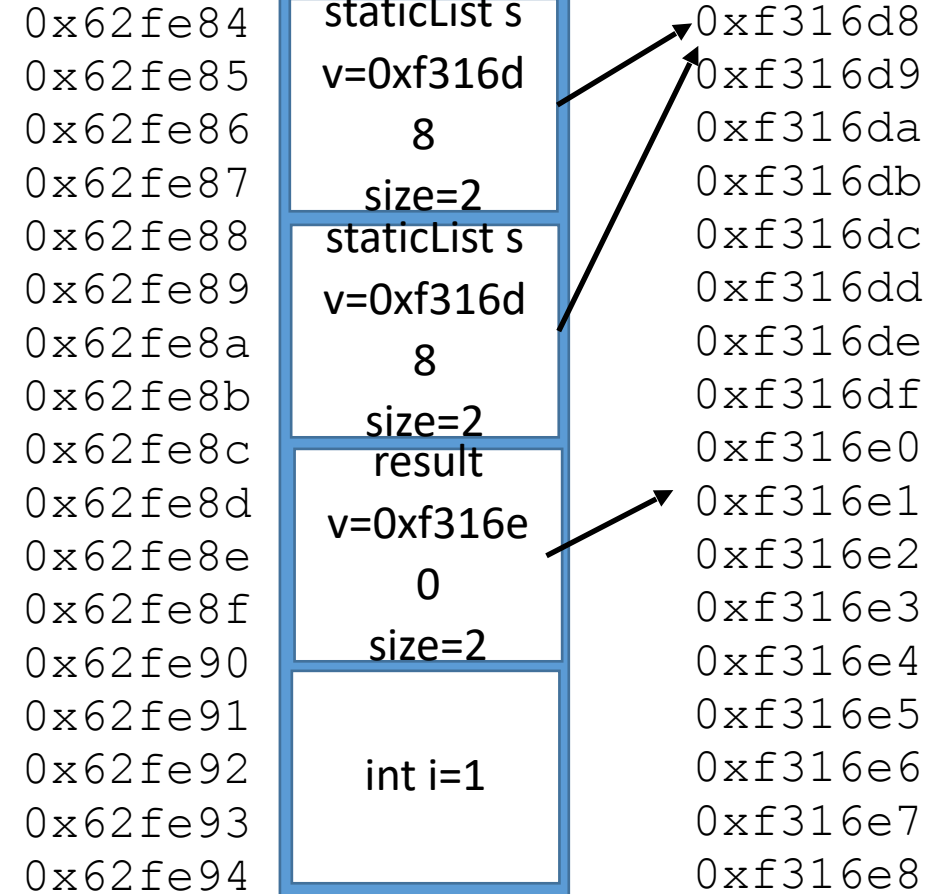
```
staticList reverse(staticList s) {
    staticList result(s.size);

    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

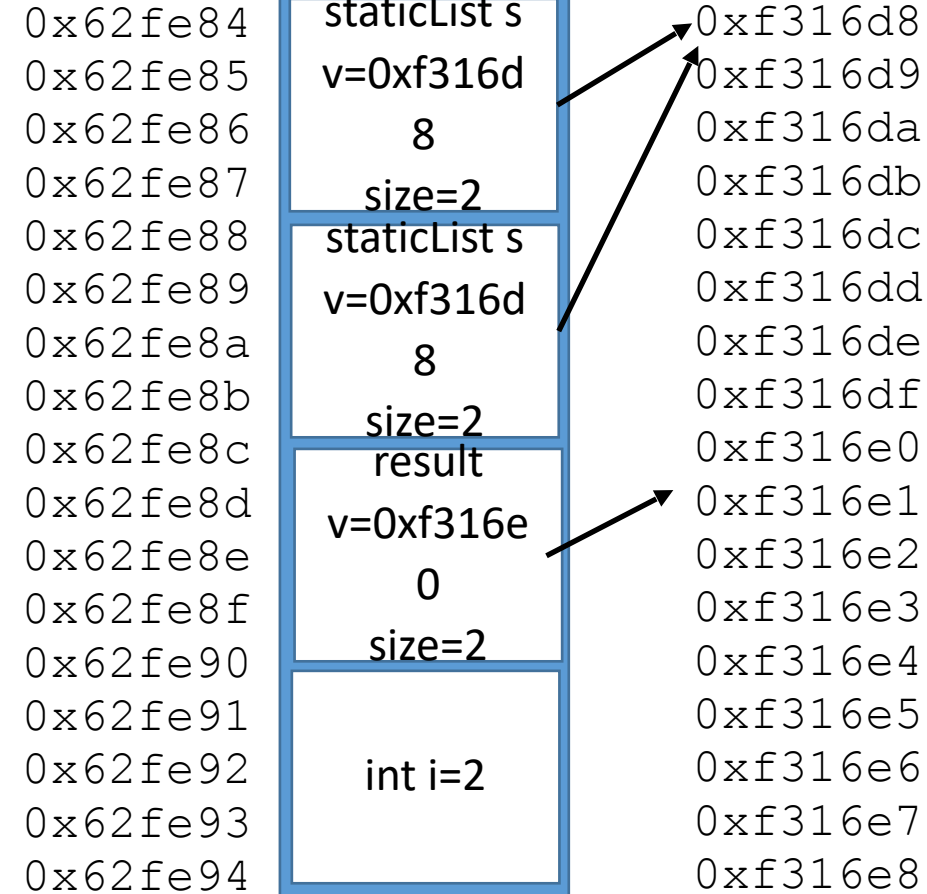
```
staticList reverse(staticList s) {
    staticList result(s.size);

    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

```
staticList reverse(staticList s) {
    staticList result(s.size);

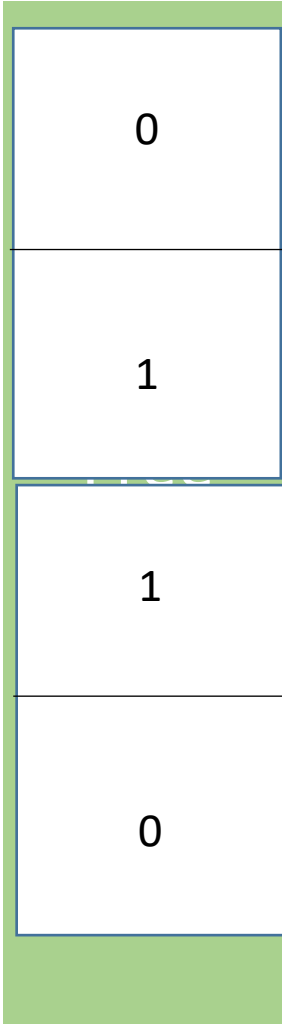
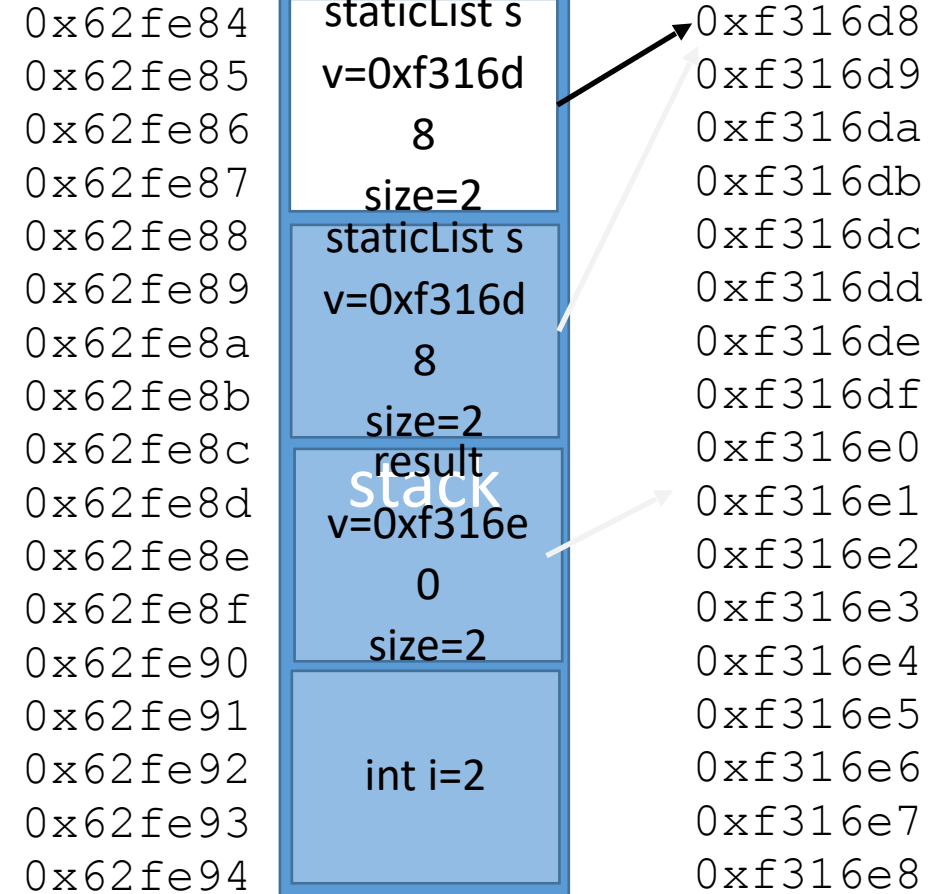
    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

Return value
v=0xf316e0
size=2

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

```
staticList reverse(staticList s) {
    staticList result(s.size);

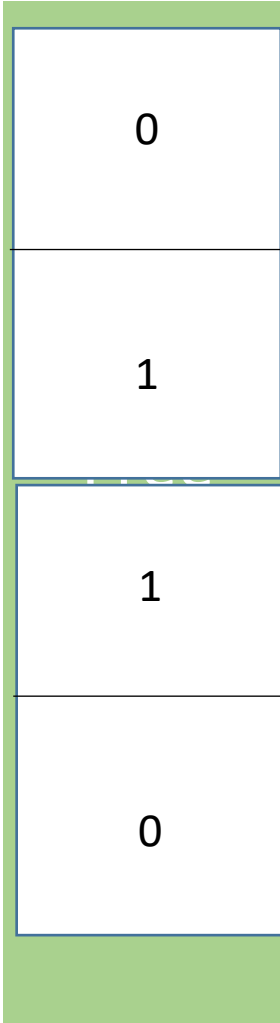
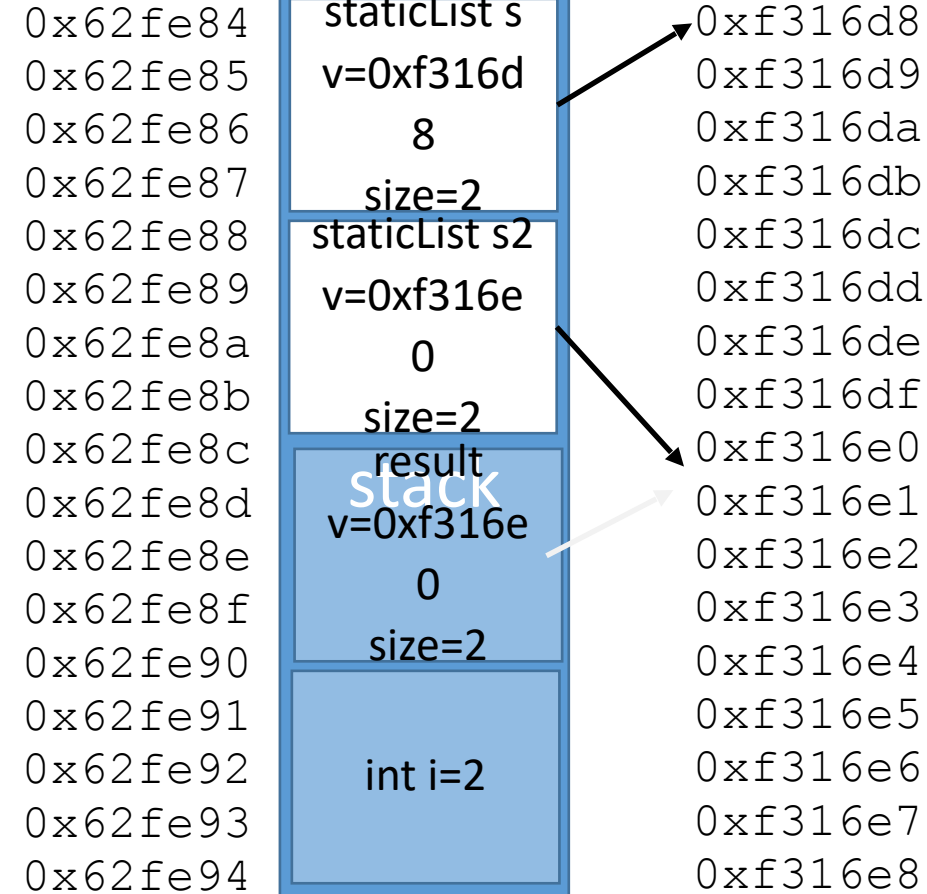
    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

Return value
v=0xf316e0
size=2

```
struct staticList {
    int* v;
    int size;
    ...
};
```



Struct en Call-By-Value

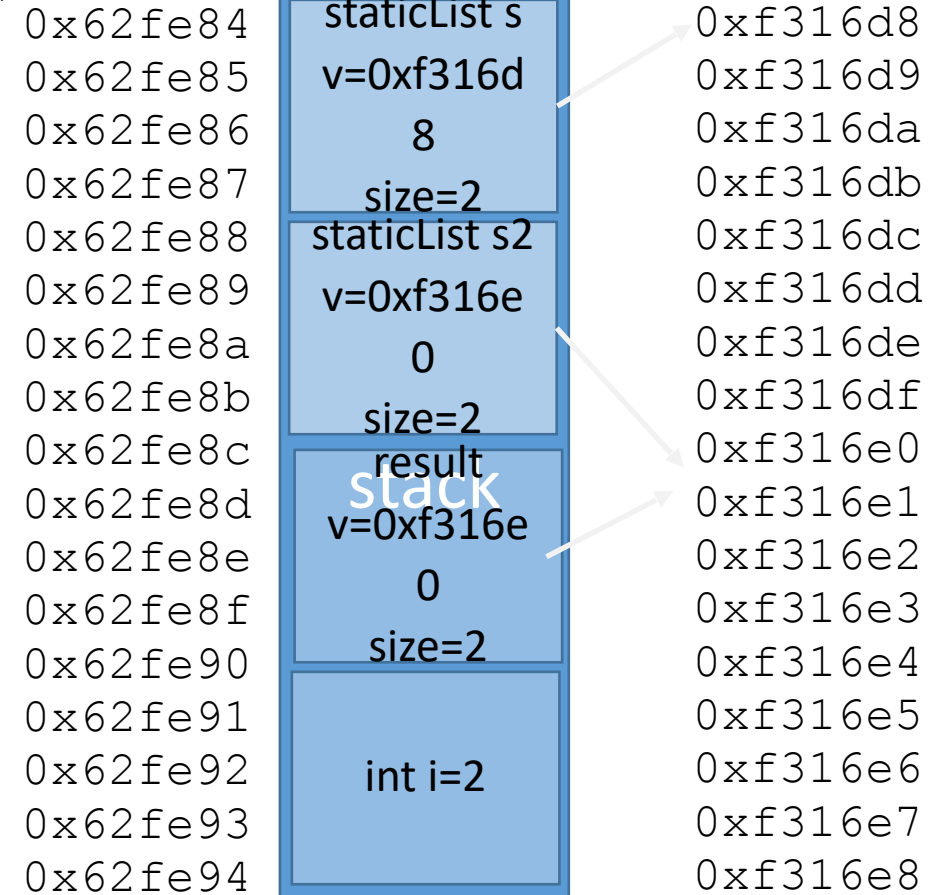
```
staticList reverse(staticList s) {
    staticList result(s.size);

    for (int i=0; i<s.size; i++) {
        result.v[i]=s.v[s.size-1-i];
    }

    return result;
}

int main() {
    staticList s=staticList(2);
    s.v[0]=0; s.v[1]=1;
    staticList s2=reverse(s);
    return 0;
}
```

```
struct staticList {
    int* v;
    int size;
    ...
};
```



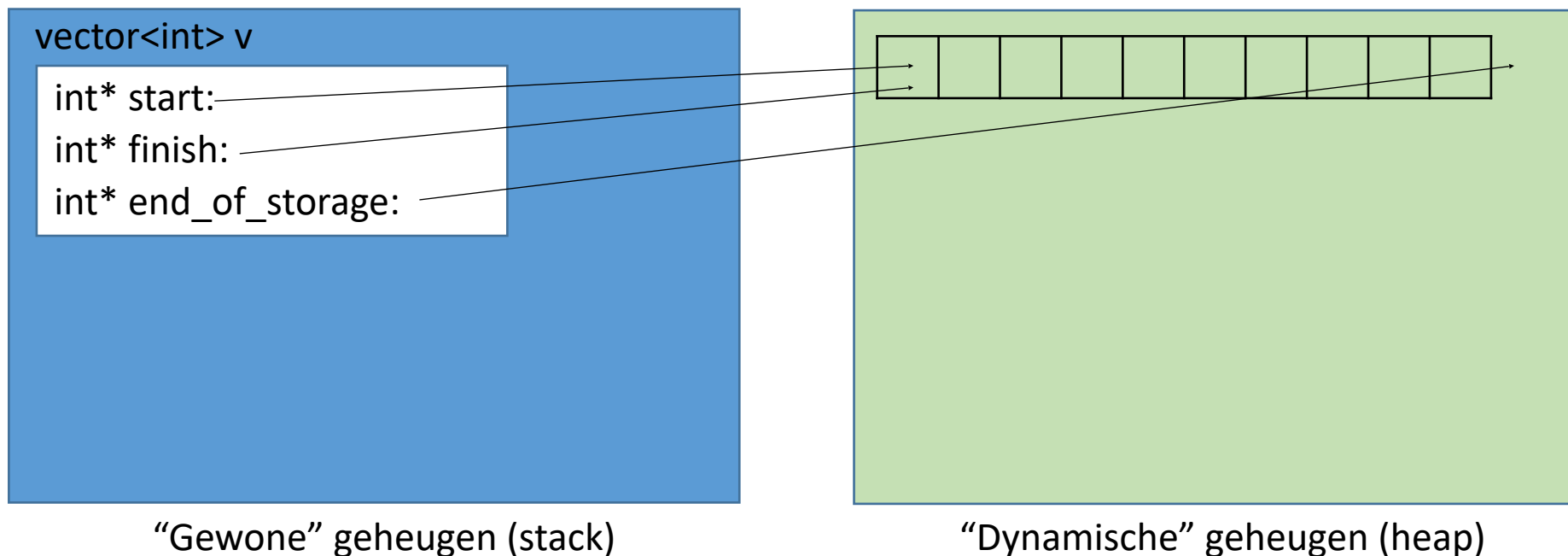
Intermezzo: sizeof(container)

- Sizeof(vector), sizeof(map), ... werken niet zoals verwacht

```
vector<int> v;  
cout << sizeof(v) << endl;  
  
v.push_back(1);  
v.push_back(2);  
v.push_back(3);  
cout << sizeof(v) << endl;
```

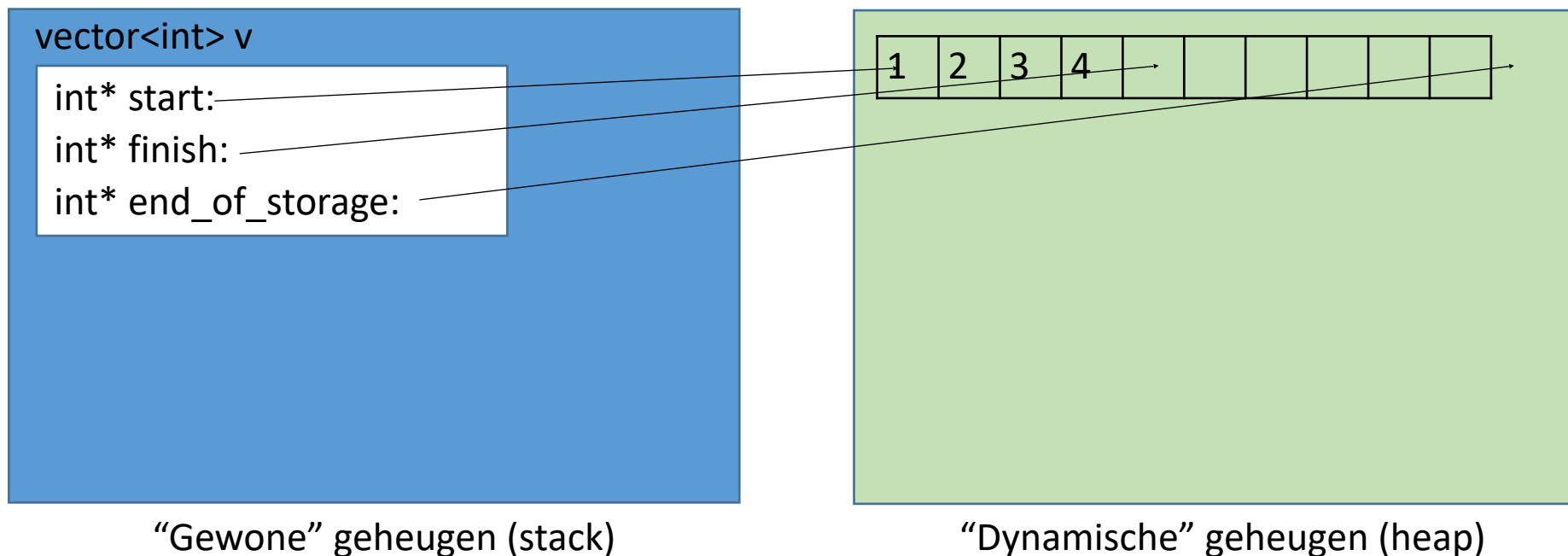

Intermezzo: sizeof(container)

- `sizeof(vector)`, `sizeof(map)`, ... werken niet zoals verwacht
- Dit komt omdat deze containers gebruik maken van *dynamische* geheugen allocatie



Intermezzo: sizeof(container)

- `sizeof(vector)`, `sizeof(map)`, ... werken niet zoals verwacht
- Dit komt omdat deze containers gebruik maken van *dynamische* geheugen allocatie



Oefening 1

- Maak een functie *string* lees(int n)* die een lijst van n namen inleest en deze dan teruggeeft

Voor strings gebruik

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

Oefening 2

Maak een functie:

- Input:
 - Een geordende lijst getallen (int^*) v
 - De lengte van de lijst n
 - En een getal i (int)
- Output:
 - Een nieuwe lijst (int^*) van lengte $n+1$ waarin i op de juiste plaats ingevoegd werd

Vb Struct - matrix

```
struct matrix {  
    double** data;  
    int r;  
    int c;  
  
    matrix(int rows, int cols) {  
        r = rows;  
        c = cols;  
        data = new double*[rows];  
        for (int i = 0; i < rows; i++) {  
            data[i] = new double[cols];  
            for (int j = 0; j < cols; j++) {  
                data[i][j] = 0.0;  
            }  
        }  
    }  
};
```

```
void printMatrix(matrix M) {  
    for (int i = 0; i < M.r; i++) {  
        for (int j = 0; j < M.c; j++) {  
            cout << M.data[i][j] << "\t";  
        }  
        cout << endl;
```