Universiteit Antwerpen

Inleiding Programmeren
C++ Geheugenbeheer; pointers

Tom Hofkens

Universiteit Antwerpen

Wat zagen we al?

- C++ basis
  - Statisch getypeerde taal
  - Gecompileerd
  - Gelijkaardige controlestructuren als Python

- Echter:
  - Je *moet* alle variabelen expliciet typeren en typering is strikt

Universiteit Antwerpen

Deze les

- Pointers
  - Variabelen als alias voor geheugenlocaties
  - Pointer type
  - Referencing en dereferencing
  - Rekenen met pointers
- Referentie variabelen
- Parameters bij functieaanroepen
  - Call by value / Call by reference
- Call stack

Bekijk ook volgende uitstekende tutorial:

- http://www.cplusplus.com/doc/tutorial/pointers/

Universiteit Antwerpen

Variabele als alias van een geheugenlocatie

- Een variabele kan beschouwd worden als een alias voor een geheugenlocatie
  - *At runtime* wordt er geen enkele variabele naam gebruikt; compiler schrijft code die “rechtstreeks” de geheugenlocatie benadert.

- &X is de geheugenlocatie van een variabele X (id(X) in Python).
- sizeof(X) geeft weer hoe groot de geheugenlocatie voor X is

Universiteit Antwerpen

sizeof

- Met sizeof(.) kunnen we opvragen hoeveel bytes een geheugenplaats in beslag neemt.

```
int i;
double d;
long l;

cout << "Variabelen i/d/l staan op geheugenplaatsen " << &i
     << "/" << &d << "/" << &l << endl;
cout << "Ze nemen " << sizeof(i) << "/" << sizeof(d)
     << "/" << sizeof(l) << " bytes in beslag" << endl;
```

Variabelen i/d/l staan op geheugenplaatsen 0x63fe9c/0x63fe90/0x63fe8c
Ze nemen 4/8/4 bytes in beslag

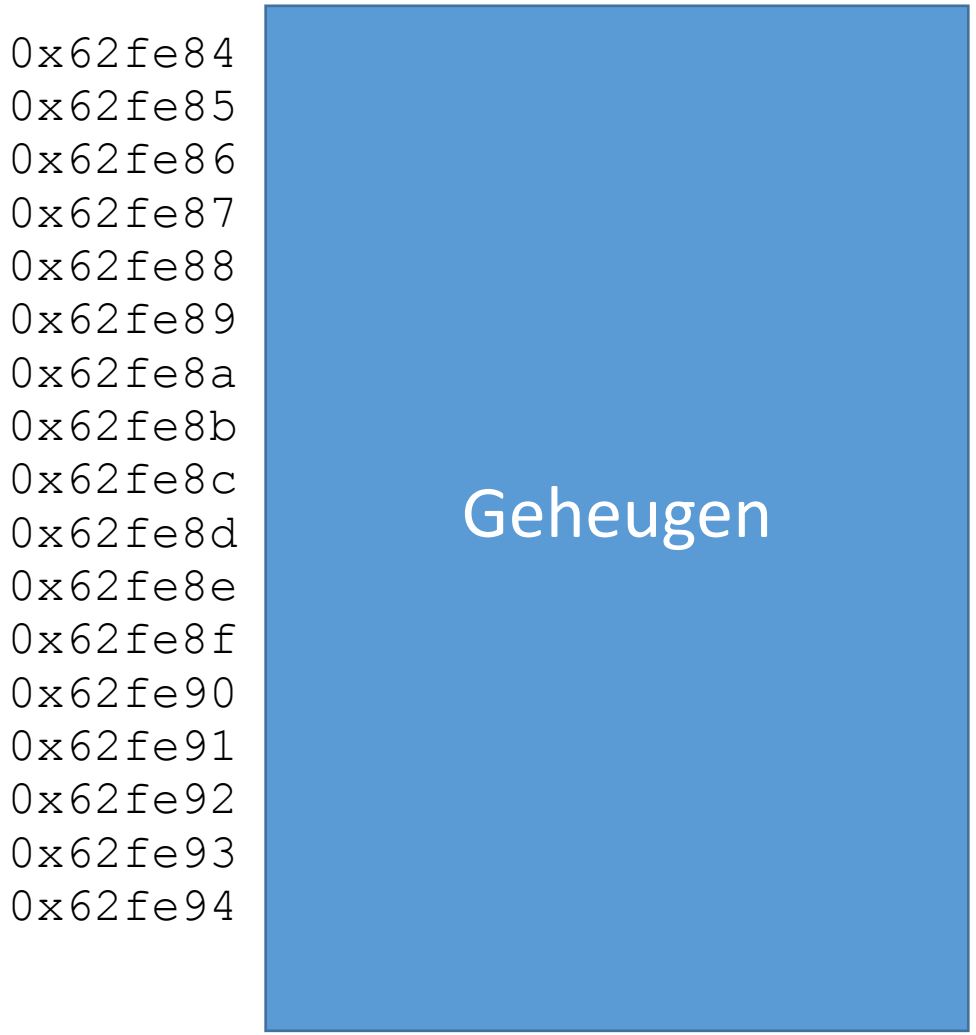
Universiteit Antwerpen

Pointers

- We kunnen geheugenlocaties rechtstreeks gebruiken in C++
- Een getypeerde geheugenlocatie is een pointer
  - &X is de geheugenlocatie van een variabele X.
  - Als X type T heeft, dan is &X van type T* (lees: *pointer naar T*)
- Als p een pointer is van type T*, dan is *p de locatie waar die naar verwijst.

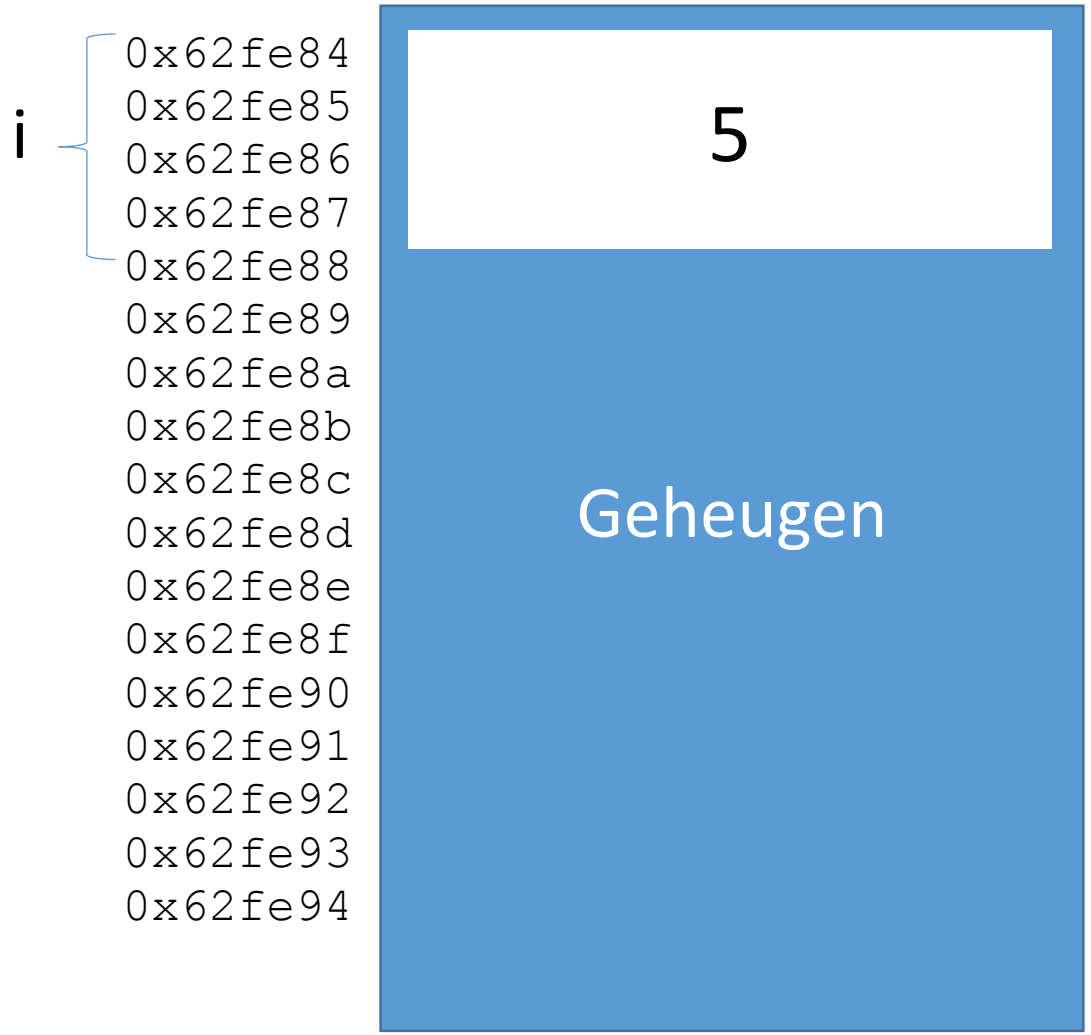
Pointers: voorbeelden

```
int main() {
```



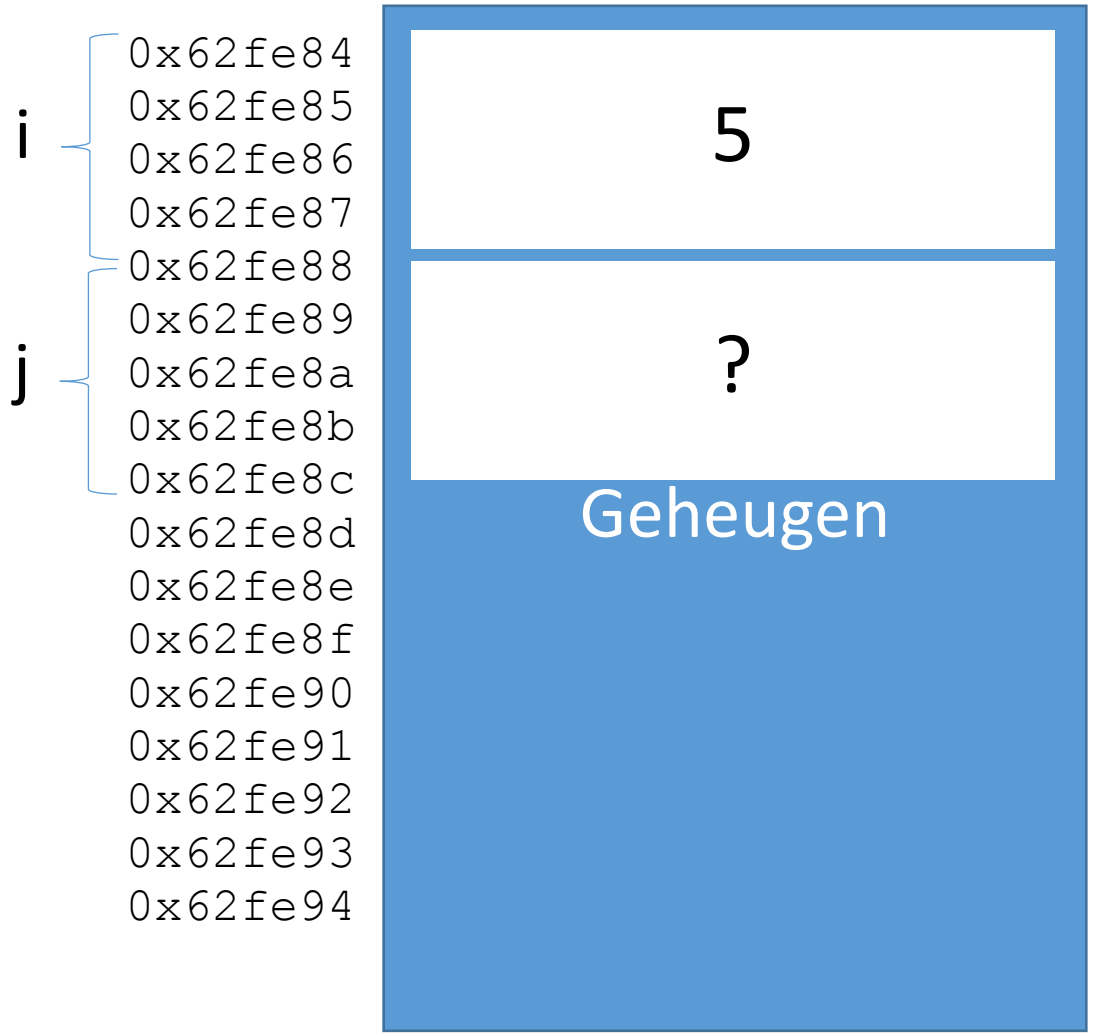
Pointers: voorbeelden

```
int main() {
    int i=5;
```



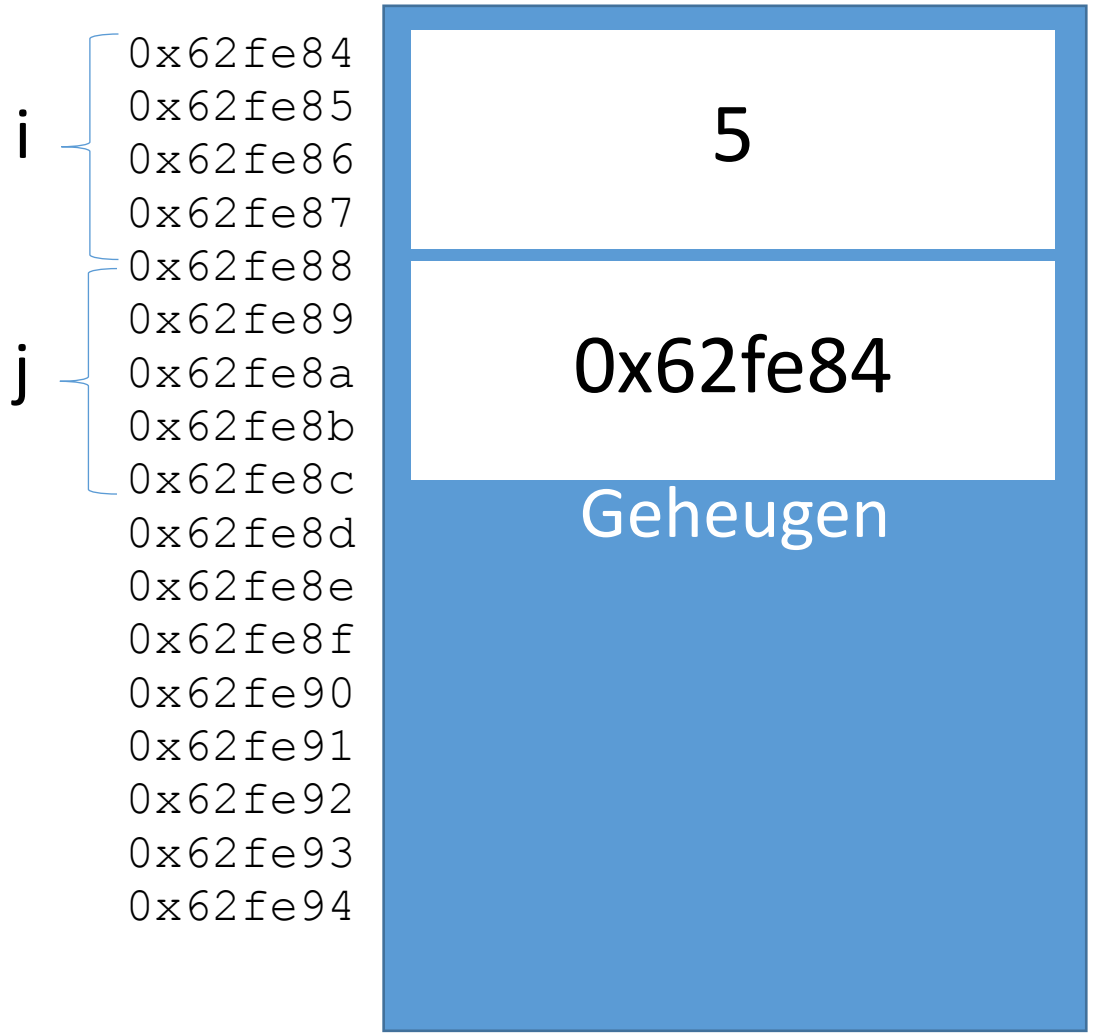
Pointers: voorbeelden

```
int main() {
    int i=5;
    int* j=&i;
```



Pointers: voorbeelden

```
int main() {
    int i=5;
    int* j=&i;
```

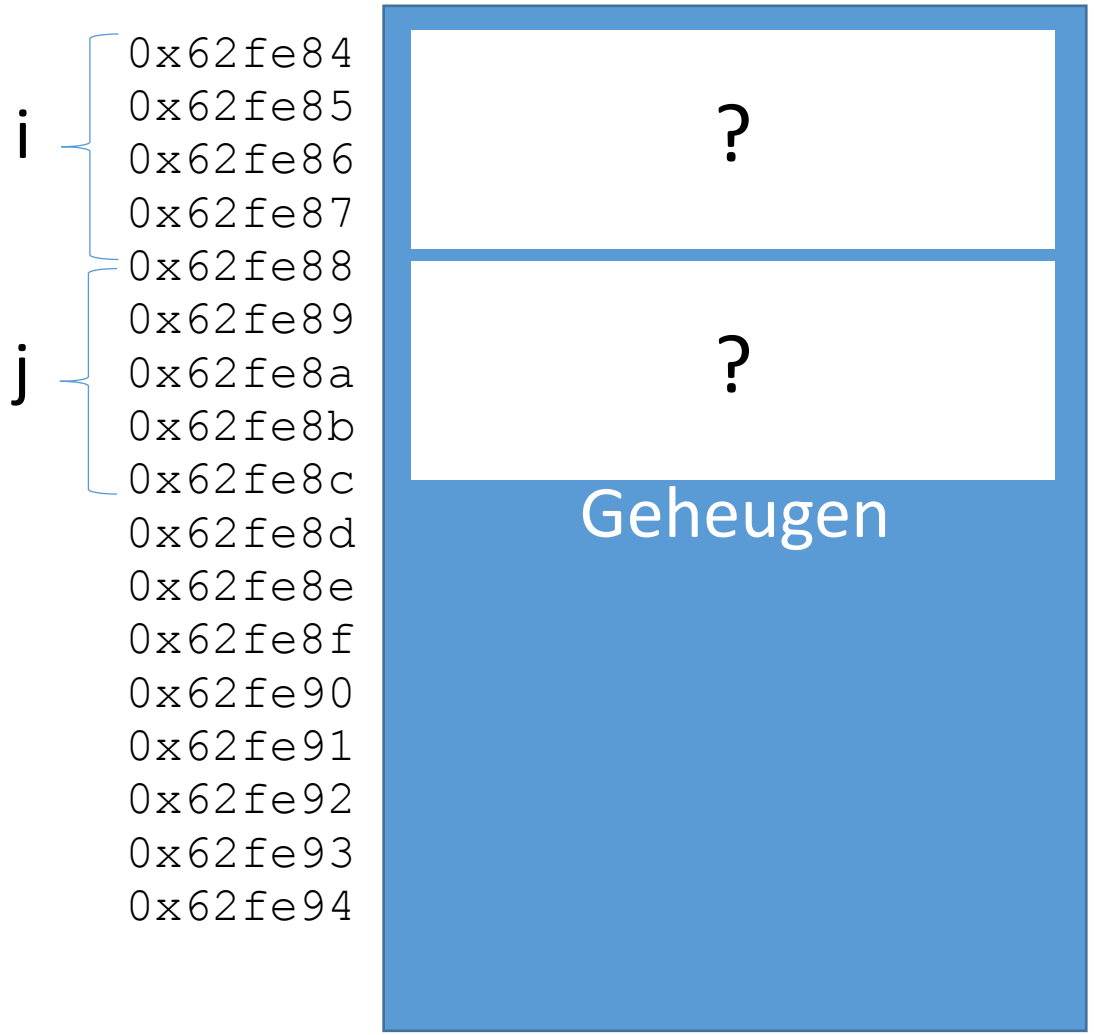


Pointers: voorbeelden

```
int main() {
    int i=5;
    int* j=&i;
    *j=3;
```

De geheugenplaats waar j naar wijst

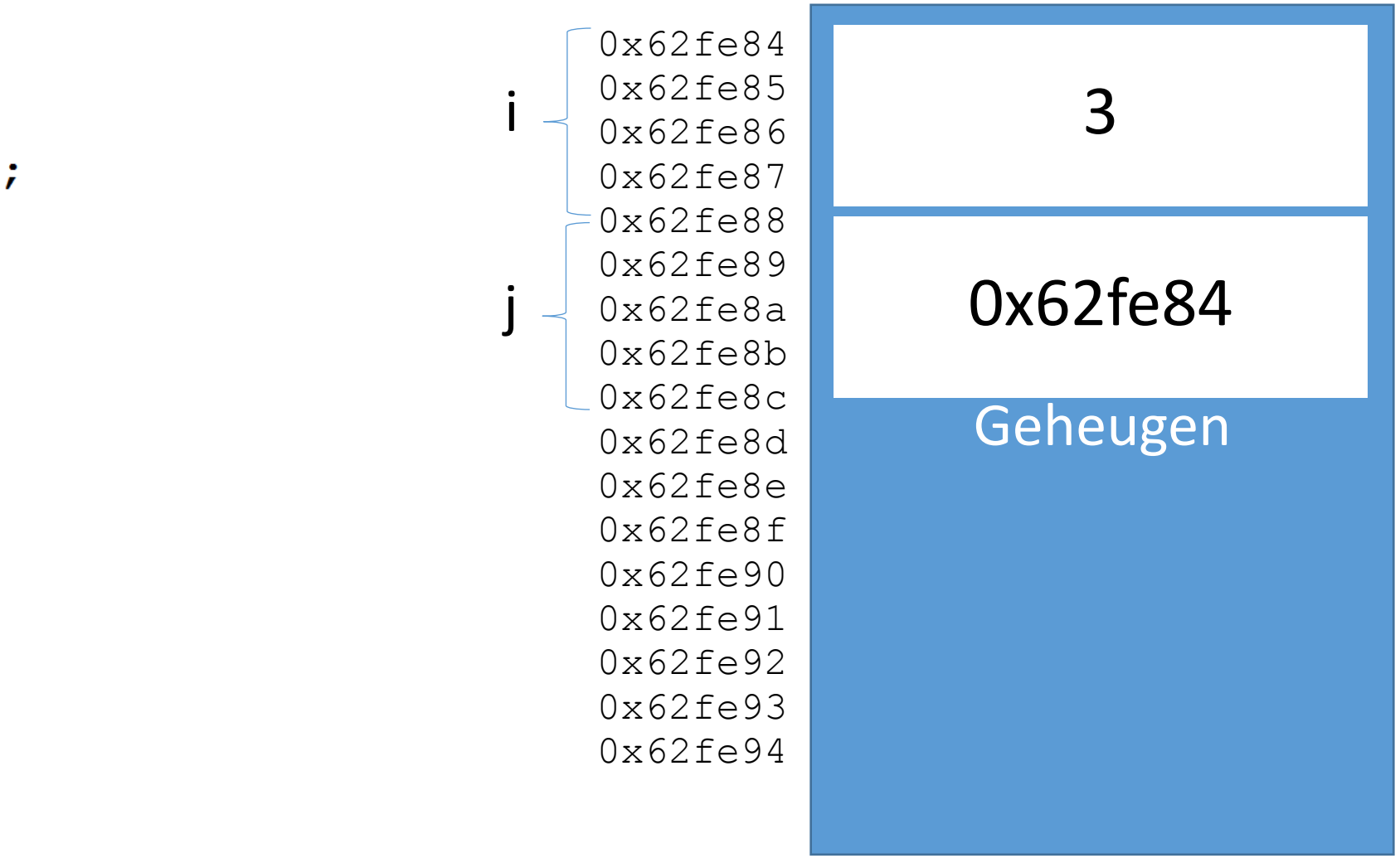
Wat is het effect op het geheugen?



Pointers: voorbeelden

```
int main() {
    int i=5;
    int* j=&i;
    *j=3;
```

De geheugenplaats waar j naar wijst



Pointers: voorbeelden

```
int main() {
    int i=5;
    int* j=&i;
    *j=3;
    cout << "i staat op locatie " << &i
         << " en bevat " << i << endl;
    cout << "j staat op locatie " << &j
         << " en bevat " << j << endl;
    cout << "Op geheugenlocatie " << j
         << " staat " << *j << endl;
```

```
i staat op locatie 0x62fe84 en bevat 3
j staat op locatie 0x62fe80 en bevat 0x62fe84
Op geheugenlocatie 0x62fe84 staat 3
```

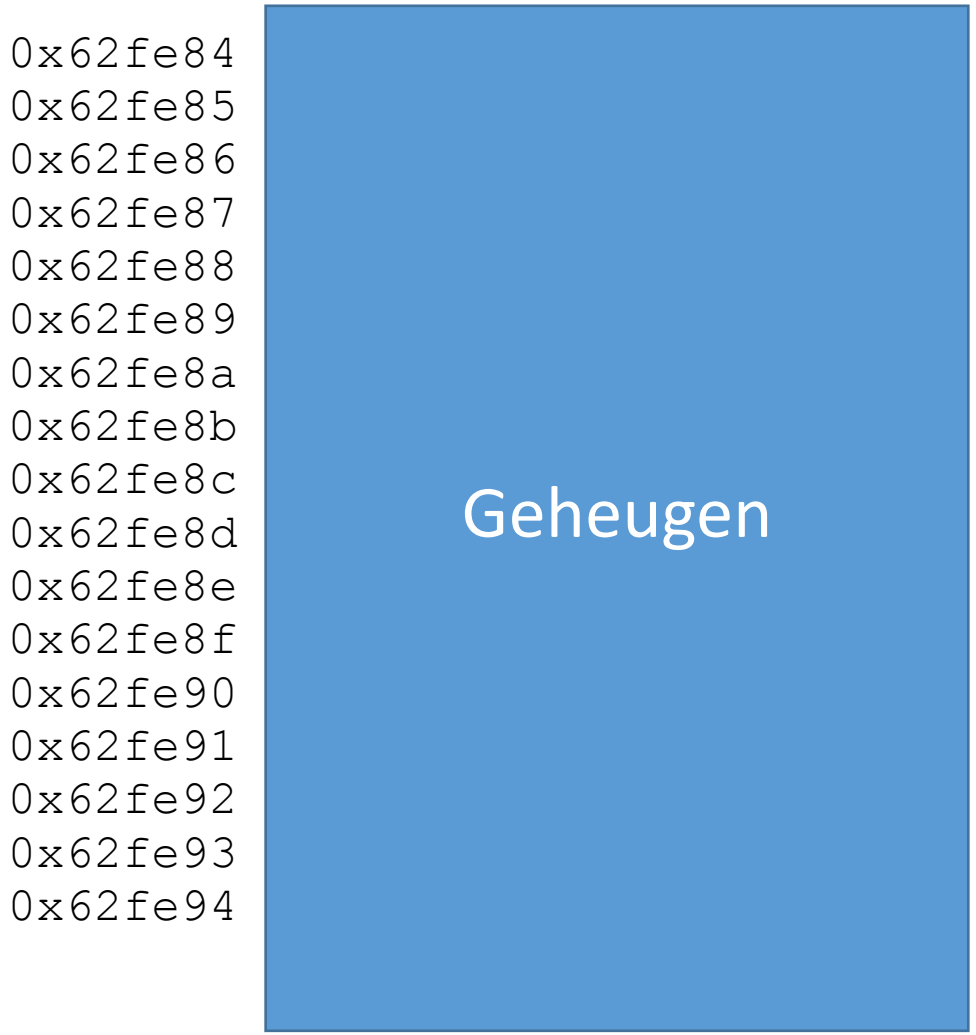
Lvalues vs Rvalues

- Lvalue: alles wat aan de linkerkant van “=” kan staan; m.a.w. *Alles wat met een opslaglocatie geassocieerd kan worden*
  - Variabele, element in een vector, element in een array
- Rvalue: alles wat aan de rechterkant van “=” kan staan
  - Uitdrukkingen, functieaanroepen (niet-void functies)

- We kunnen enkel pointers maken naar lvalues
  - Logisch; een pointer wijst naar een geheugenlocatie
  - &(x+2) kan dus niet

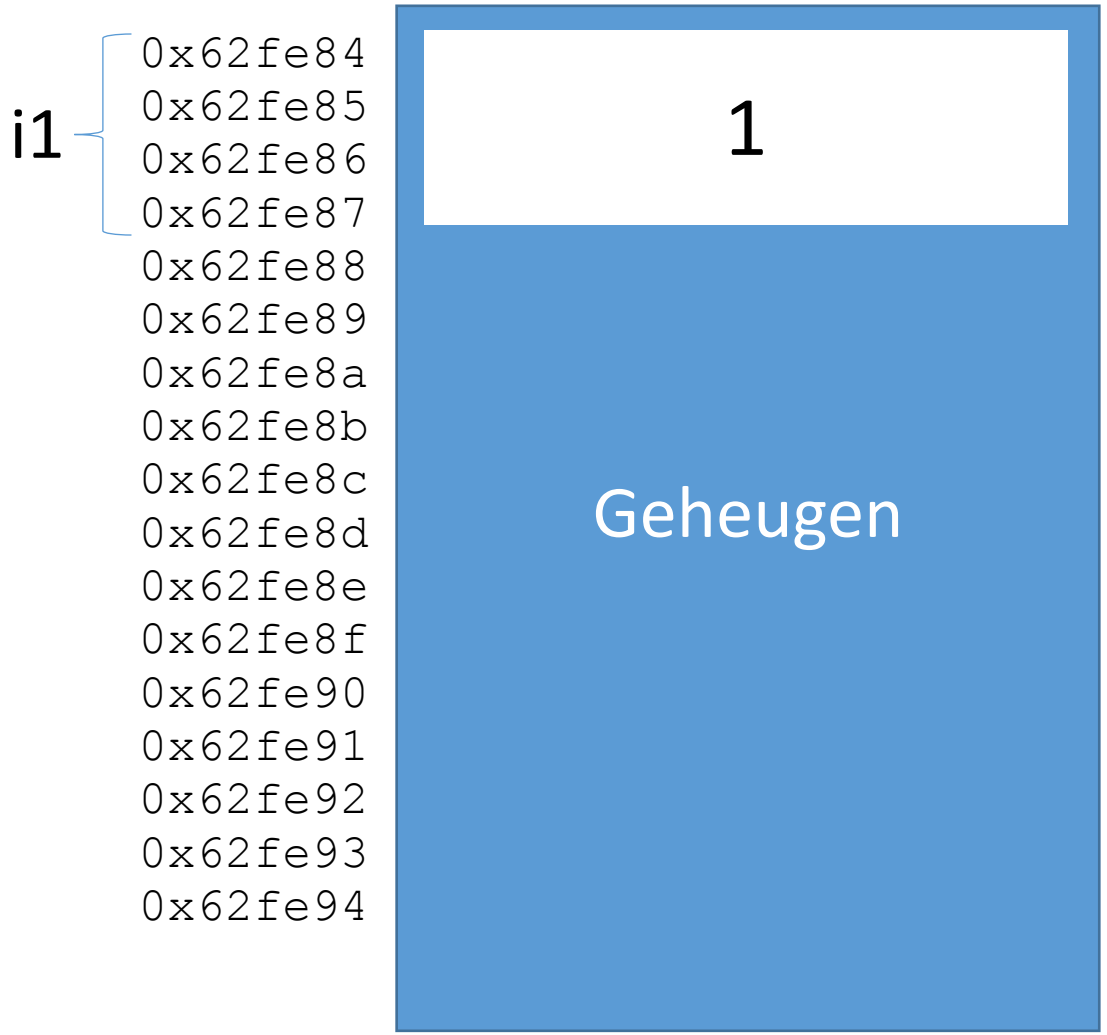
Lvalues vs Rvalues

```
int square(int x) {
    return x*x;
}

int main() {
```



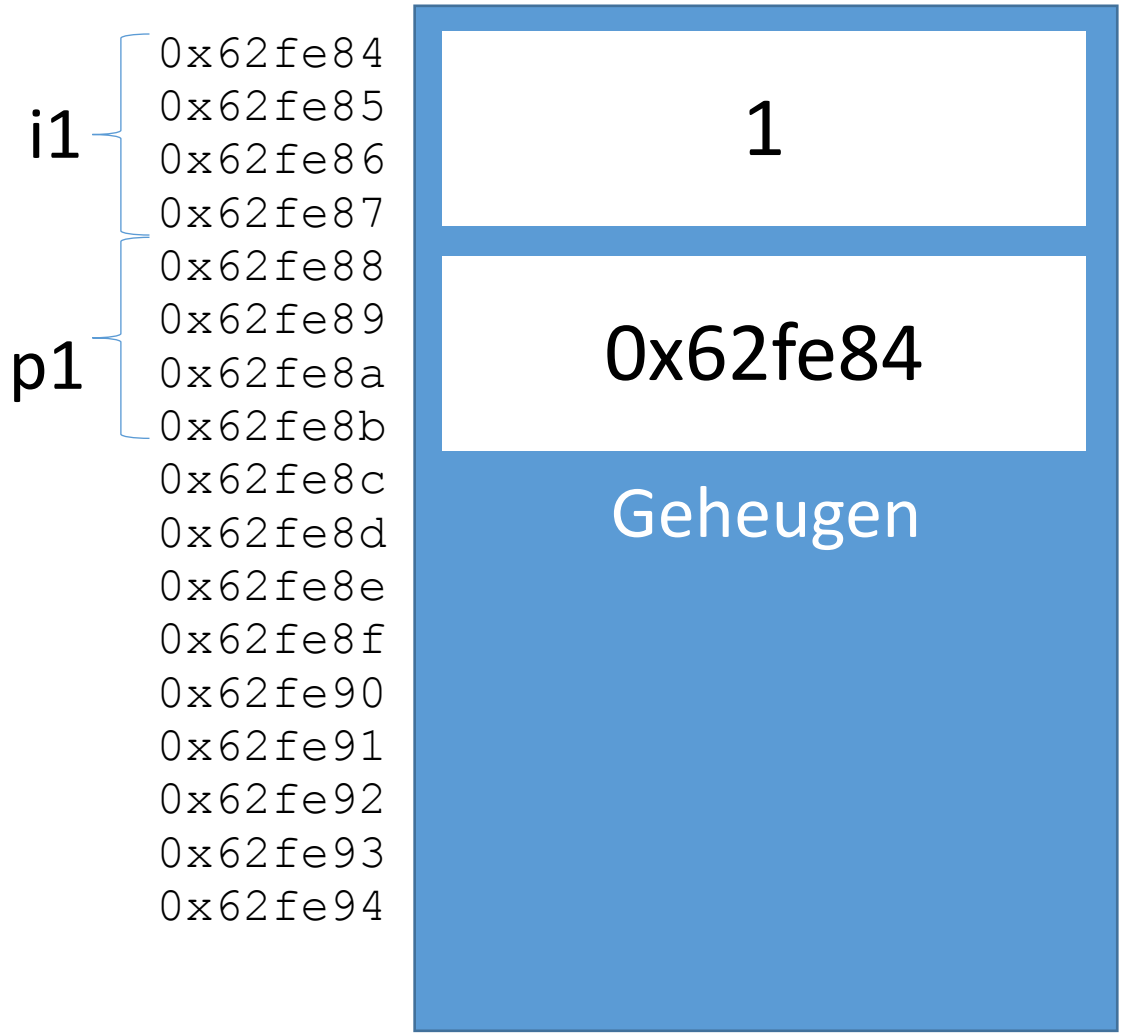
Lvalues vs Rvalues

```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
```



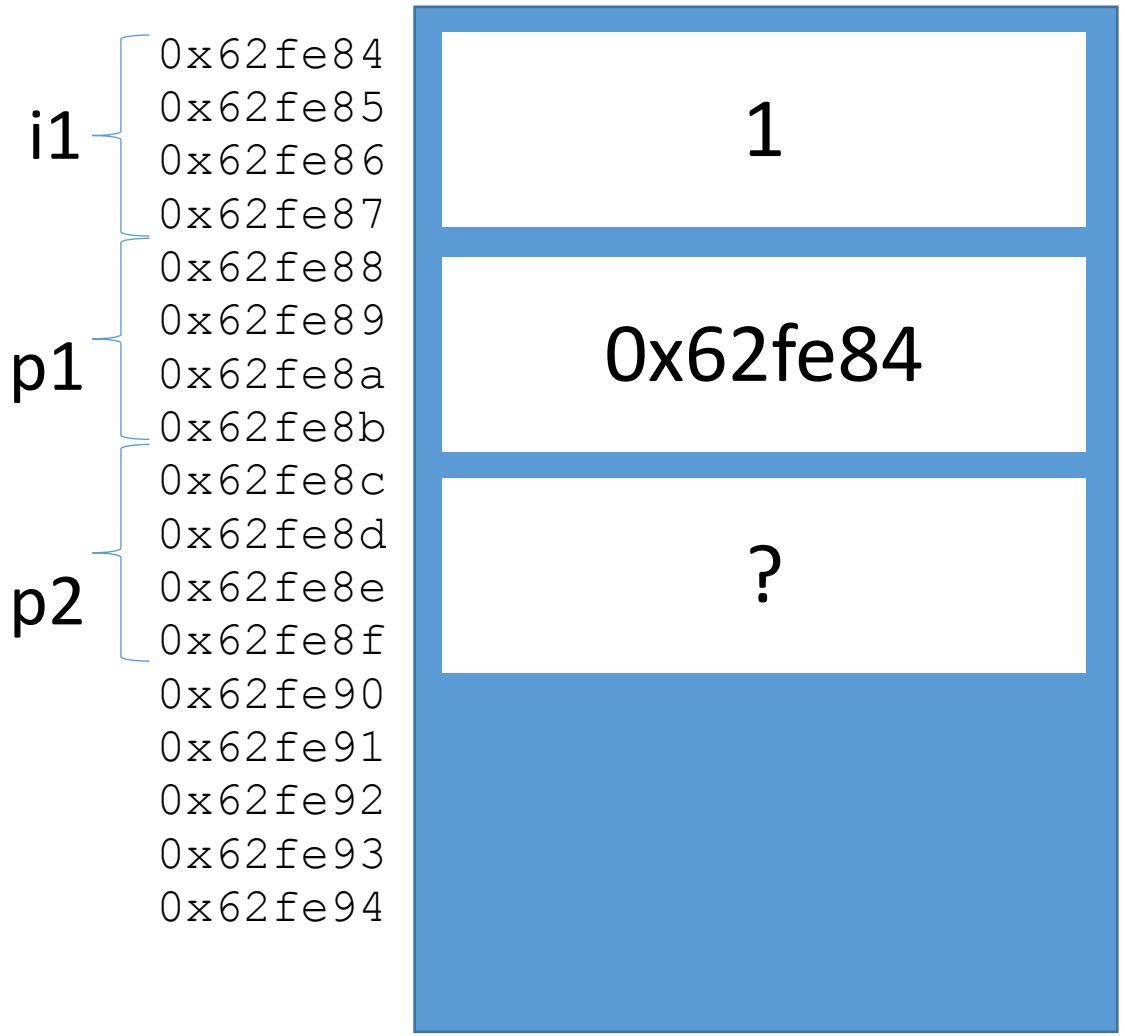
Lvalues vs Rvalues

```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
```



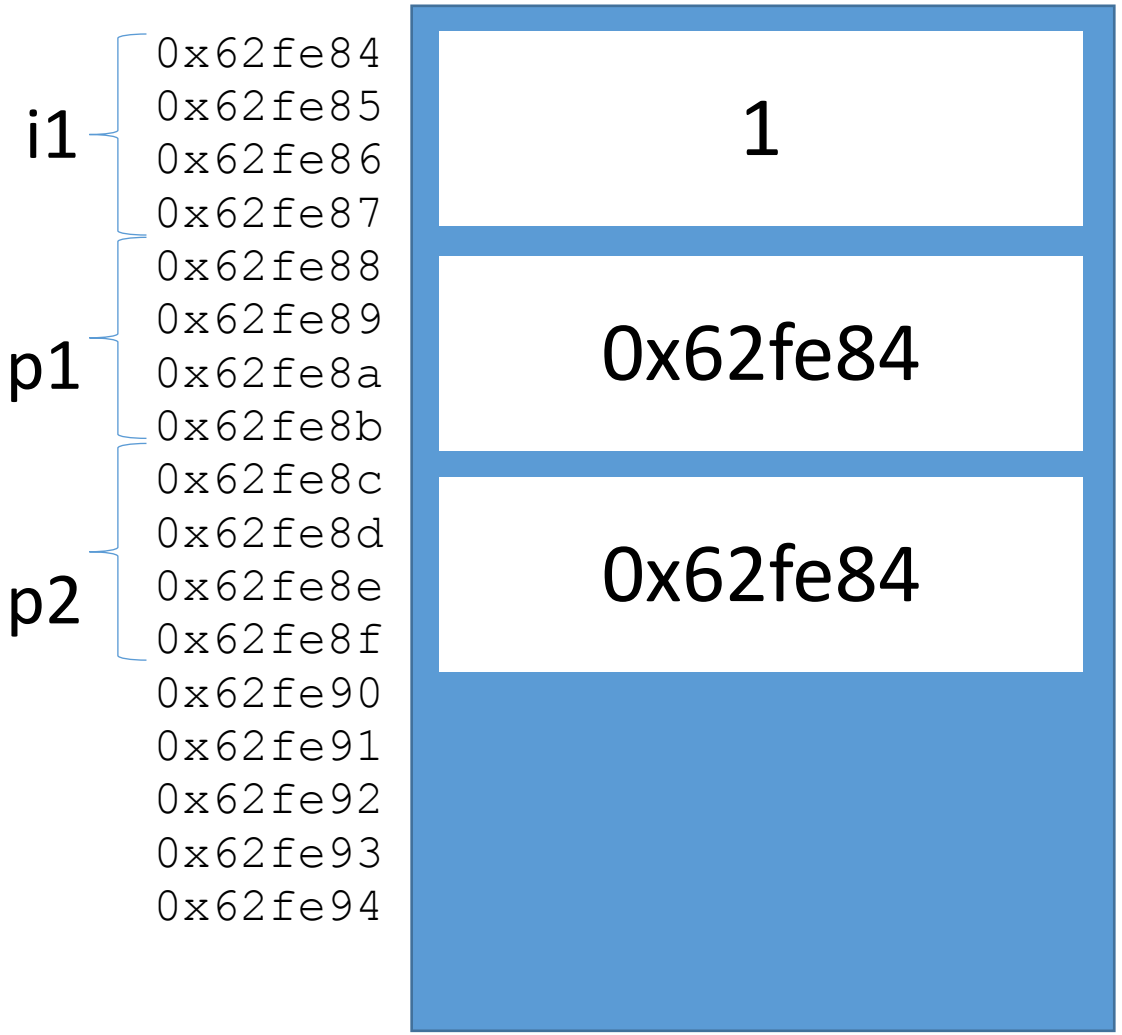
Lvalues vs Rvalues

```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
```



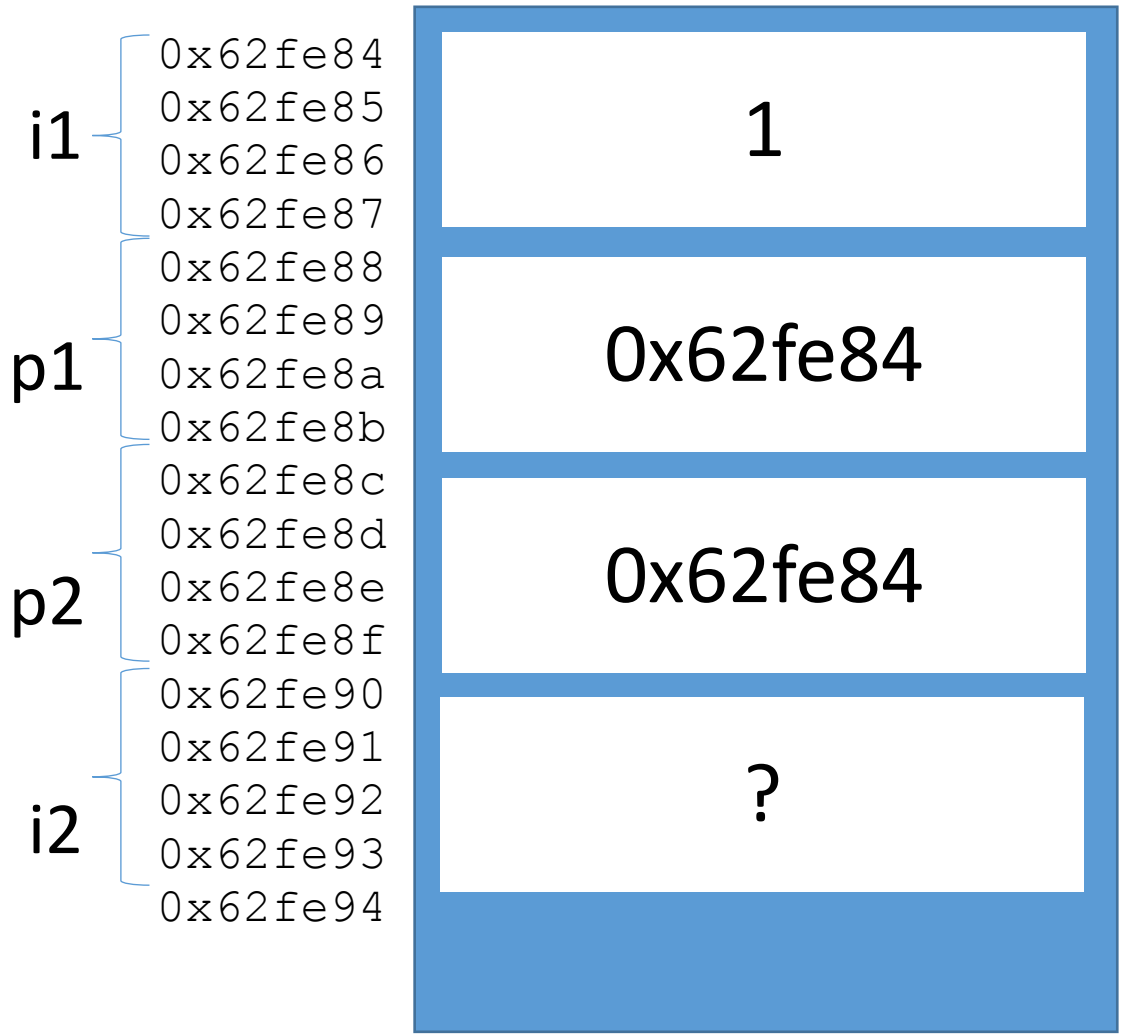
Lvalues vs Rvalues

```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
```



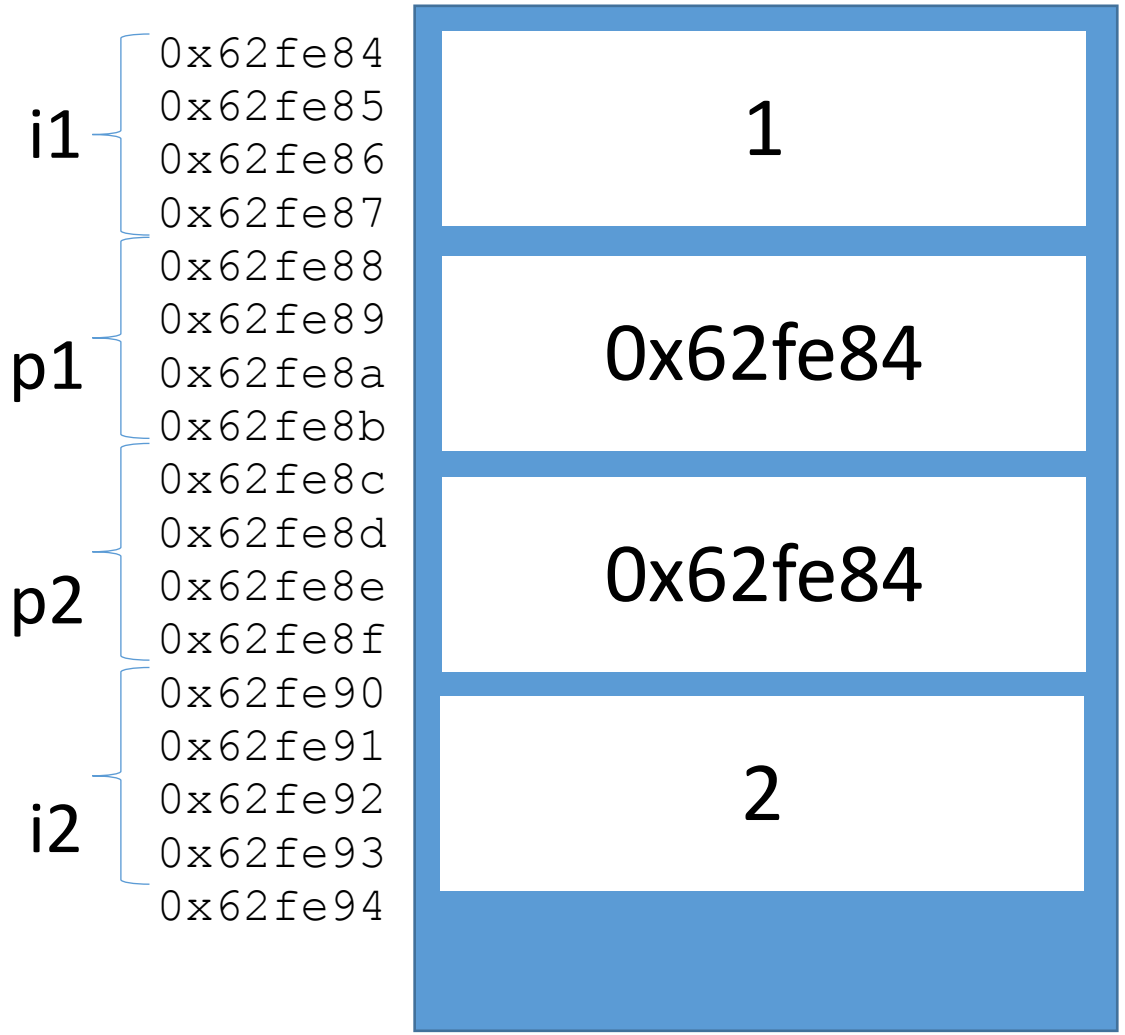
Lvalues vs Rvalues

```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
    int  i2 = *p2+1;
```



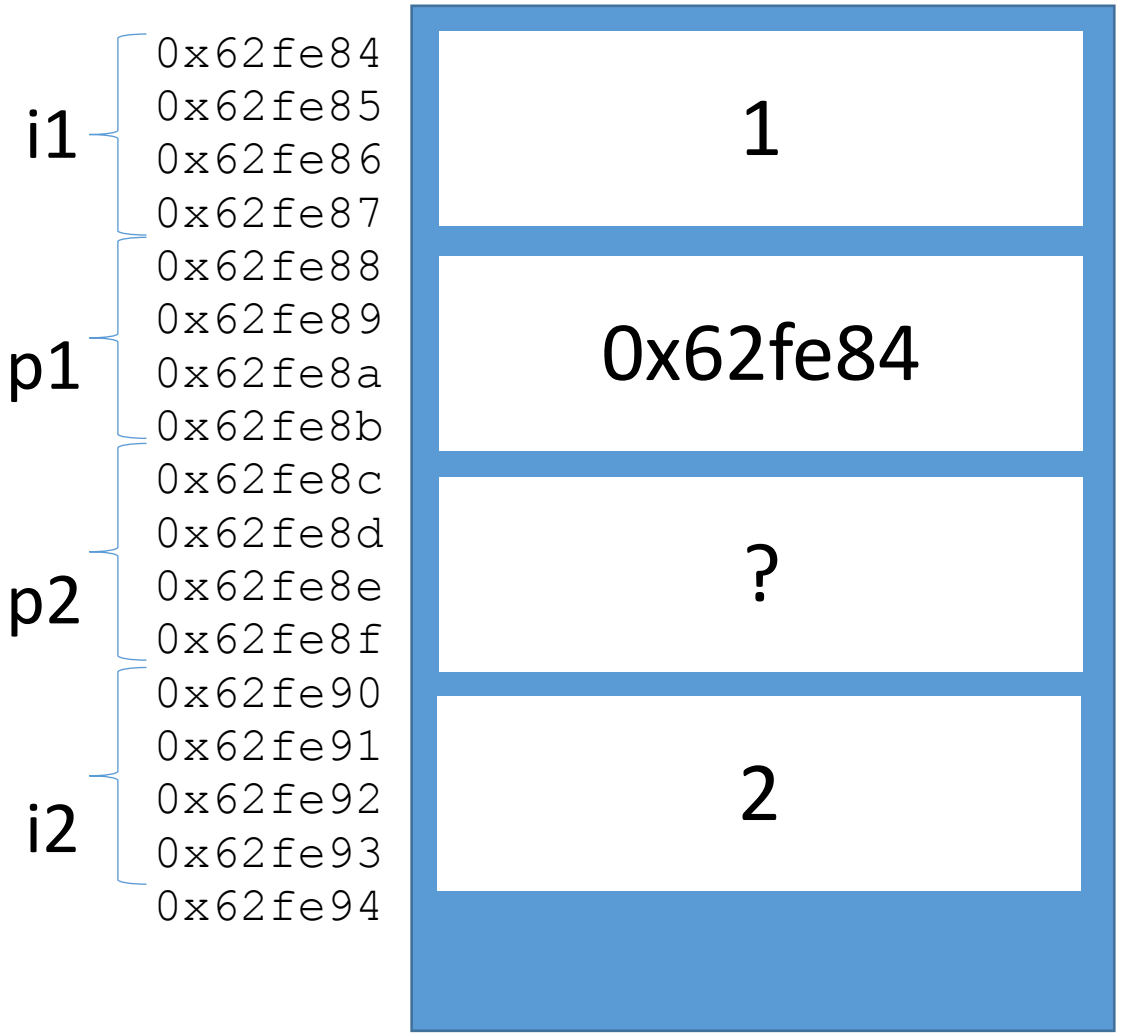
Lvalues vs Rvalues

```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
    int  i2 = *p2+1;
```



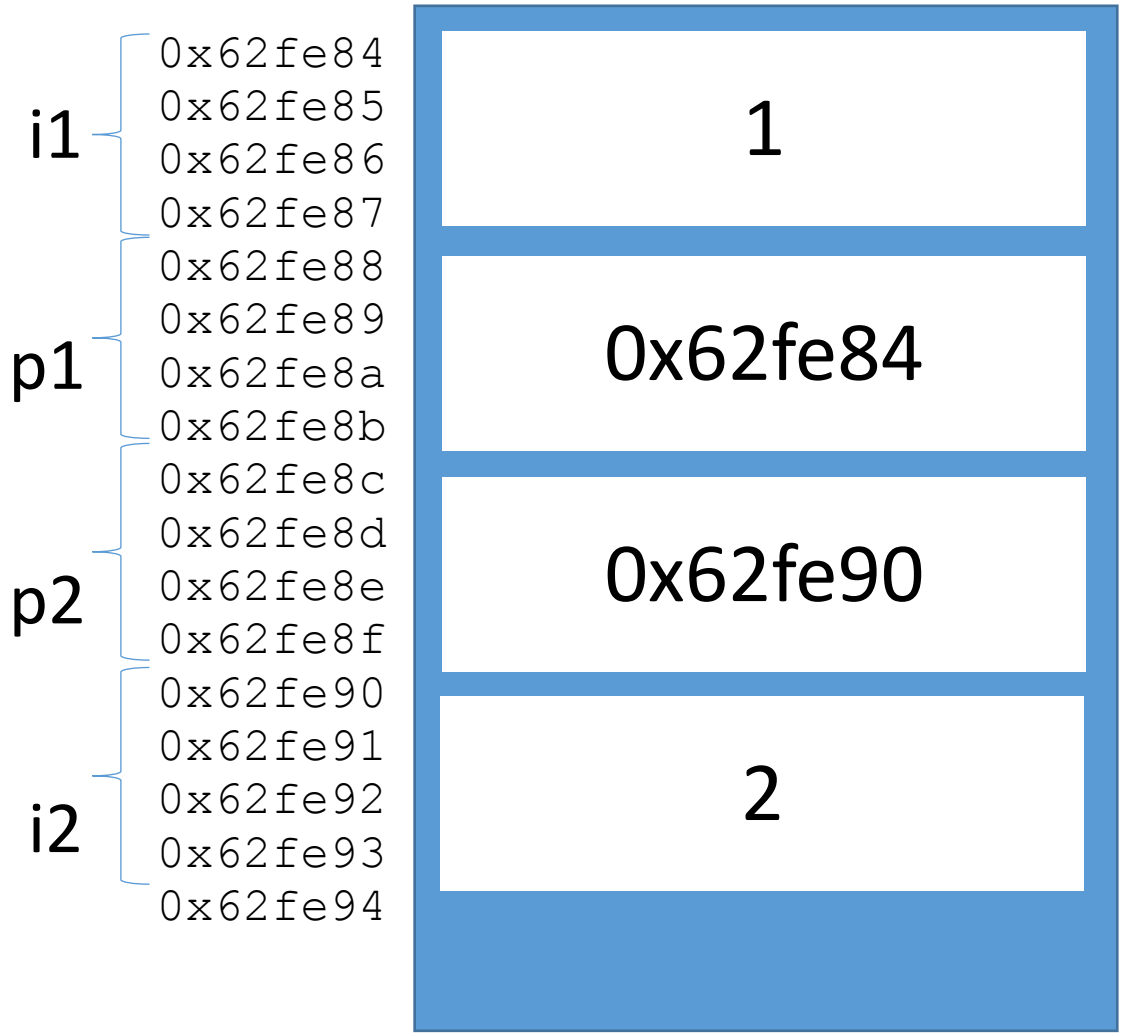
Lvalues vs Rvalues

```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
    int  i2 = *p2+1;
    p2 = &i2;
```



Lvalues vs Rvalues

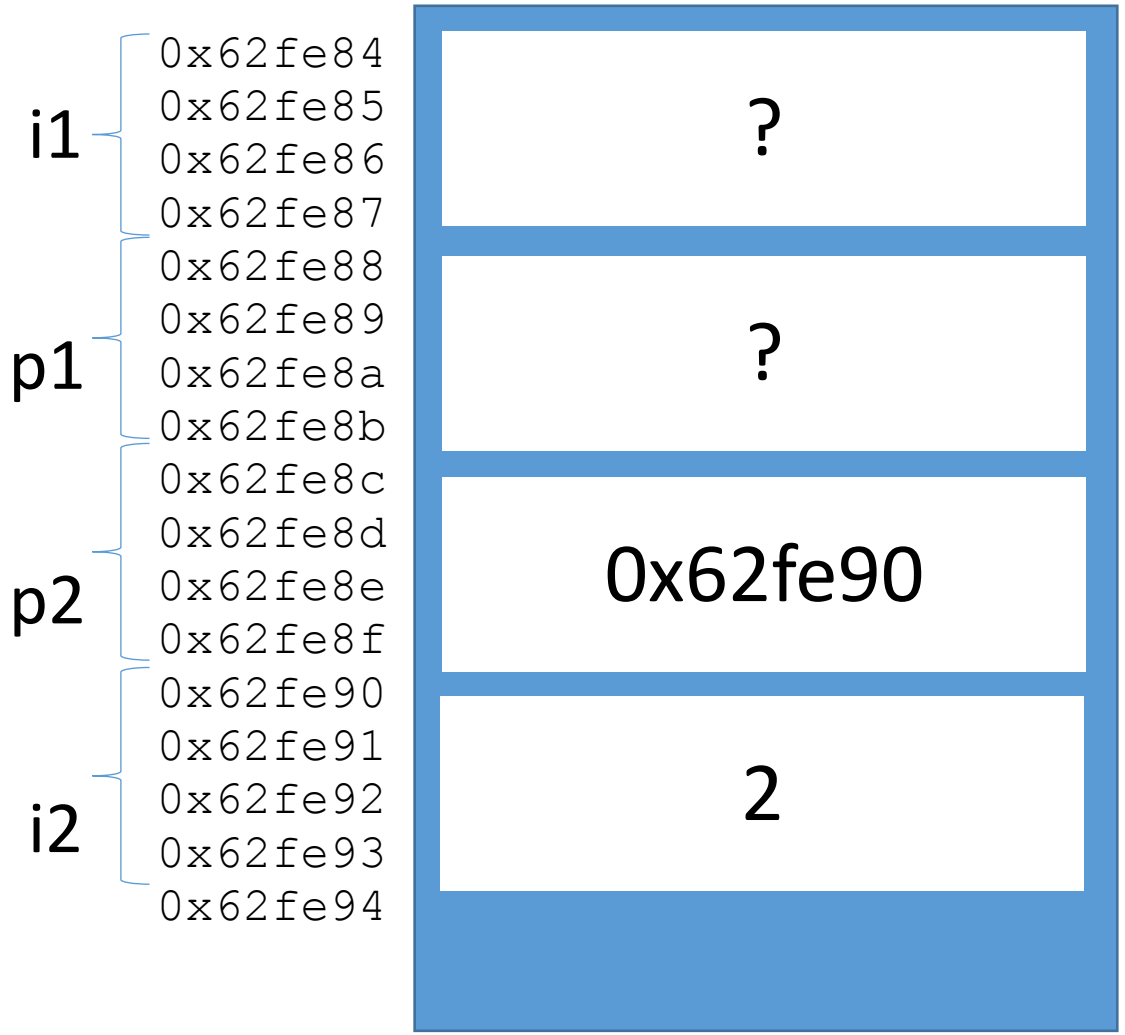
```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
    int  i2 = *p2+1;
    p2  = &i2;
```



Lvalues vs Rvalues

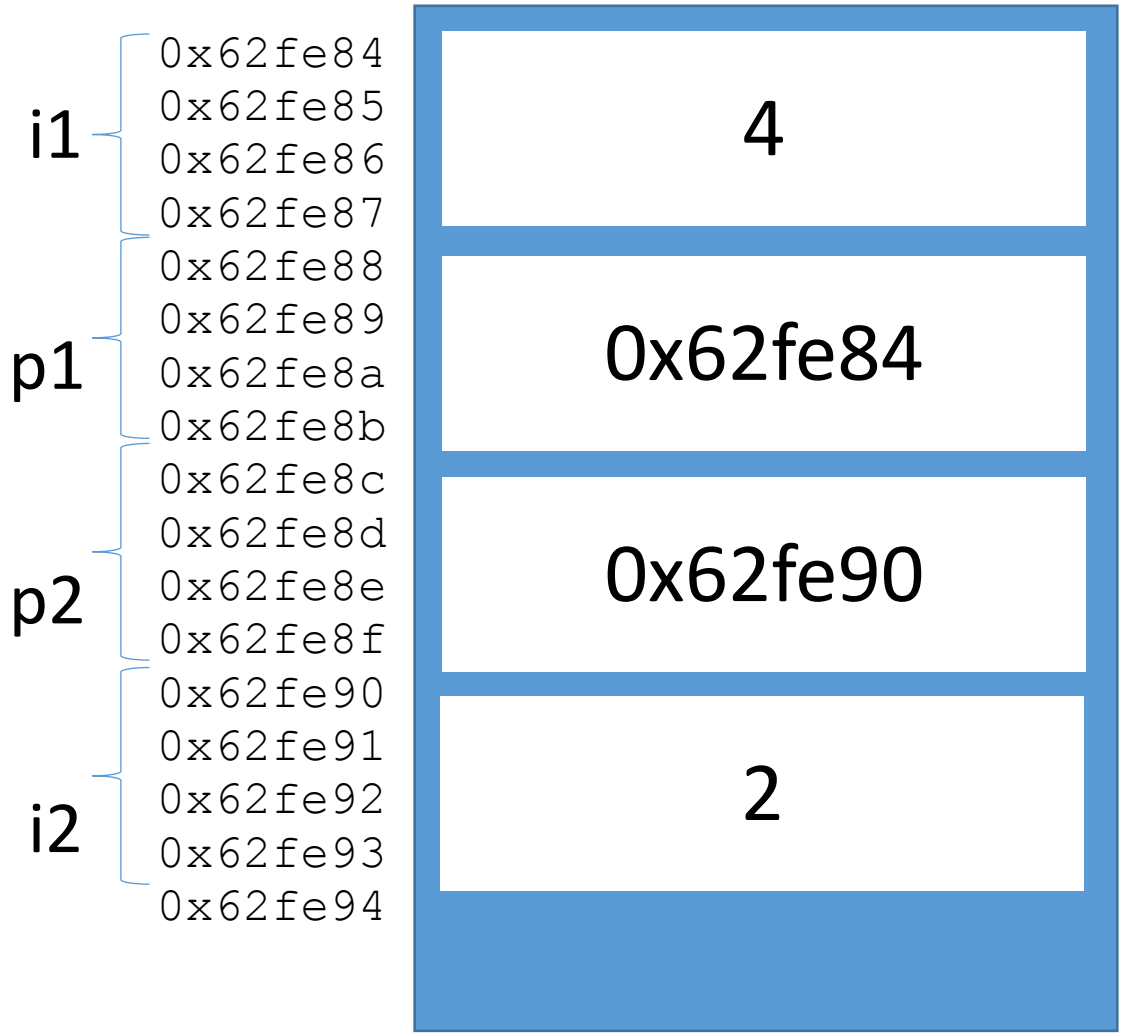
```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
    int  i2 = *p2+1;
    p2  = &i2;
    *p1 = square(*p2);
```

is dit geldig?



Lvalues vs Rvalues

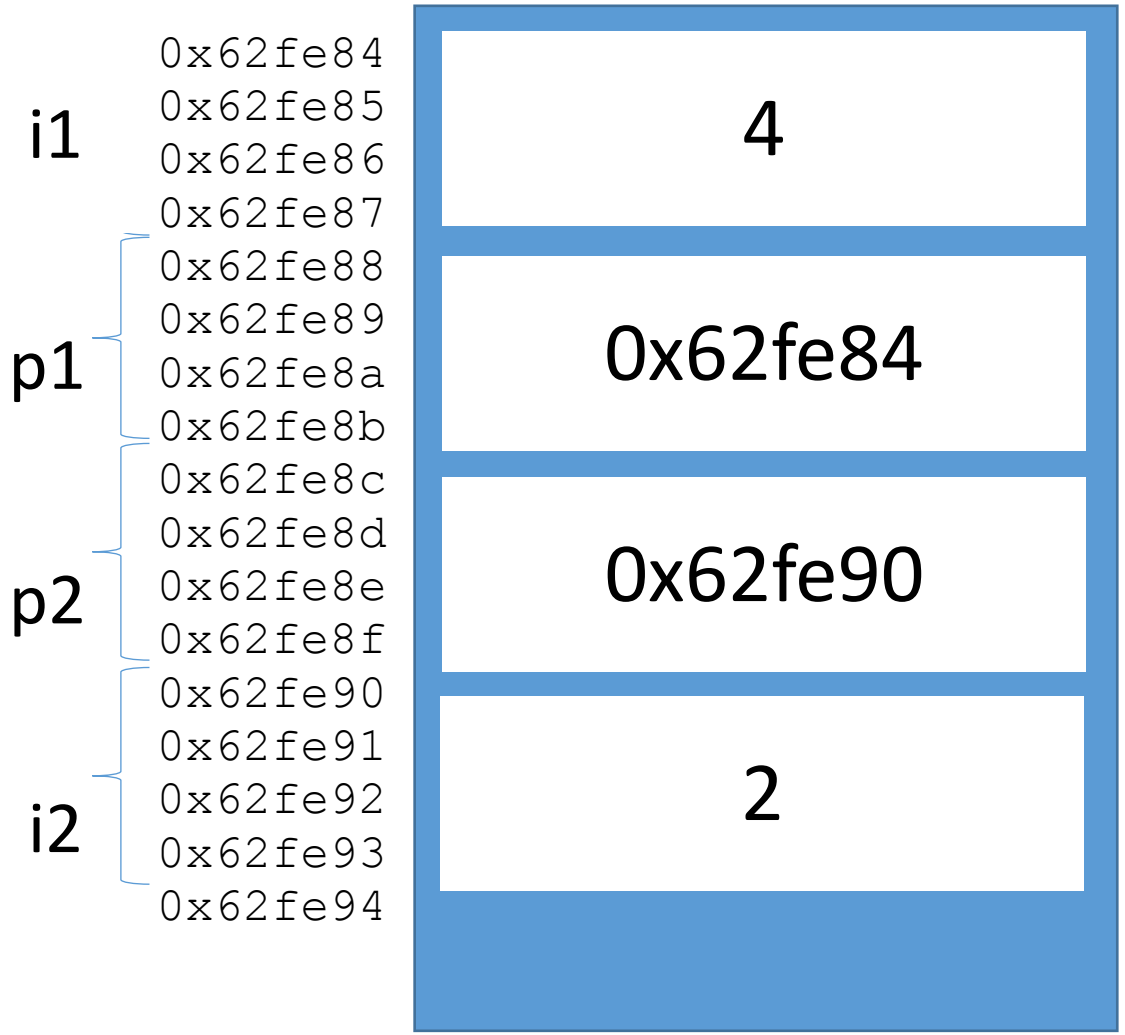
```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
    int  i2 = *p2+1;
    p2  = &i2;
    *p1 = square(*p2);
```



Lvalues vs Rvalues

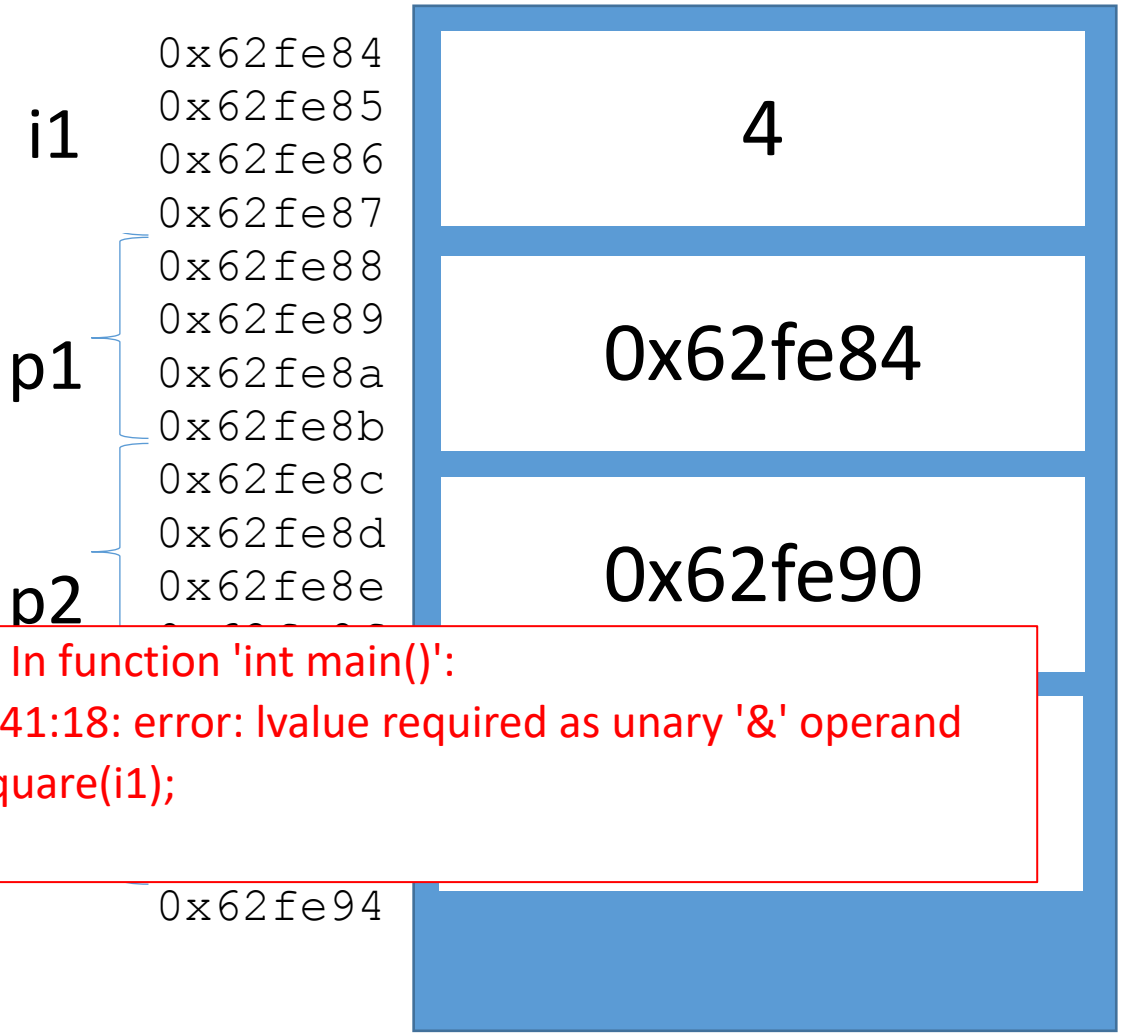
```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
    int  i2 = *p2+1;
    p2  = &i2;
    *p1 = square(*p2);
    p1  = &square(i1);
```

is dit geldig?



Lvalues vs Rvalues

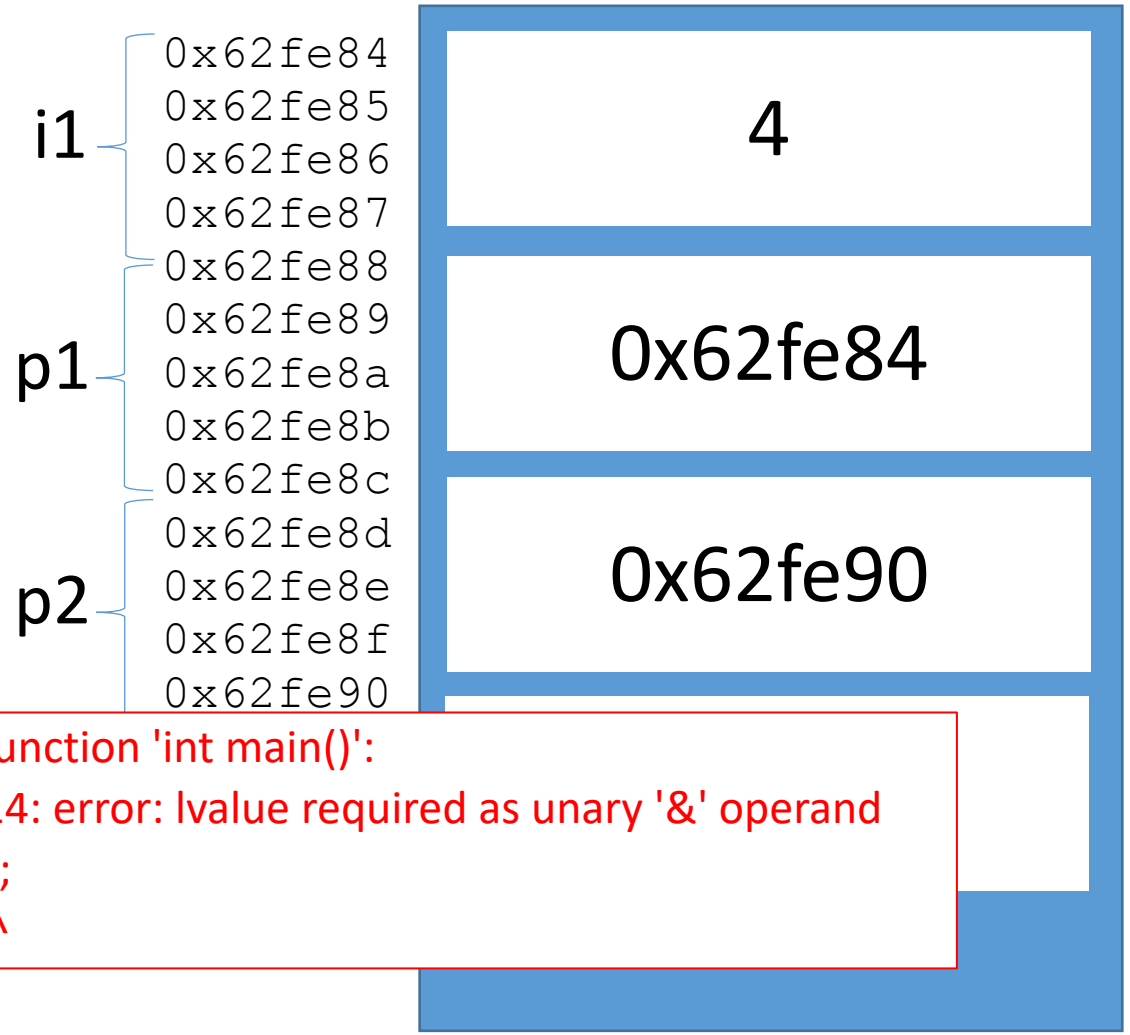
```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
    int  i2 = *p2+1;
    p2  = &i2;
    *p1 = square(*p2);
    p1  = &square(i1);
```




```
main.cpp: In function 'int main()':
main.cpp:41:18: error: lvalue required as unary '&' operand
   p1=&square(i1);
        ^
```

Lvalues vs Rvalues

```
int square(int x) {
    return x*x;
}

int main() {
    int  i1 = 1;
    int* p1 = &i1;
    int* p2 = p1;
    int  i2 = *p2+1;
    p2  = &i2;
    *p1 = square(*p2);
    //p1 = &square(i1
    p2  = &(i1+5);

    return 0;
}
```




```
main.cpp: In function 'int main()':
main.cpp:42:14: error: lvalue required as unary '&' operand
    p2=&(i1+5);
              ^
```

Vragen

- Wat is het resultaat van volgend stukje code?

```
int  i = 0;
int* p = &i;
i = (*p)+1;
cout << *p << "," << i << "," << p << endl;
```

Vragen

- Beschouw volgende variabele declaraties:

```
int v[5];
int i = 3;
```

Welke van volgende uitdrukkingen zijn lvalues?

(a) v[3] (b) v[3]+i (c) v[i] (d) v[i+1] (e) &i

Universiteit Antwerpen

Noot: Arrays en pointers

- Een array van grootte n bezet n naburige geheugenlocaties
- Een array variabele wordt in C++ gebruikt alsof het een constante pointer is (enkel te gebruiken als rvalue)

Voorbeeld:

```
int array[3]={0,1,2};
int* p = array;
*p=4;
cout << array[0] << " " << array[1] << " " << array[2] << endl;
```

```
4 1 2
```

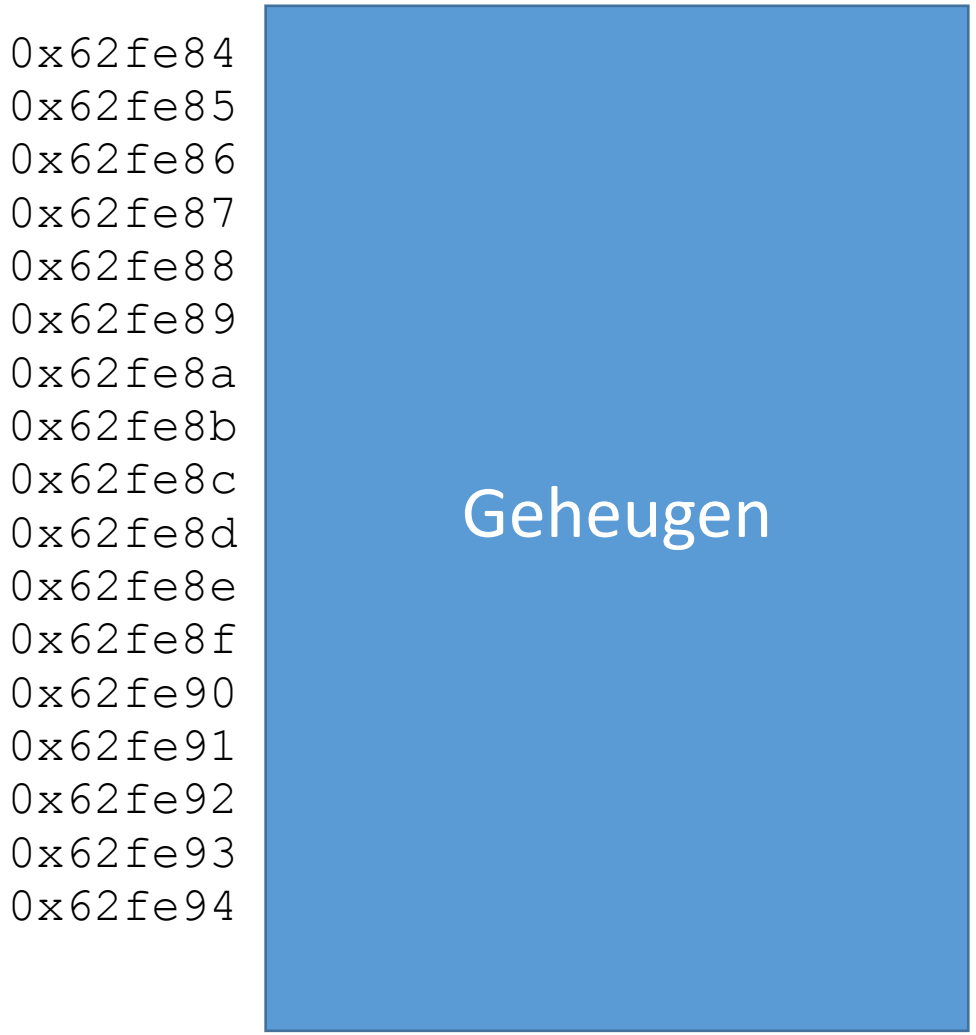
Rekenen met pointers

- Net zoals met *iterators* kunnen we ook met pointers “rekenen”

```
int  v[3];
int* p = v;
*p = 1;
p++;
*p = 2;
p++;
*p = 3;
cout << v[0] << " " << v[1] << " " << v[2]
<< endl;
```

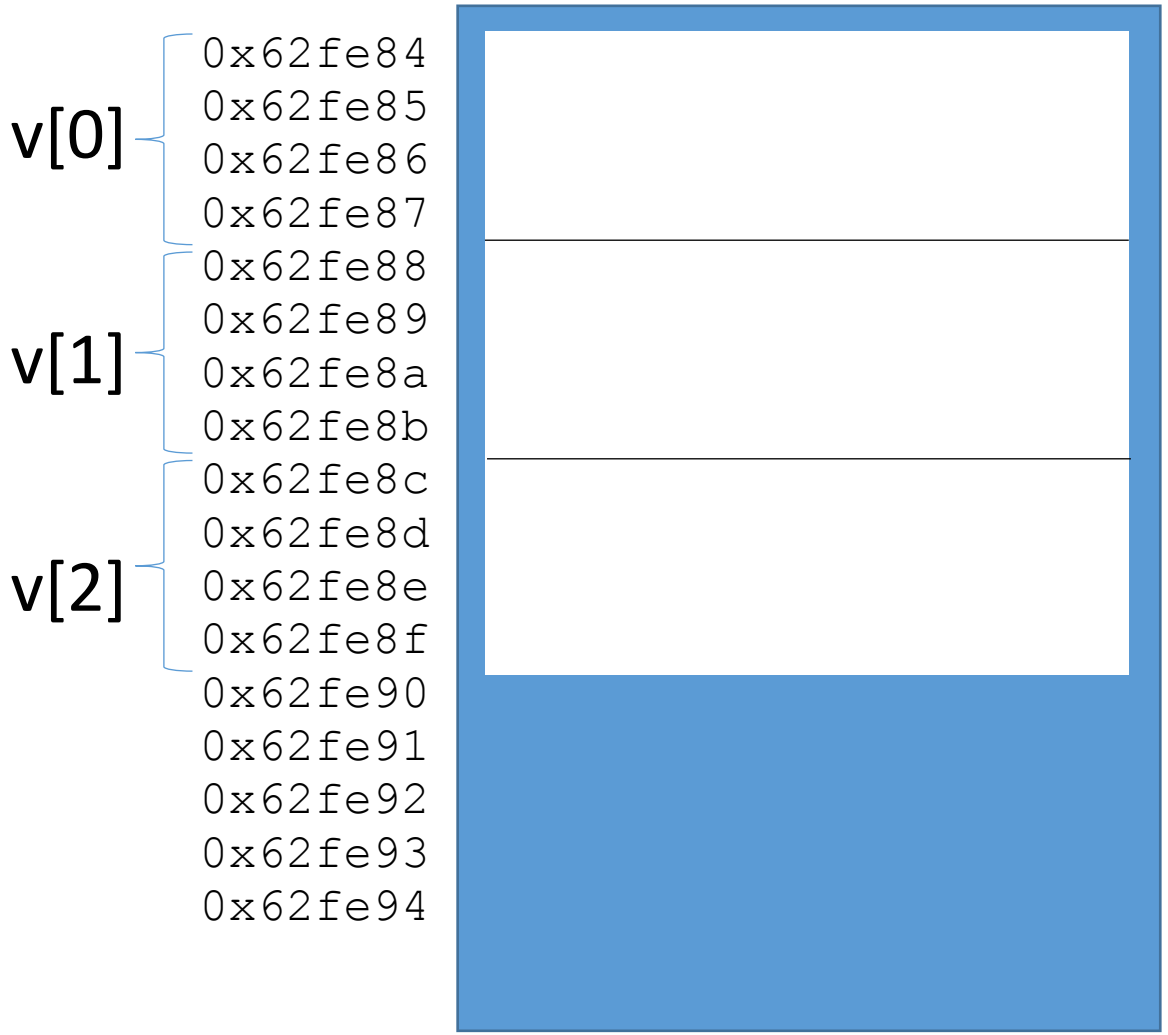
1 2 3

Rekenen met pointers



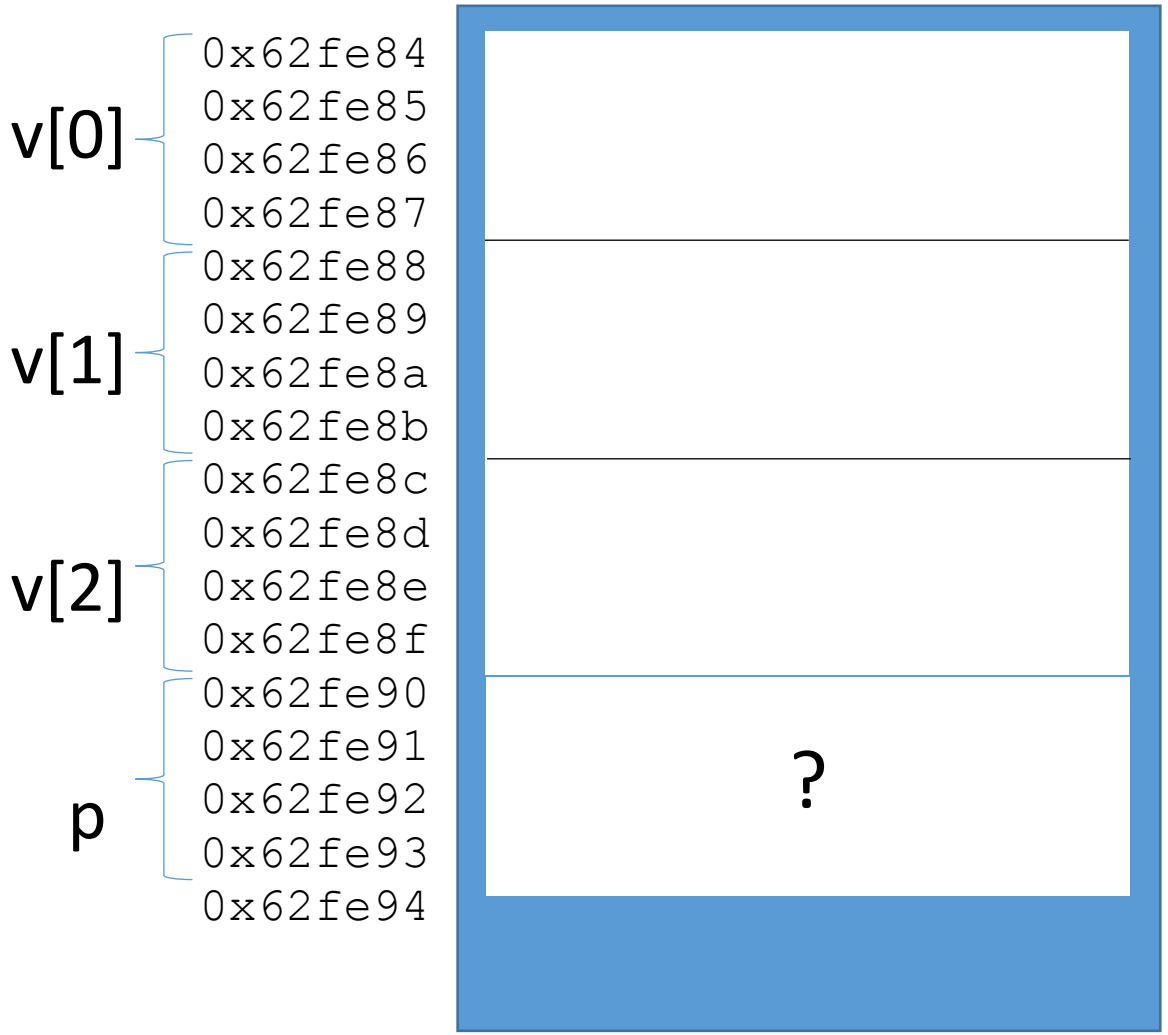
Rekenen met pointers

```
int  v[3];
```



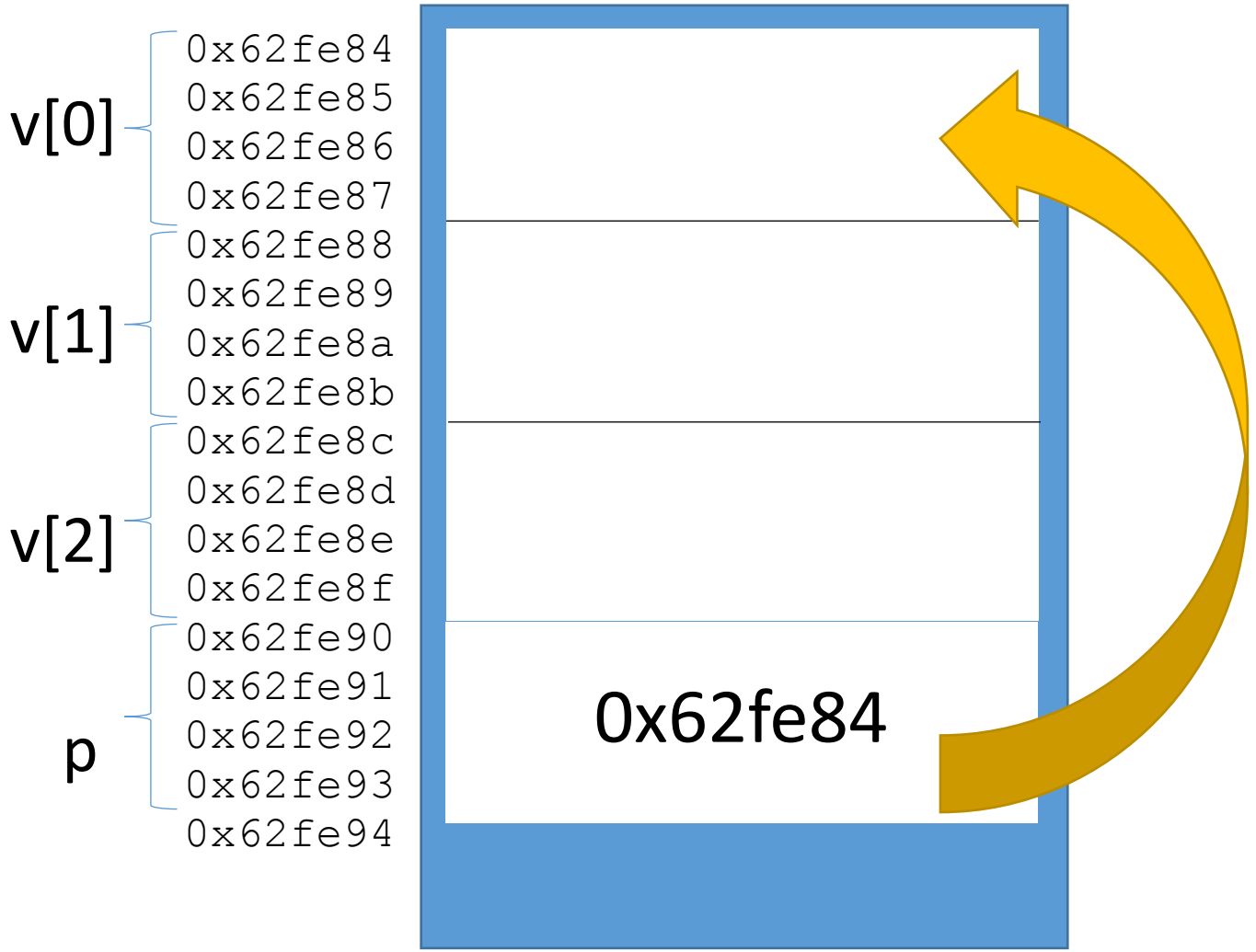
Rekenen met pointers

```
int  v[3];
int* p = v;
```



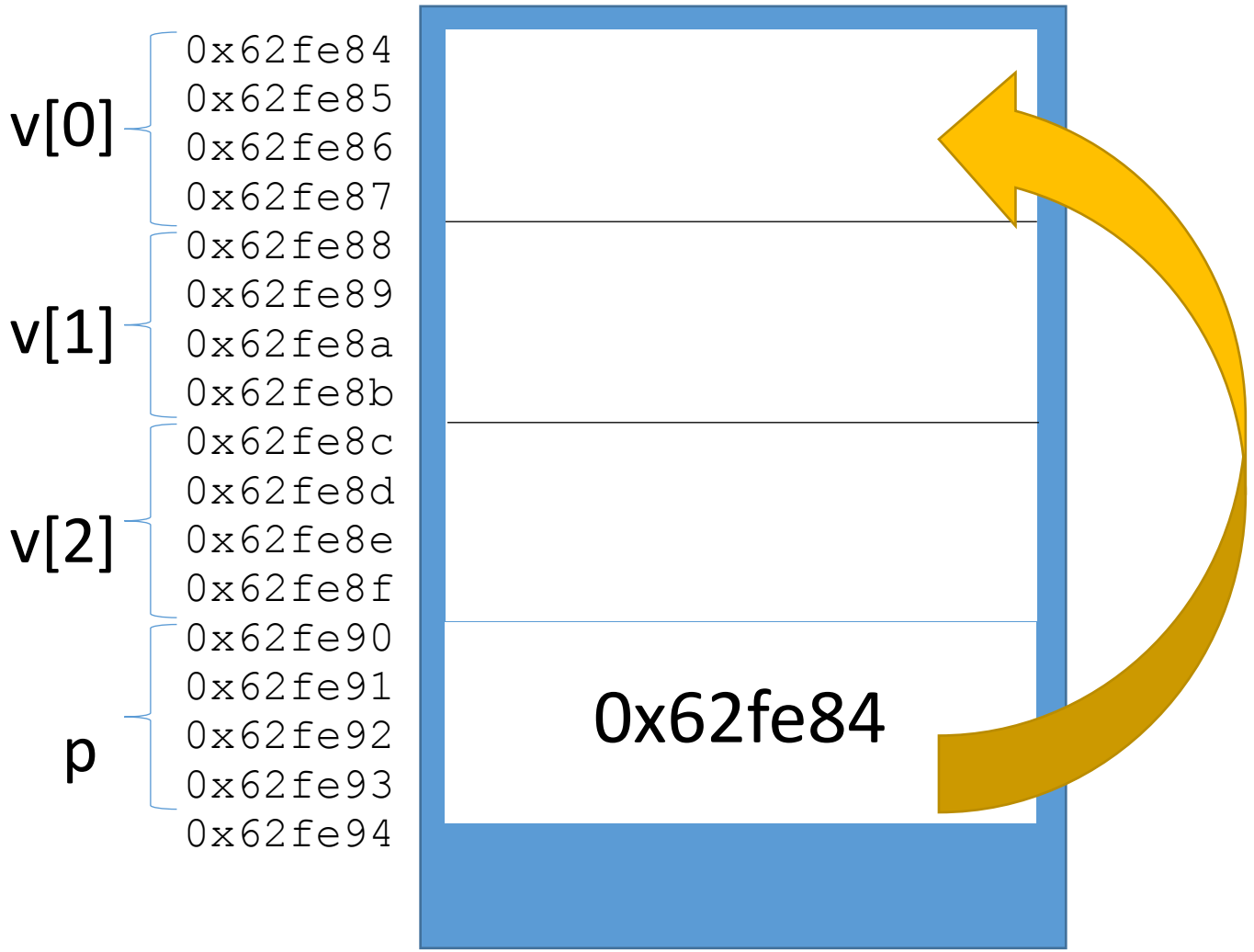
Rekenen met pointers

```
int  v[3];
int* p = v;
```



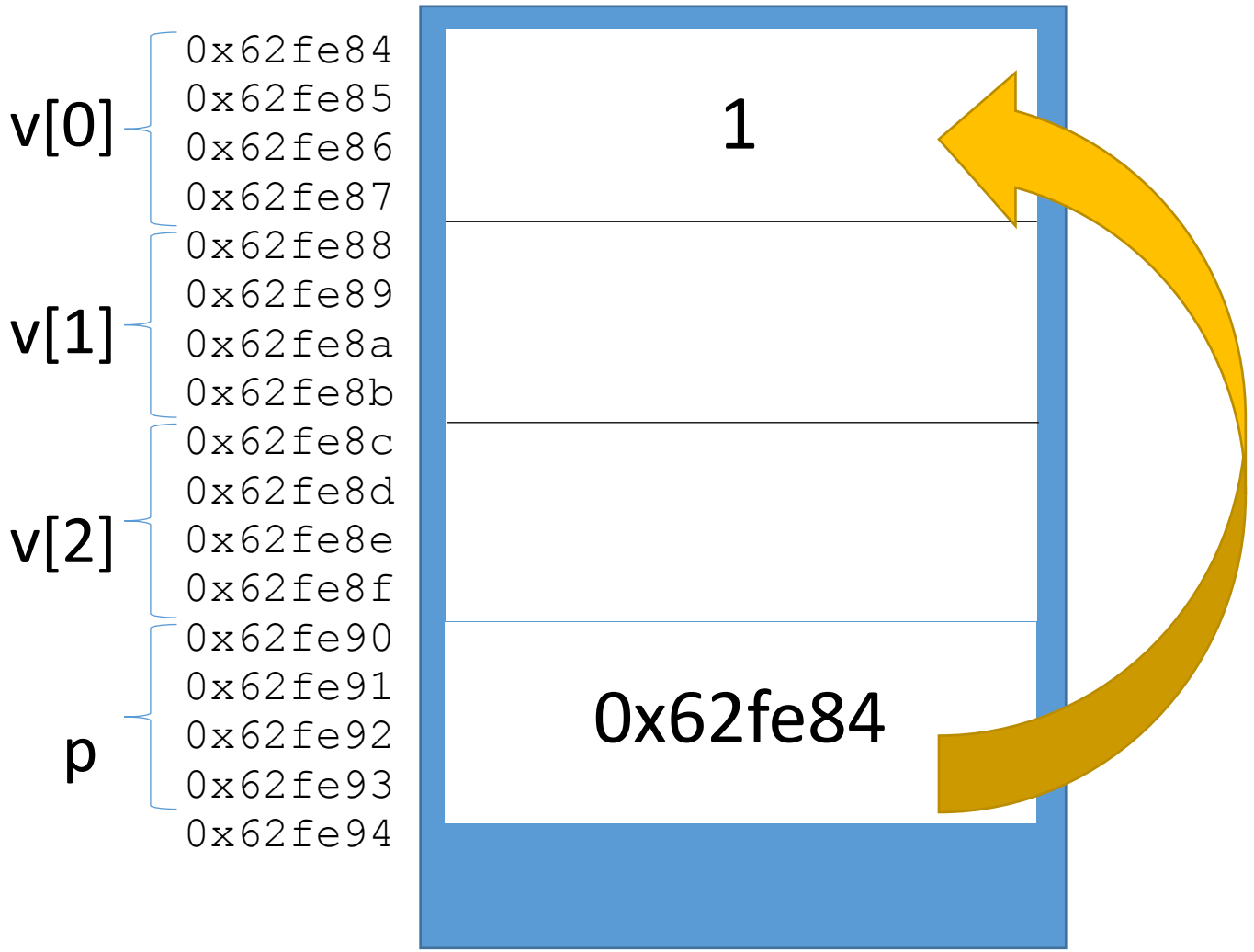
Rekenen met pointers

```
int  v[3];
int* p = v;
*p = 1;
```



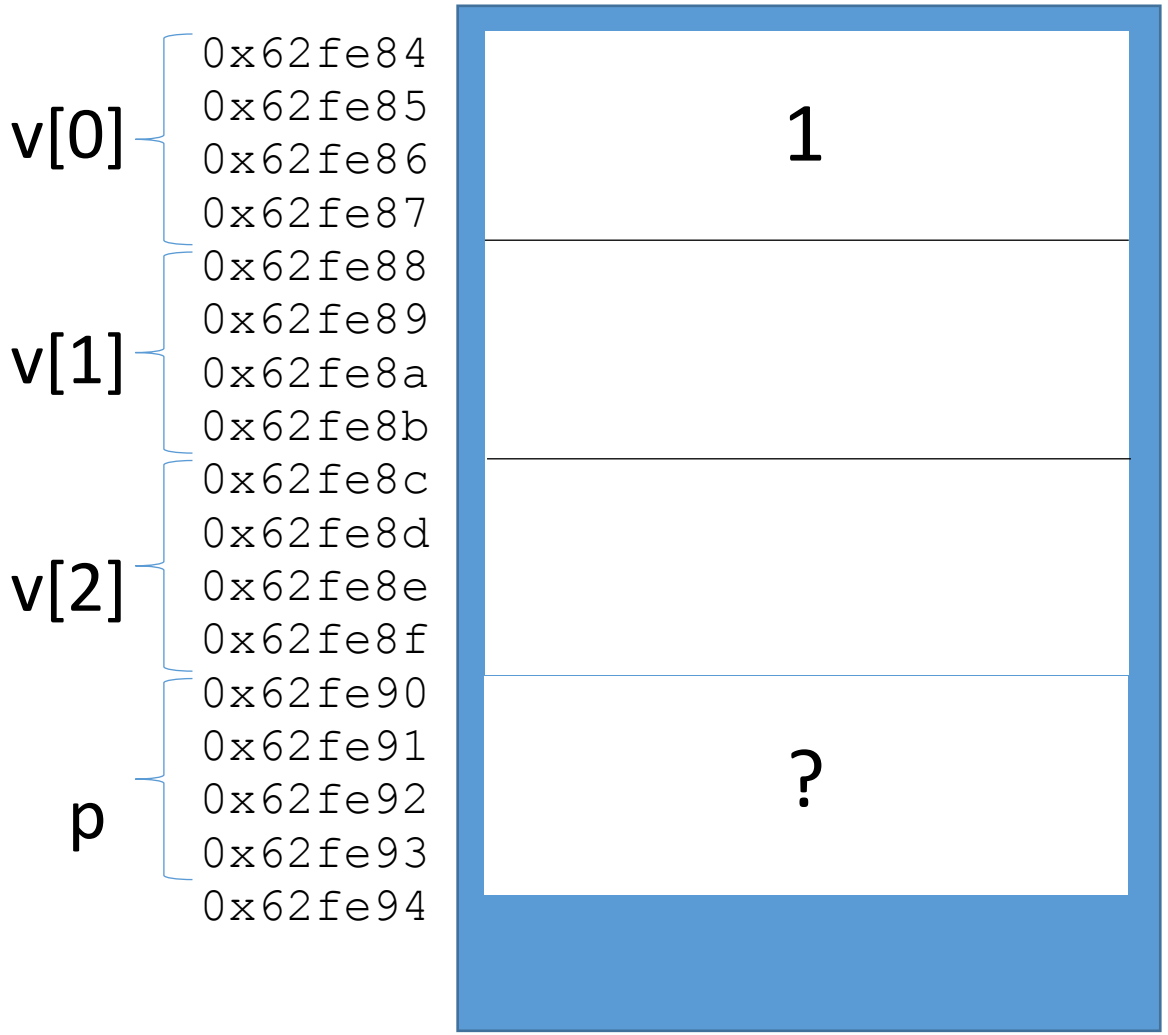
Rekenen met pointers

```
int  v[3];
int* p = v;
*p = 1;
```



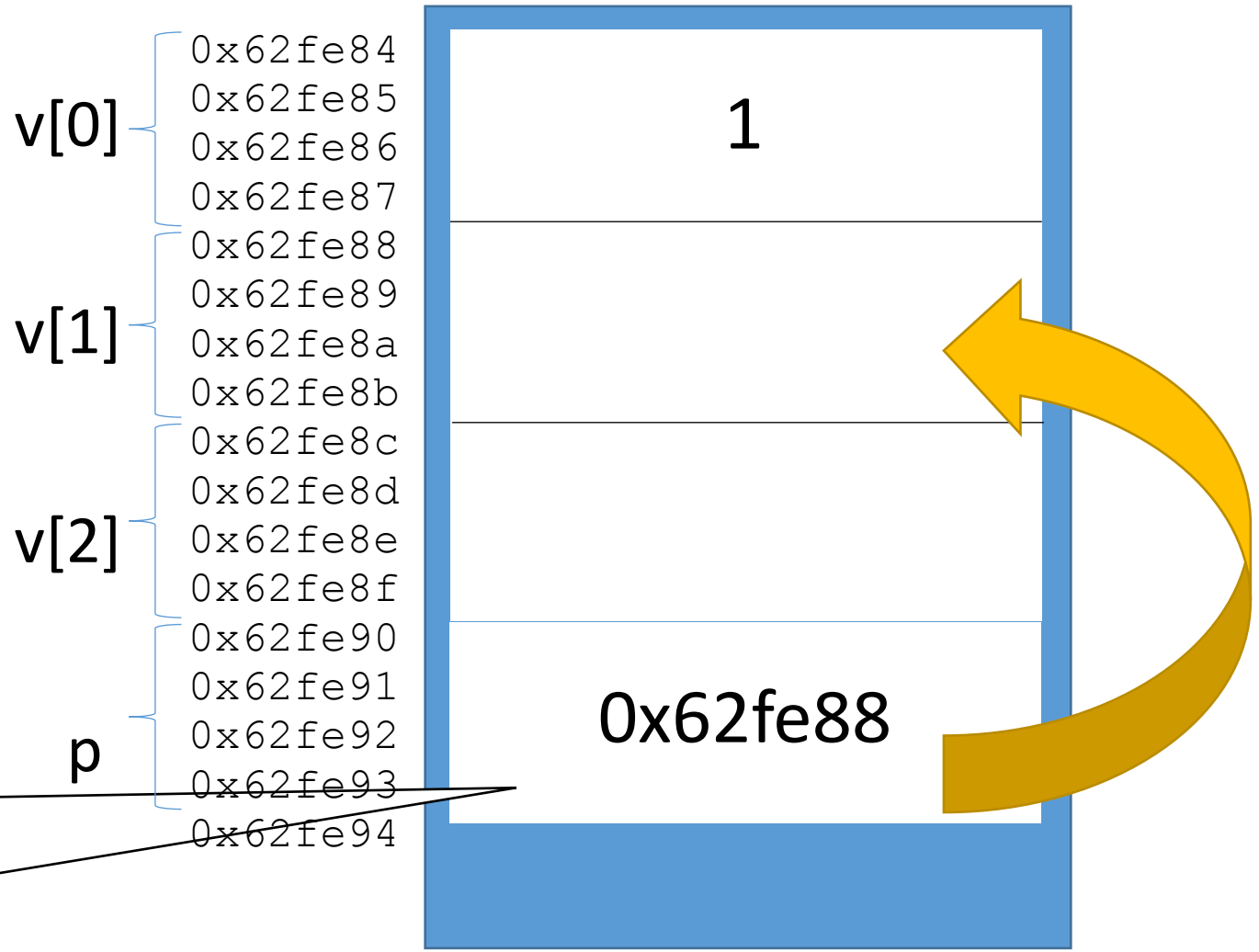
Rekenen met pointers

```
int  v[3];
int* p = v;
*p = 1;
p++;
```



Rekenen met pointers

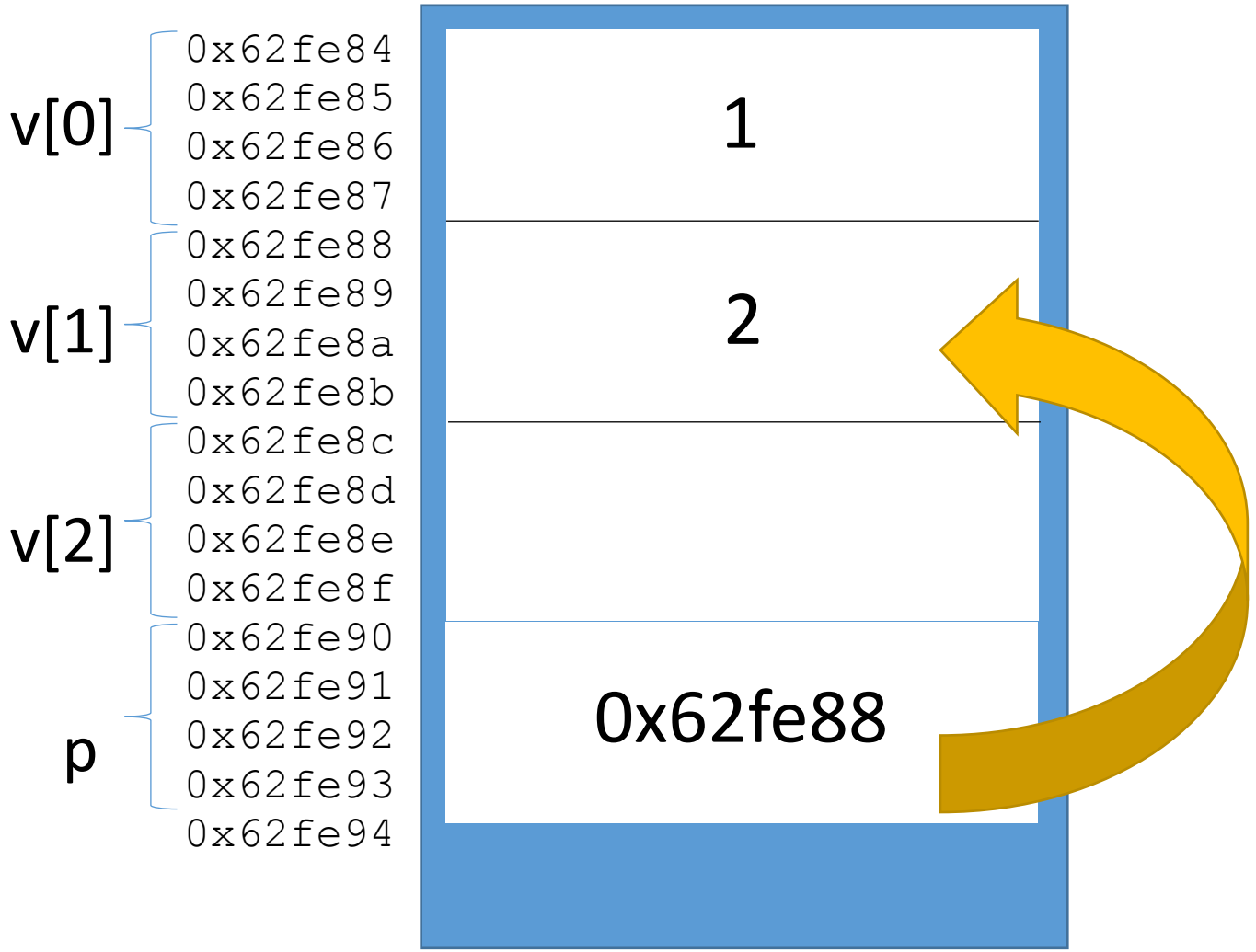
```
int  v[3];
int* p = v;
*p = 1;
p++;
```



p++ verhoogt p met sizeof(type van *p); in dit geval dus met sizeof(int)!

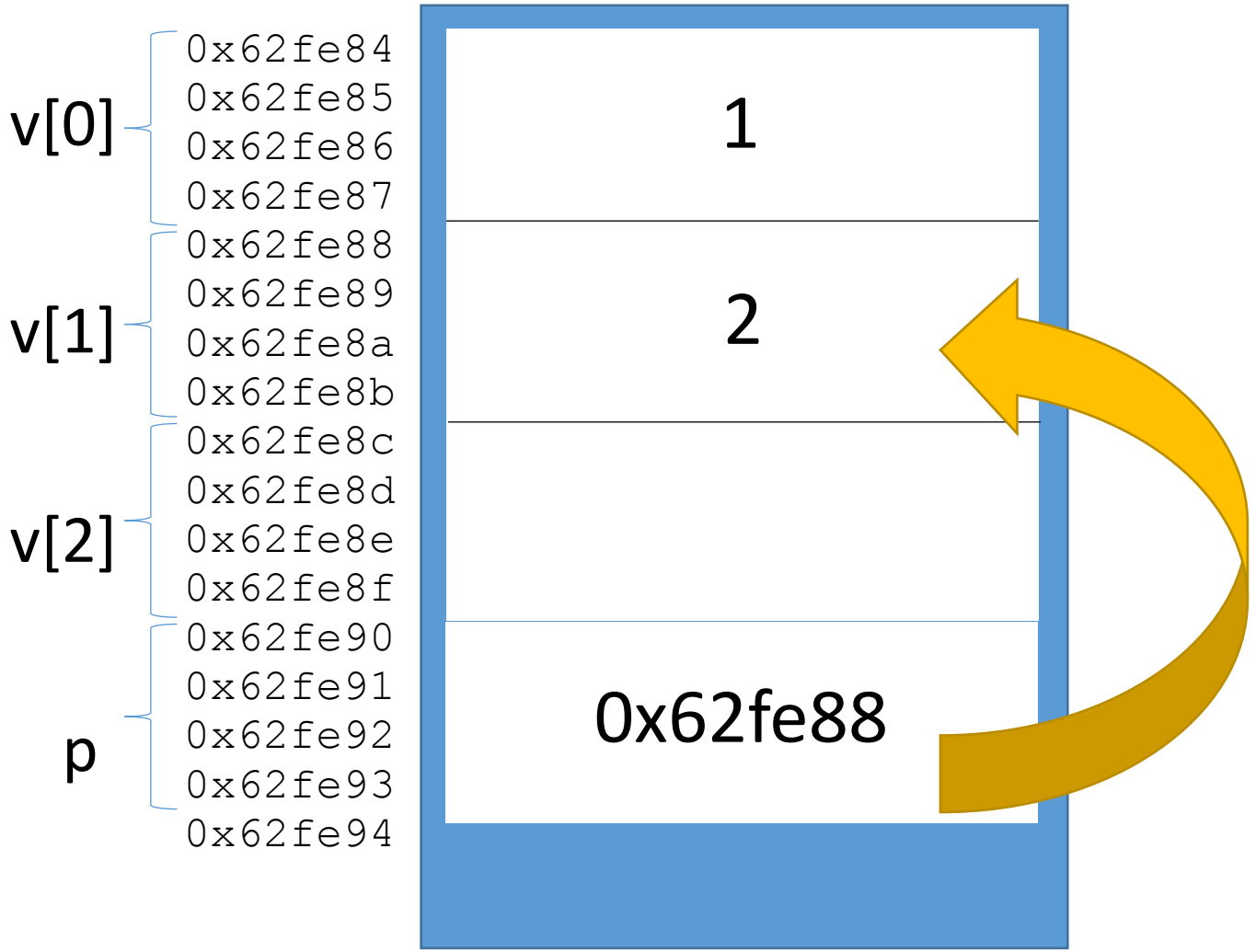
Rekenen met pointers

```
int  v[3];
int* p = v;
*p = 1;
p++;
*p = 2;
```



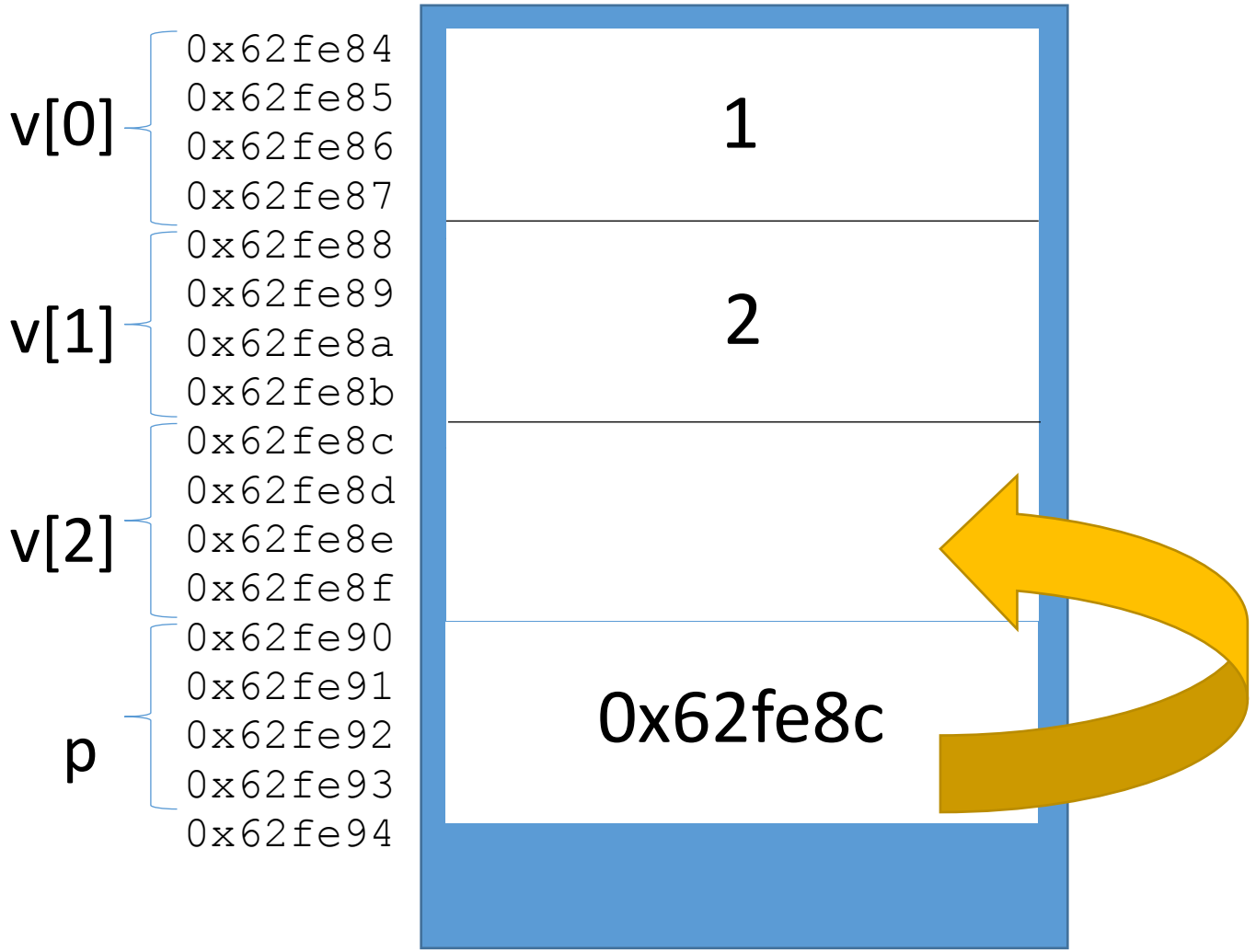
Rekenen met pointers

```
int  v[3];
int* p = v;
*p = 1;
p++;
*p = 2;
```



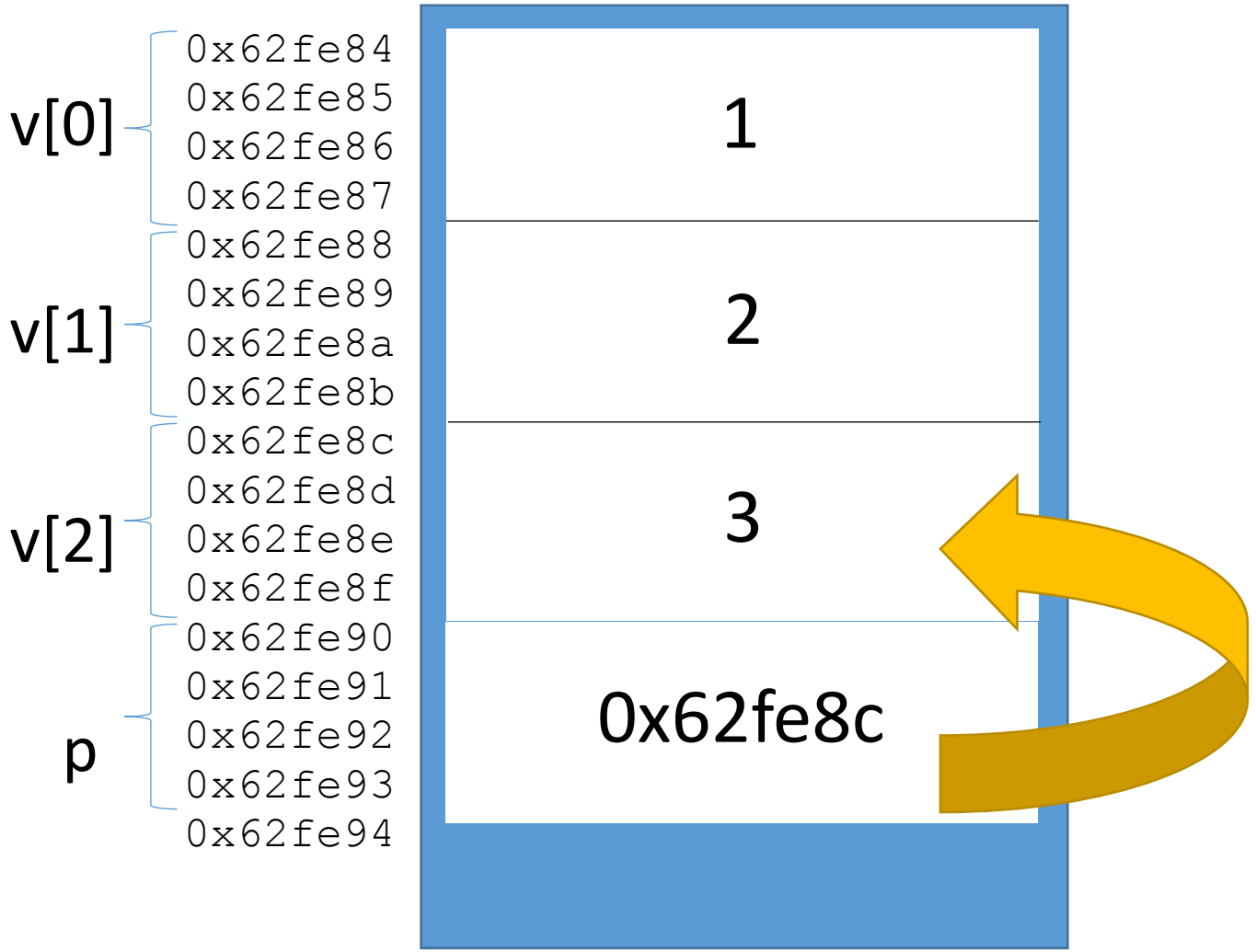
Rekenen met pointers

```
int  v[3];
int* p = v;
*p = 1;
p++;
*p = 2;
p++;
```



Rekenen met pointers

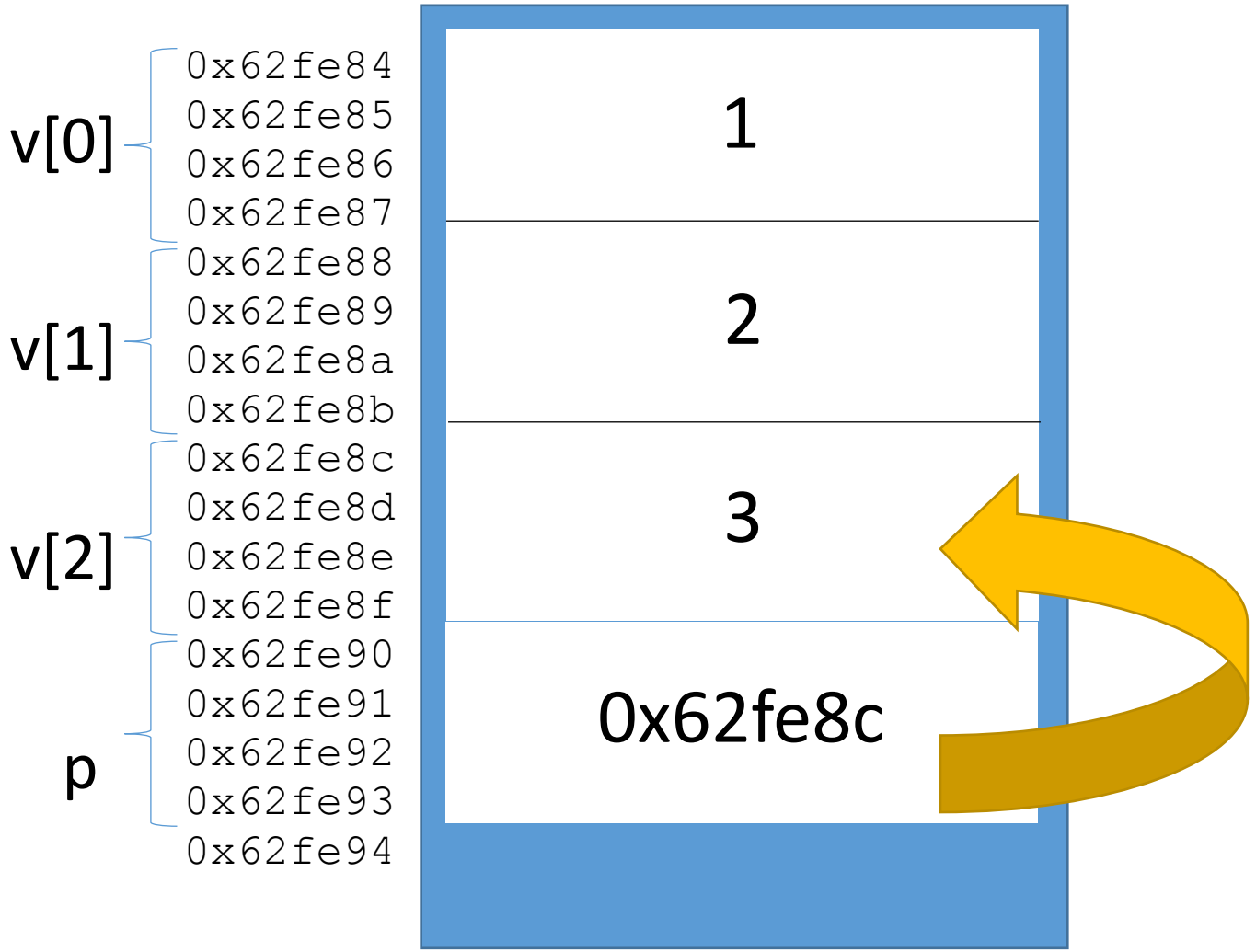
```
int  v[3];
int* p = v;
*p = 1;
p++;
*p = 2;
p++;
*p = 3;
```



Rekenen met pointers

```
int  v[3];
int* p = v;
*p = 1;
p++;
*p = 2;
p++;
*p = 3;
cout << v[0] << " " <<
v[1]
<< " " << v[2] << endl;
```

1 2 3



Rekenen met pointers

- Let op want dit is heel verwarrend: array IS een pointer en omgekeerd!

```
int array[3];

// met array notatie
array[1] = 5;

// met pointer notatie
*(array+1) = 5;
```

Rekenen met pointers

- Let op want dit is heel verwarrend!

```
int array[3];
int *p = array;

// met array notatie
p[1] = 5;

// met pointer notatie
*(p+1) = 5;
```

Rekenen met pointers

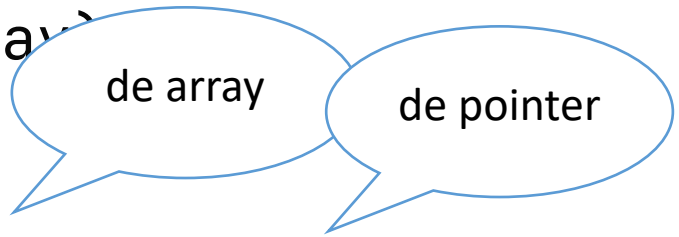
- Bij functies nog lastiger:

```
void print(int a[]){
   cout << sizeof(a) << endl;
}

int main(){
   int array[3];
   cout << sizeof(array) << endl;
   print(array);
}

// output?
```

Rekenen met pointers

- Bij functies nog lastiger:

```
void print(int a[]){
   cout << sizeof(a) << endl;
}

int main(){
   int array[3];
   cout << sizeof(array) << endl;
   print(array);
}

// geeft: 12 en dan 4
```



Rekenen met pointers

- Je had dus ook dit kunnen schrijven:

```
void print(int* a){
   cout << sizeof(a) << endl;
}

int main(){
   int array[3];
   cout << sizeof(array) << endl;
   print(array);
}

// geeft: 12 en dan 4
```

Rekenen met pointers

- Gevolg: bij arrays moet je de size meegeven

```
void print(int a[], int size){
   for(int i = 0; i < size; i++)
      cout << a[i] << endl;
}

int main(){
   int array[3];
   print(array);
}
```

Universiteit Antwerpen

Rekenen met pointers

- Andere operaties op pointers van type T* :
  - p++ : verhoog adres met sizeof(T)
  - p-- : verlaag adres met sizeof(T)
  - p += 5; of p=p+5; : verhoog adres met 5*sizeof(T)
  - p -= 3; of p=p-3; : verlaag adres met 3*sizeof(T)
  - *(p+4) = c ; : zet c op geheugenplaats p + 4*sizeof(T)

Noot:

- voor een statische array v wordt intern v[3] herschreven als *(v+3).
- Je kan bovendien de notatie p[i] gebruiken als alternatief voor *(p+i).

Rekenen met pointers

```
int v[3];
int* p=v;
*p = 1;
*(p+1) = 2;
*(p+2) = 3;
cout << v[0] << " " <<
v[1]
          << " " << v[2]
<< endl;
```

=

```
int v[3];
int* p=v;
p[0] = 1;
p[1] = 2;
p[2] = 3;
cout << v[0] << " "
<< v[1]
          << " " <<
v[2] << endl;
```

Universiteit Antwerpen

Deze les

- Pointers
  - Variabelen als alias voor geheugenlocaties
  - Pointer type
  - Referencing en dereferencing
  - Rekenen met pointers
- Referentie variabelen
- Parameters bij functieaanroepen
  - Call by value / Call by reference
- Call stack

Bekijk ook volgende uitstekende tutorial:

- http://www.cplusplus.com/doc/tutorial/pointers/

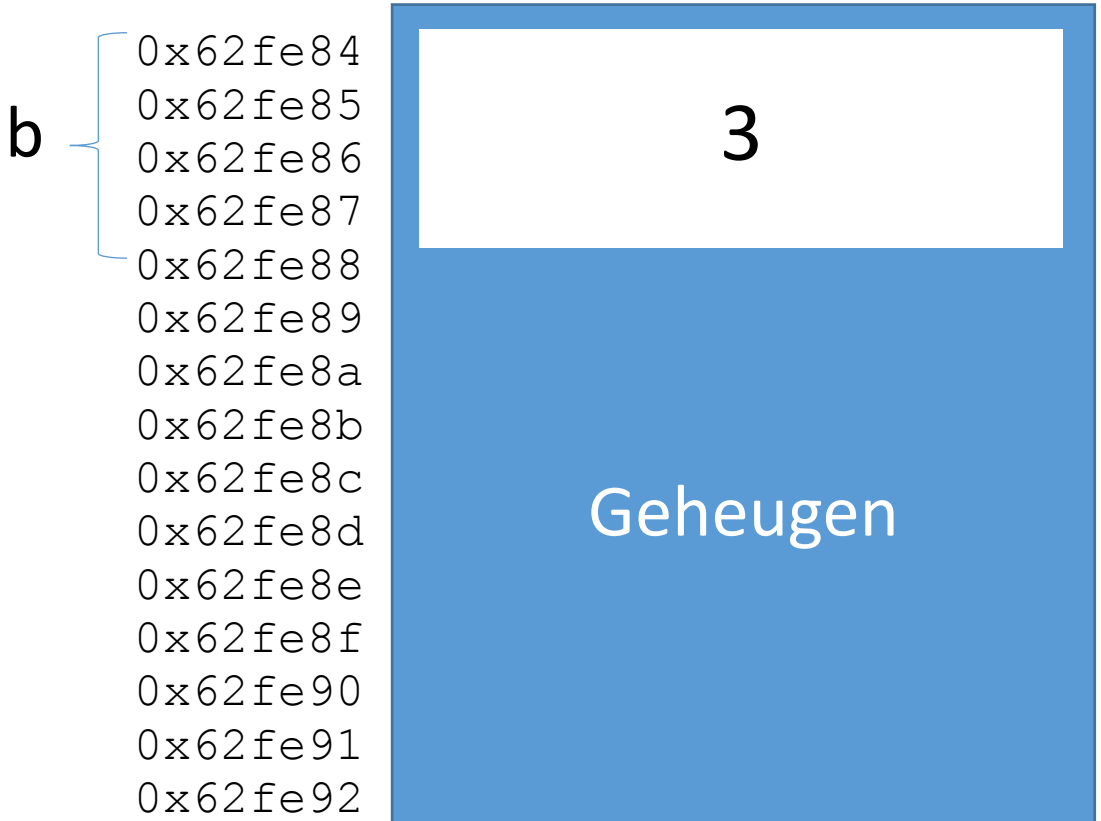
Referentie variabelen

- Als y van type T is, dan kunnen we een tweede variabele naar dezelfde geheugenplaats laten verwijzen: via de constructie:

  T& x = y;

Bijvoorbeeld:

int b = 3;



Referentie variabelen

- Als y van type T is, dan kunnen we een tweede variabele naar dezelfde geheugenplaats laten verwijzen: via de constructie:

  T& x = y;

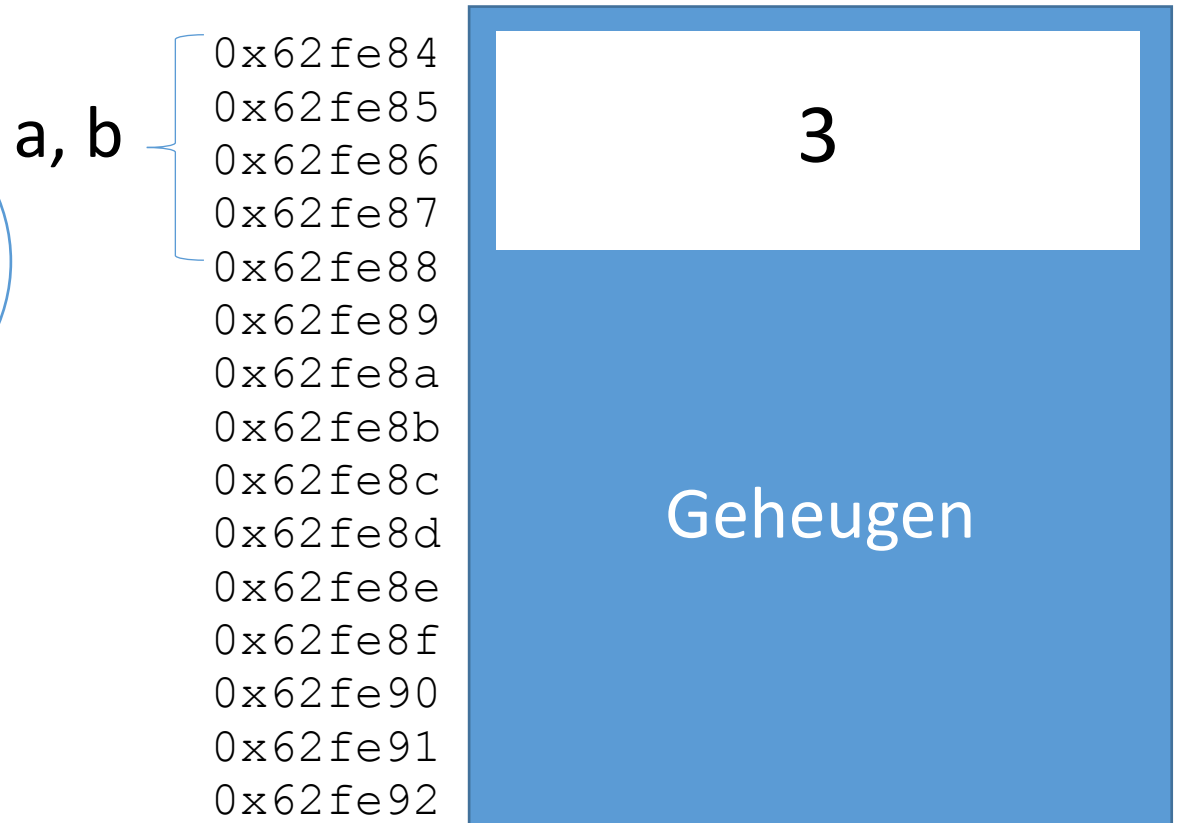
Bijvoorbeeld:

```
int b = 3;
int &a = b;
```

moet je initialiseren want kan later niet gewijzigd worden

a en b zijn aliassen of 'synoniemen'!

Let op: & is een deel van het type zoals *



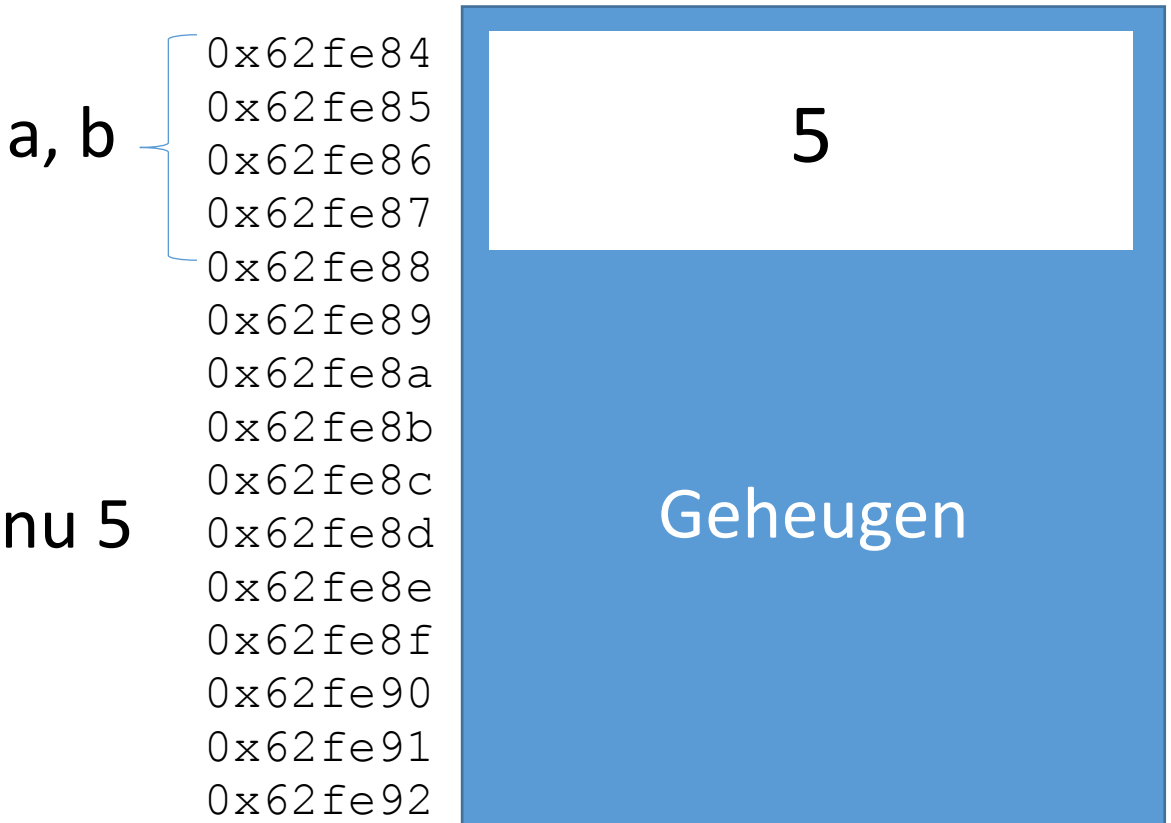
Referentie variabelen

- Als y van type T is, dan kunnen we een tweede variabele naar dezelfde geheugenplaats laten verwijzen: via de constructie:

  T& x = y;

Bijvoorbeeld:

```
int b = 3;
int &a = b;
b = 5;          // Zowel a als b zijn nu 5
```



Referentie variabelen

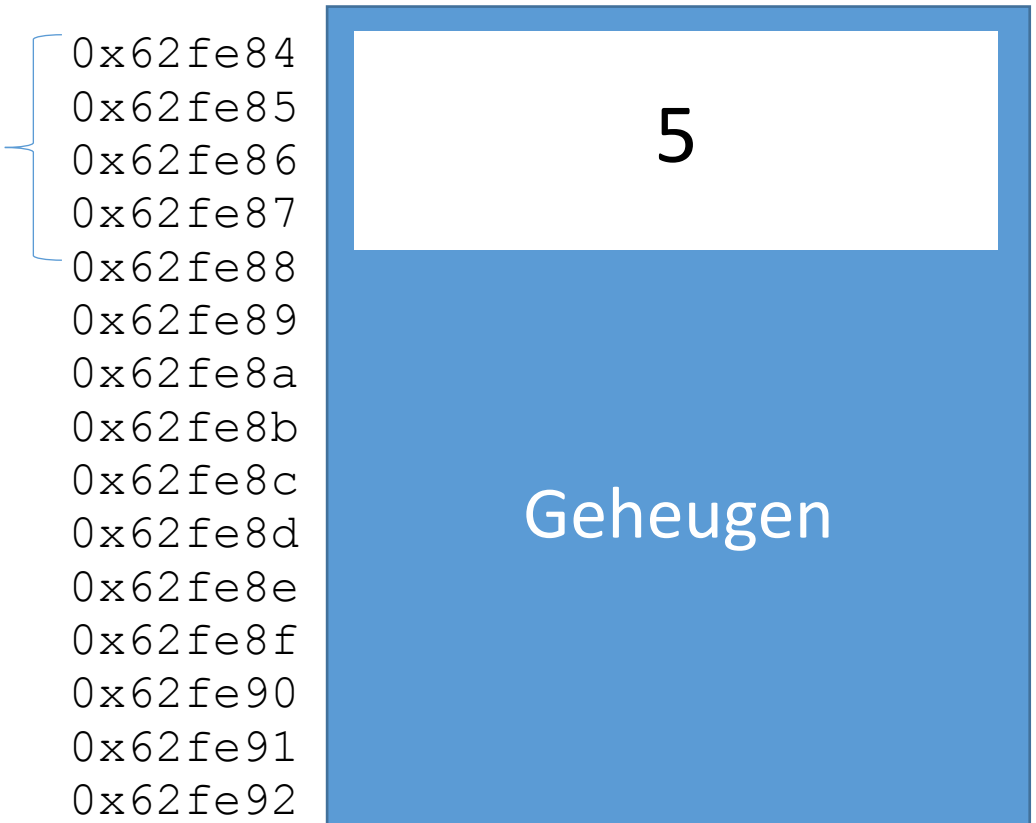
- Als y van type T is, dan kunnen we een tweede variabele naar dezelfde geheugenplaats laten verwijzen: via de constructie:

  T& x = y;

Bijvoorbeeld:

```
int b = 3;
int &a = b;
b = 5;            // Zowel a als b zijn nu 5
cout << &a << " " << &b;
```

```
0x62fe84 0x62fe84
```



Referentie variabelen

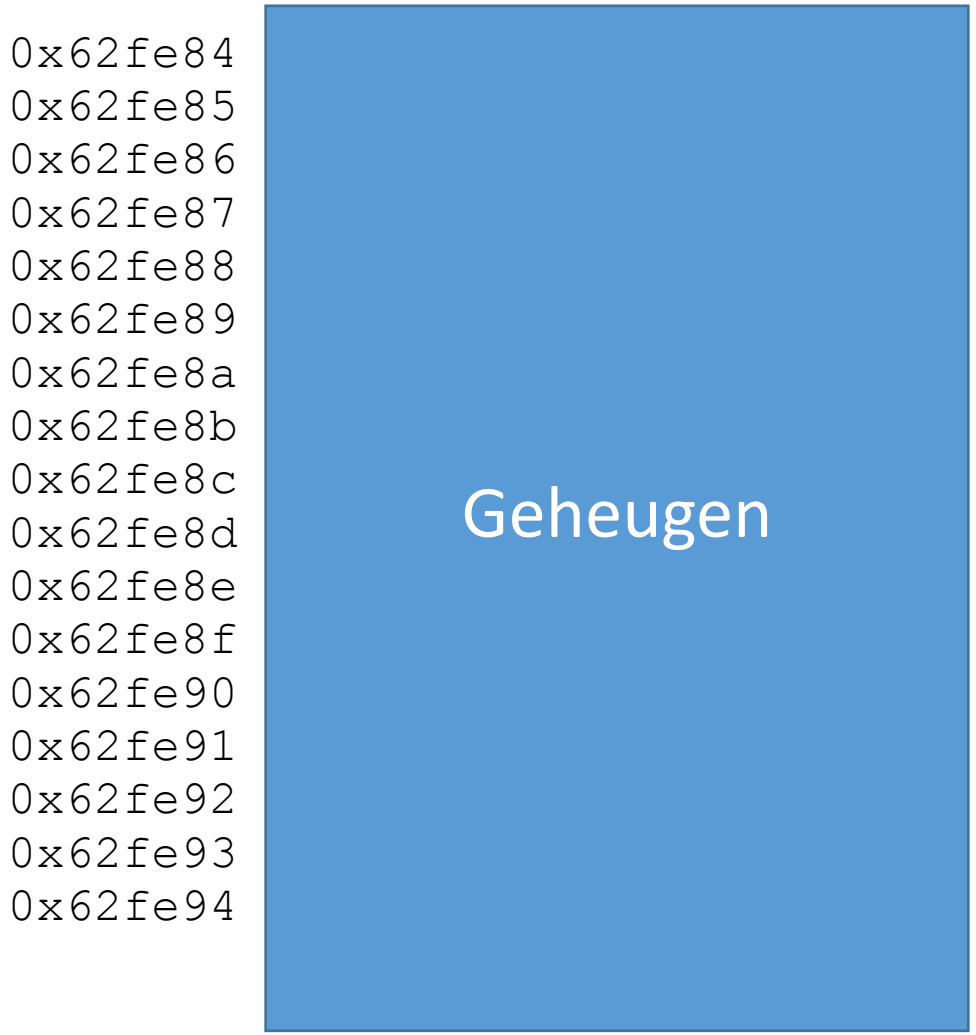
- Net zoals een pointer kan een referentie variabele aan elke lvalue van het juiste type gelijk gesteld worden:

```
int a[3] = {0, 1, 2};
int &i1 = a[1]; // hoeveel geheugenplaatsen zijn er in gebruik?
```

Referentie variabelen

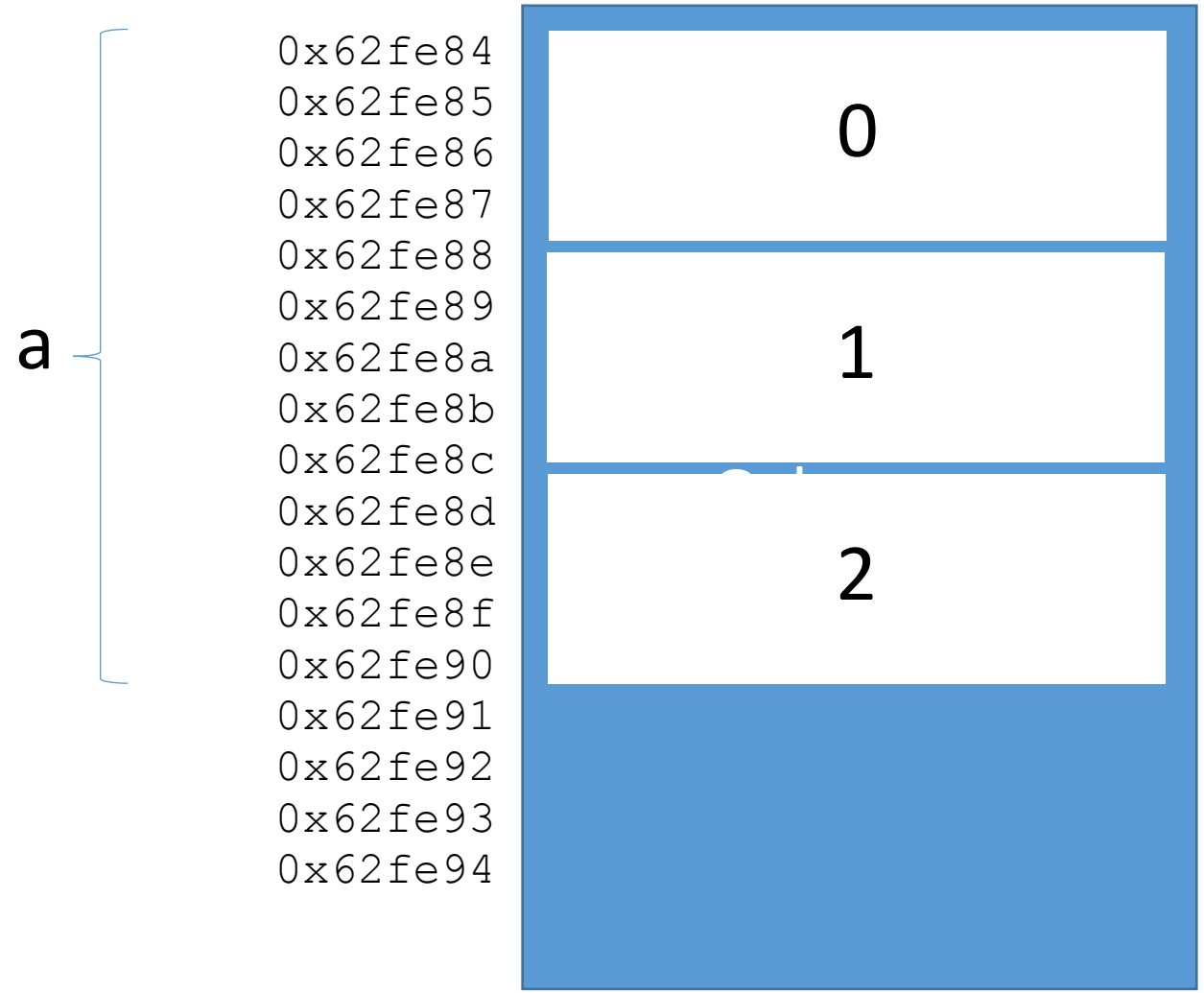
- Net zoals een pointer kan een referentie variabele aan elke lvalue van het juiste type gelijk gesteld worden:

Referentie variabelen



Referentie variabelen

```
int a[3] = {0, 1, 2};
```

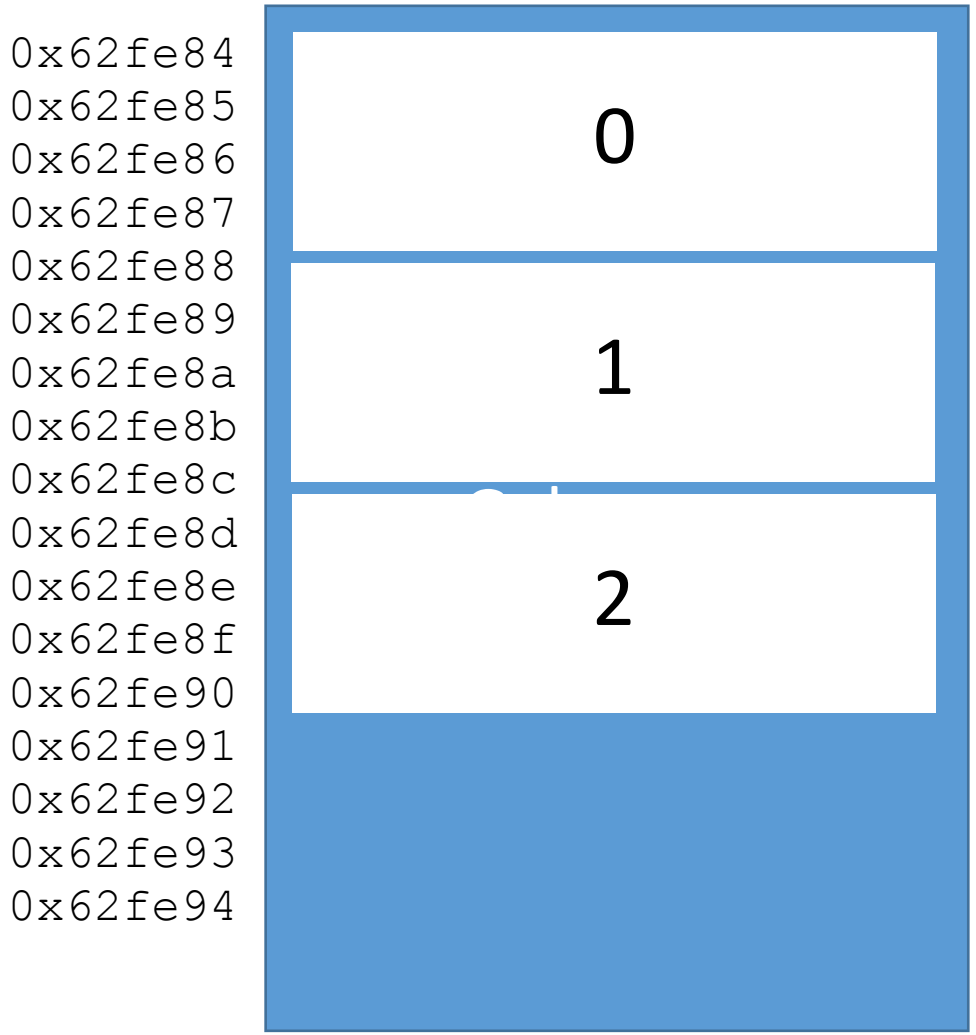


Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
```

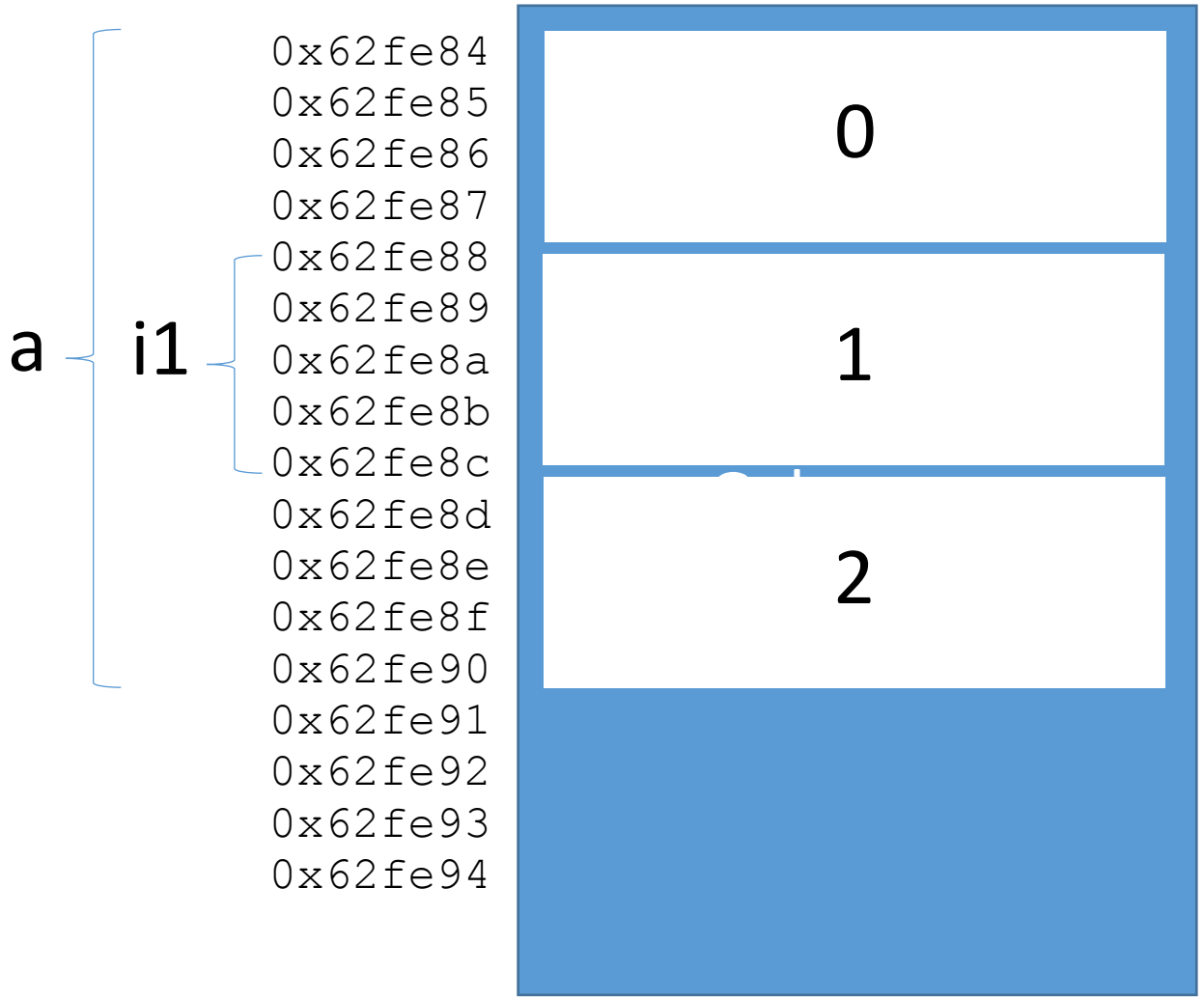
Hoeveel geheugenplaatsen zijn er nu in gebruik?

a



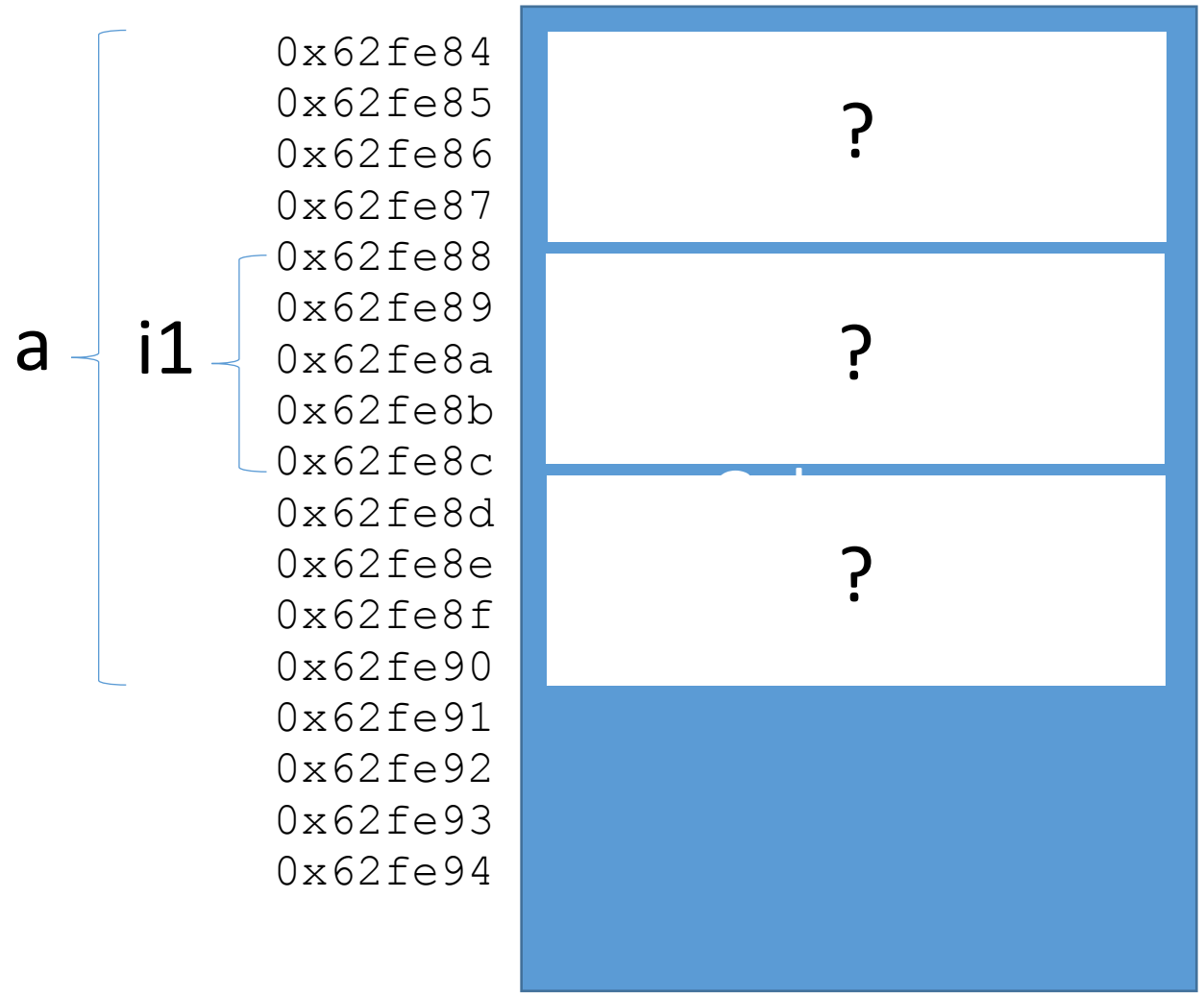
Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
```



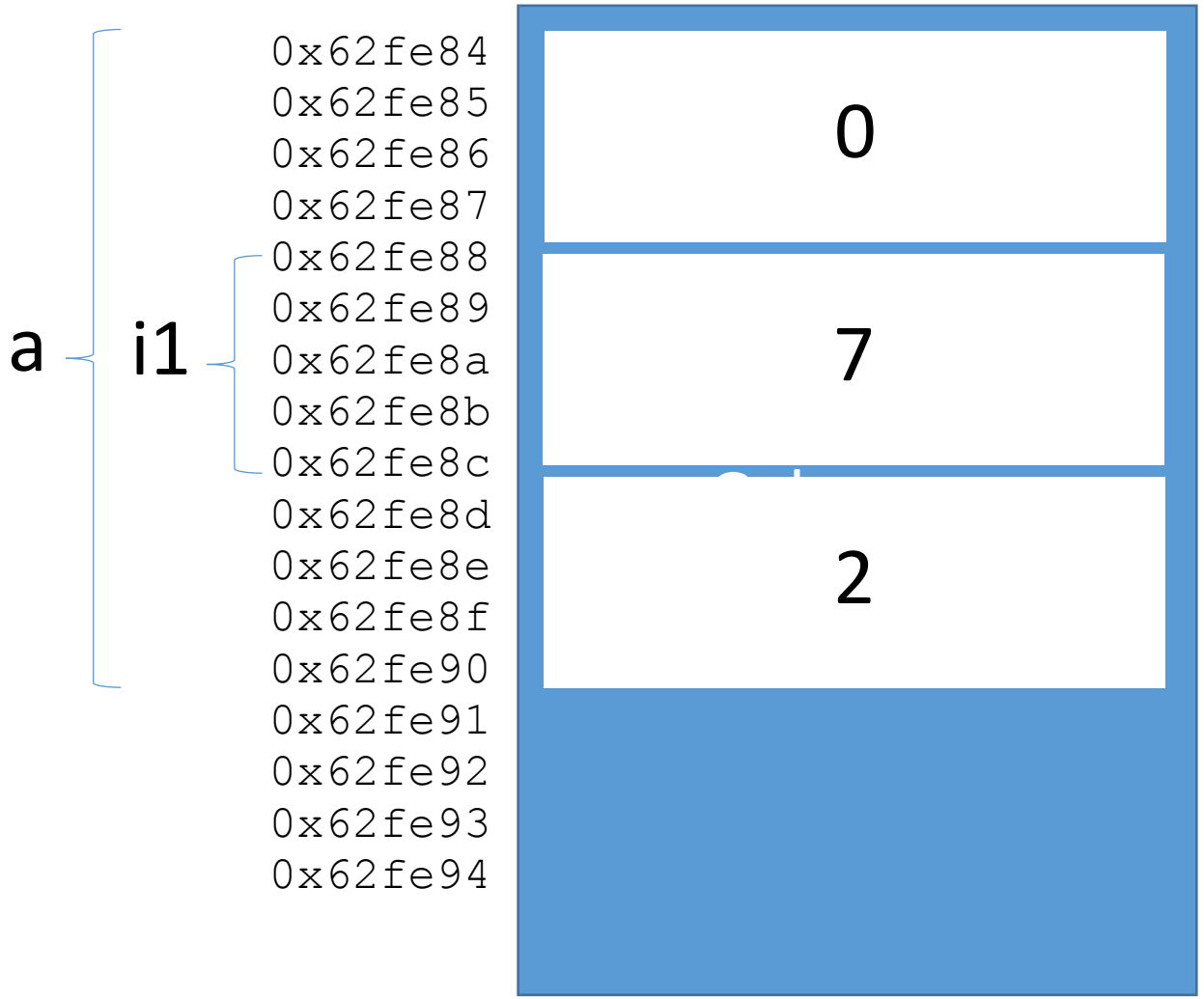
Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
```



Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
```



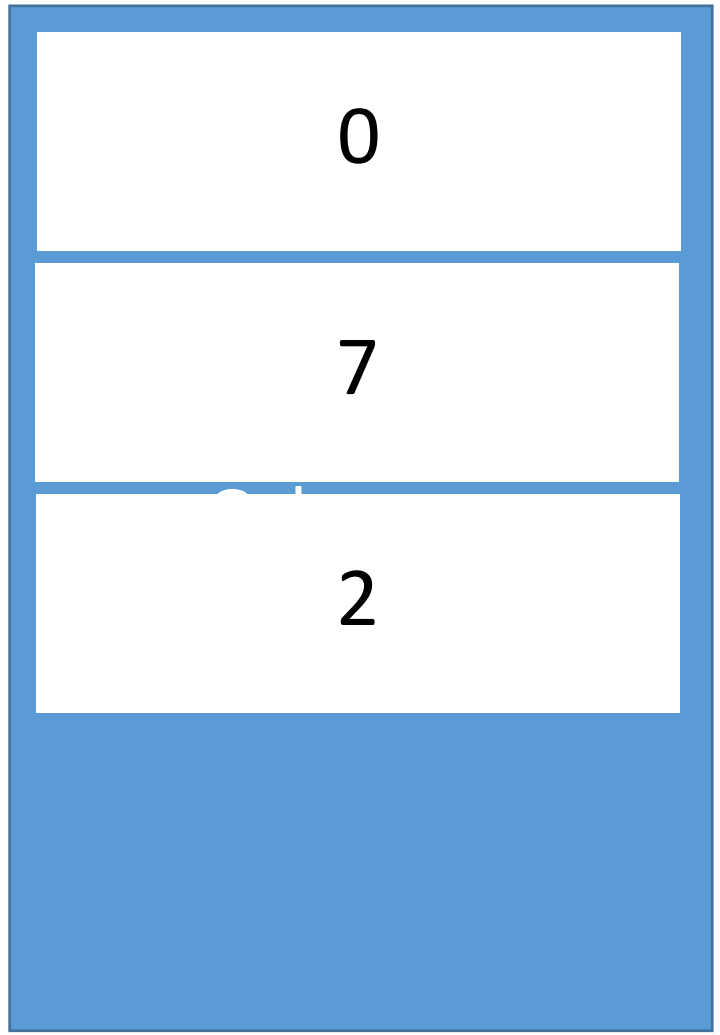
Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
int* p = &i1;
```

komt er nu geheugen bij?

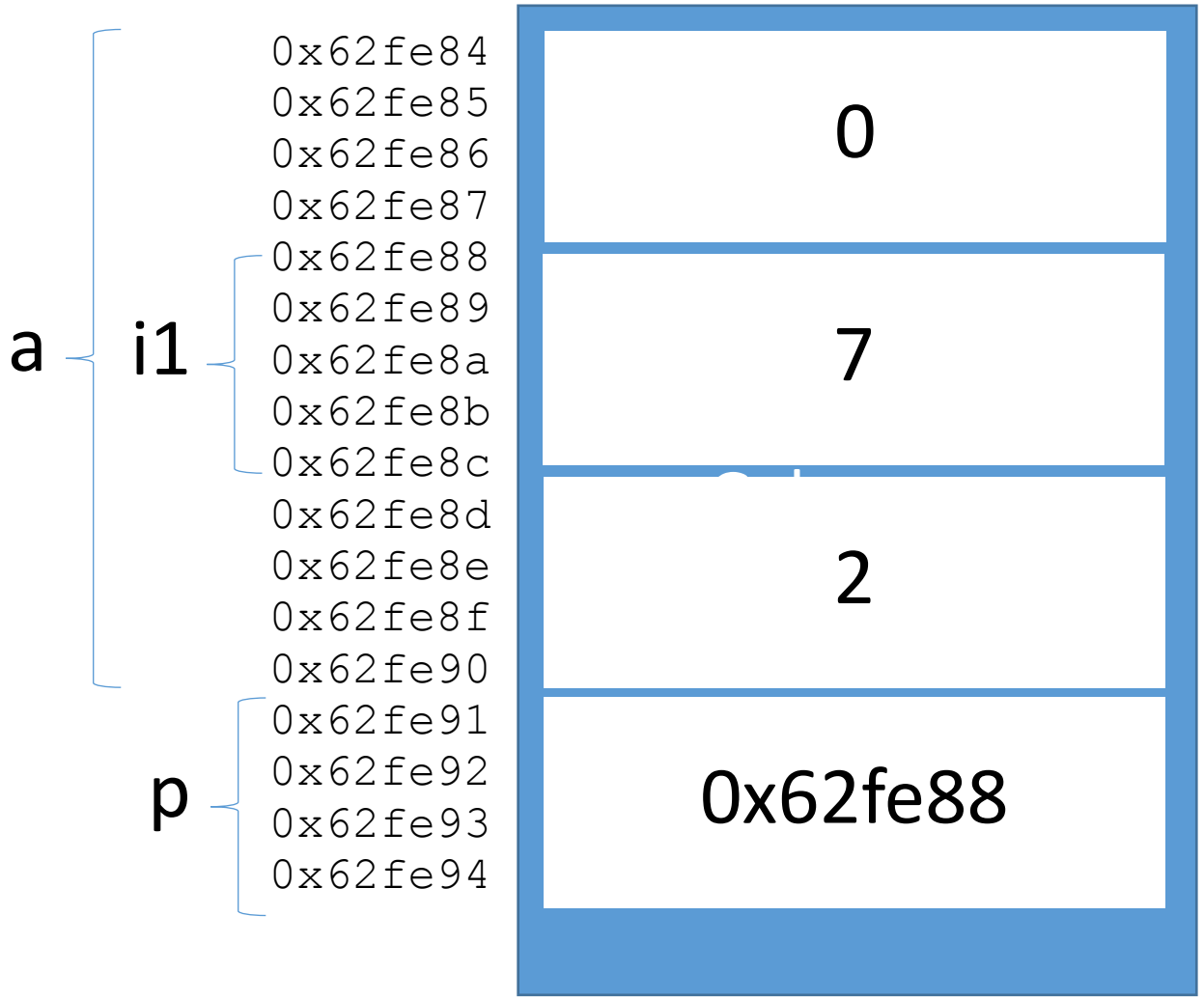
a

i1



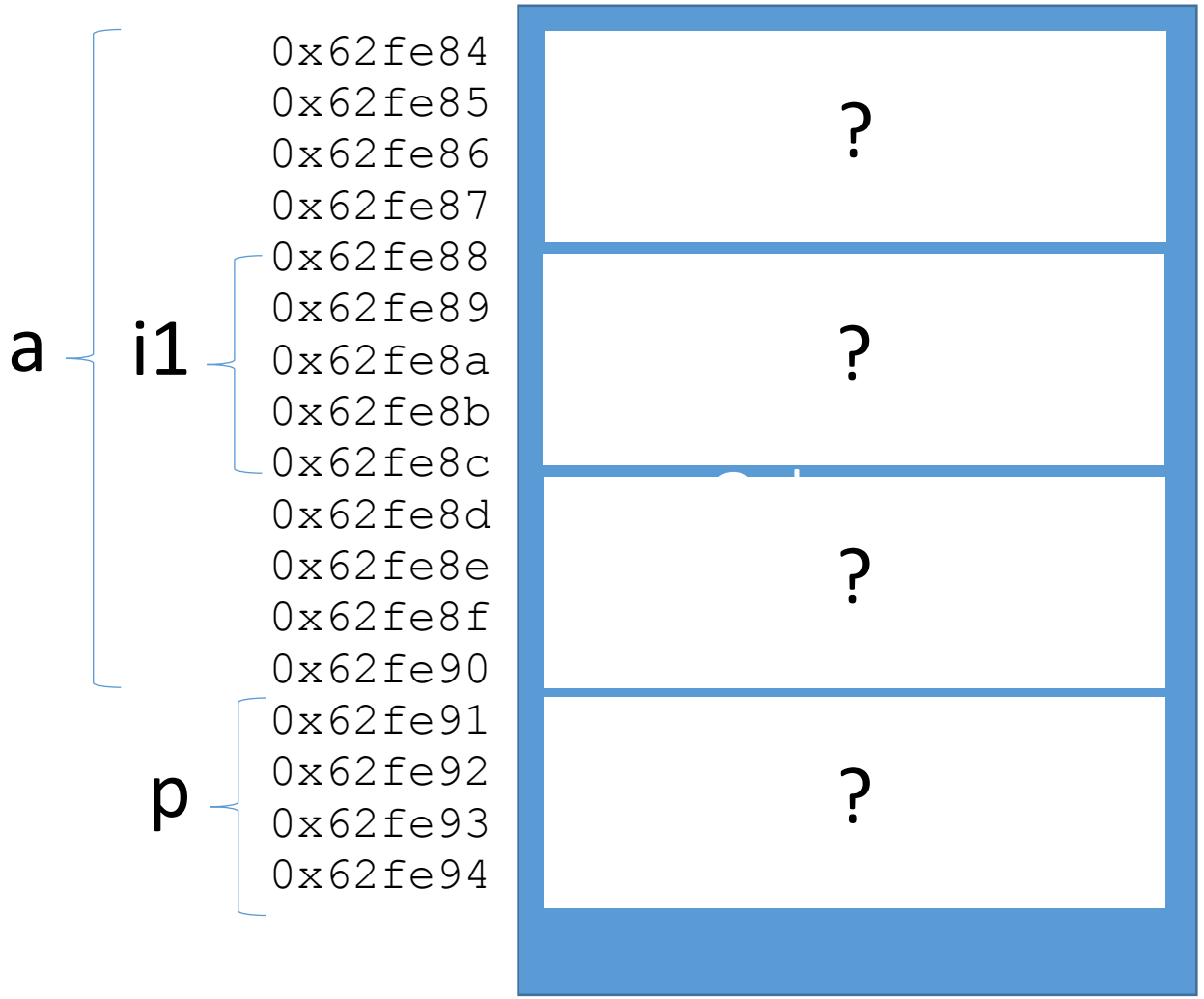
Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
int* p = &i1;
```



Referentie variabelen

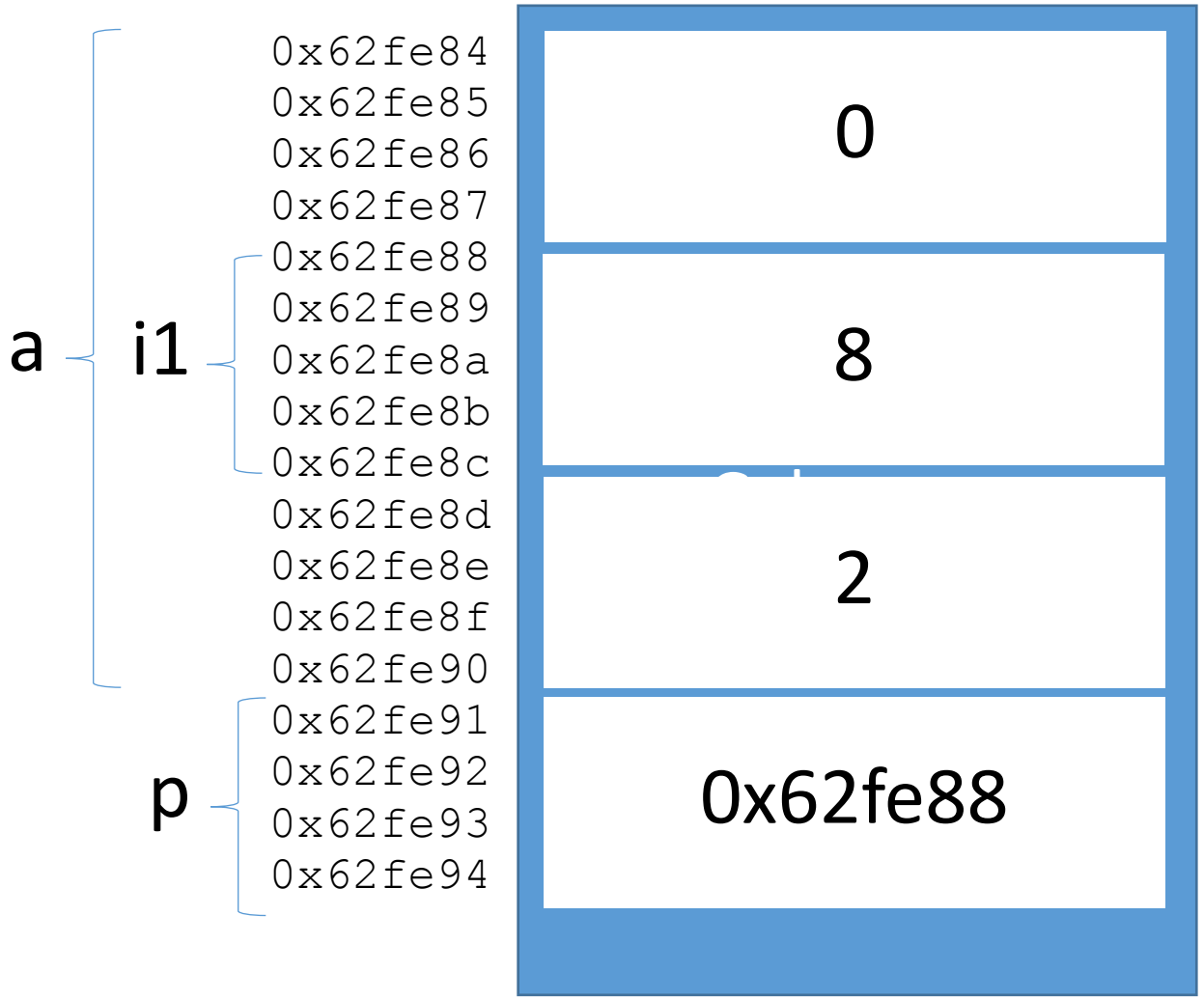
```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
int* p = &i1;
*p = 8;
```



Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
int* p = &i1;
*p = 8;
```

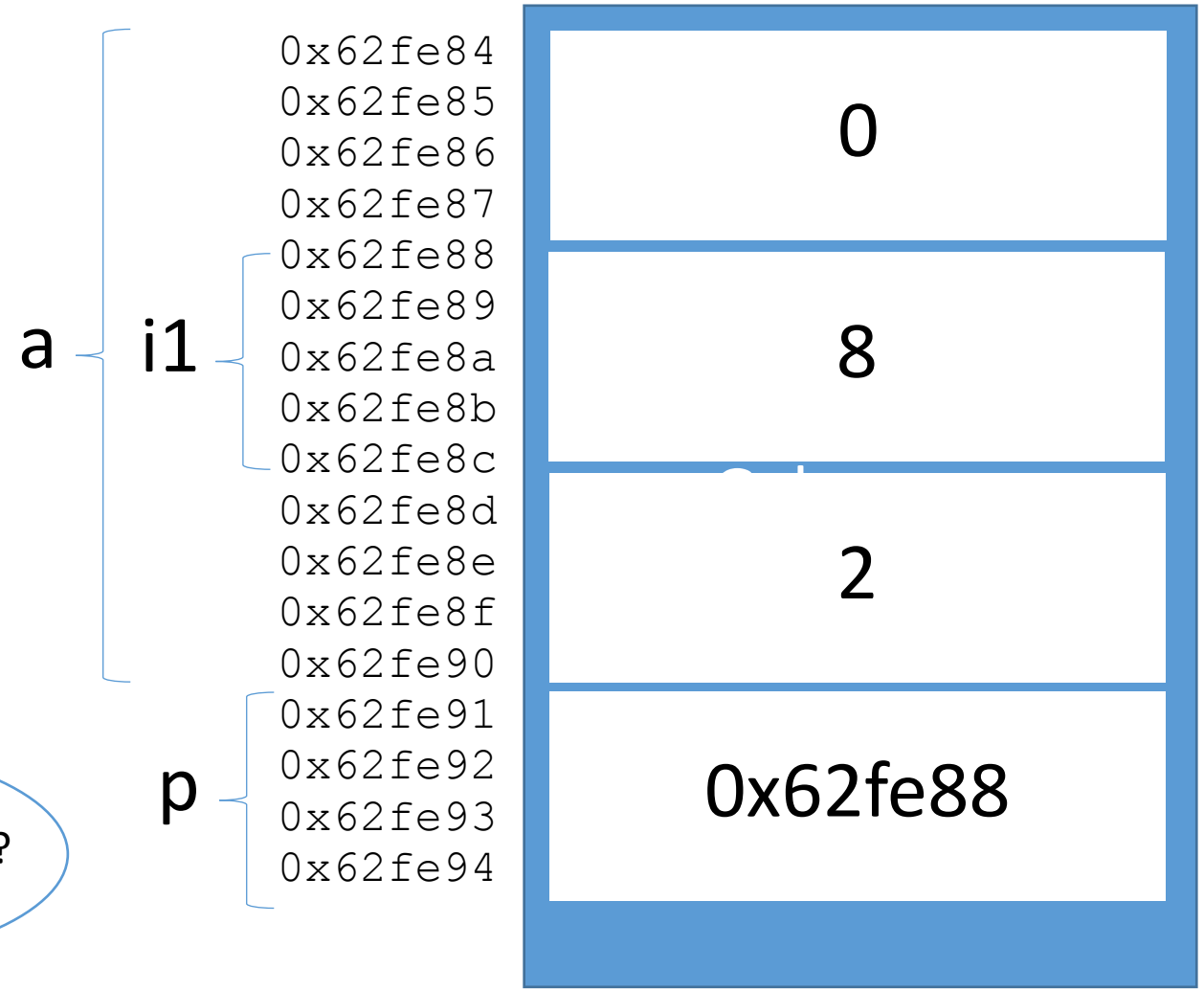
kijk na!



Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
int* p = &i1;
*p = 8;
int &i2 = *p;
```

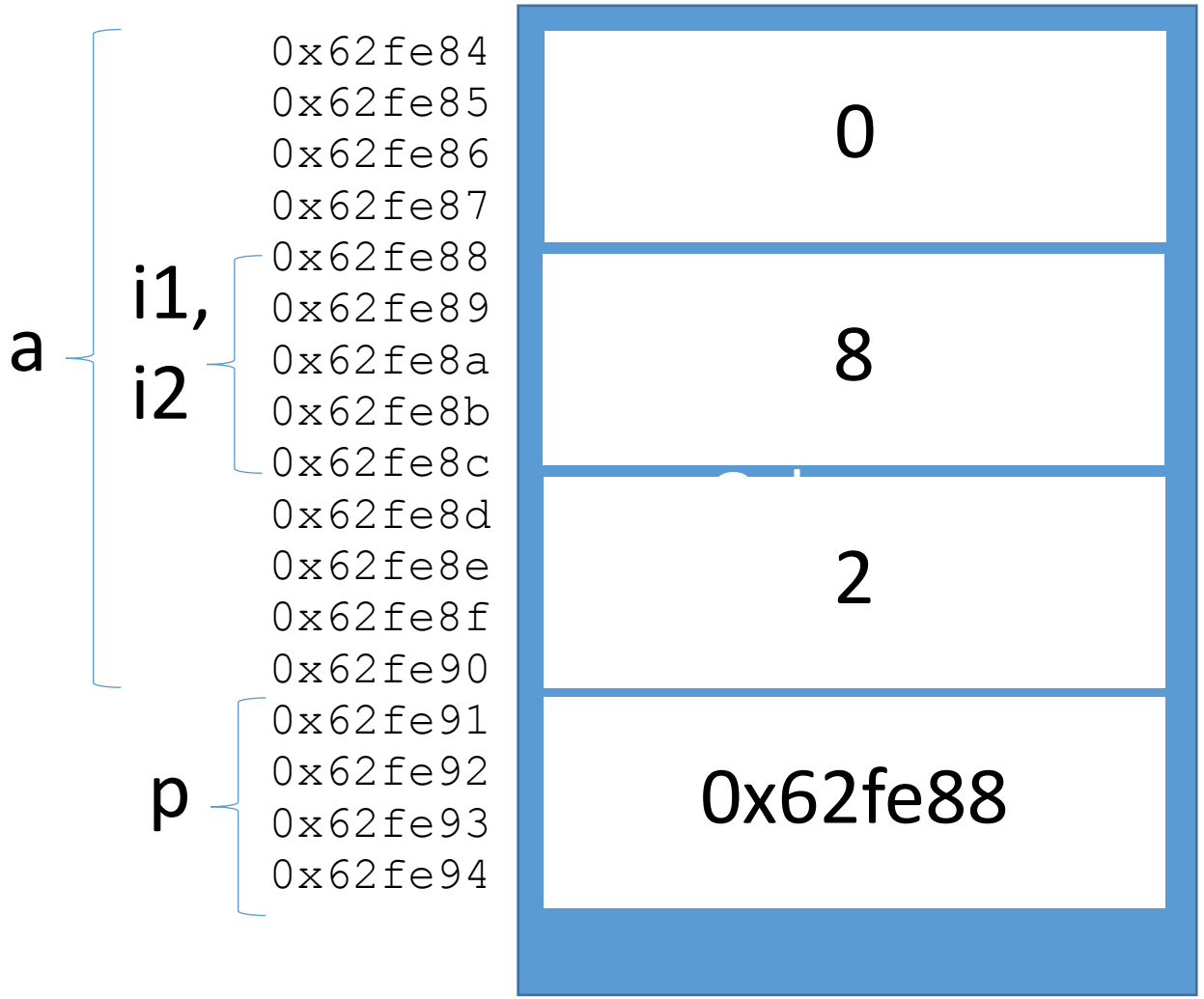
komt er nu geheugen bij?



Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
int* p = &i1;
*p = 8;
int &i2 = *p;
```

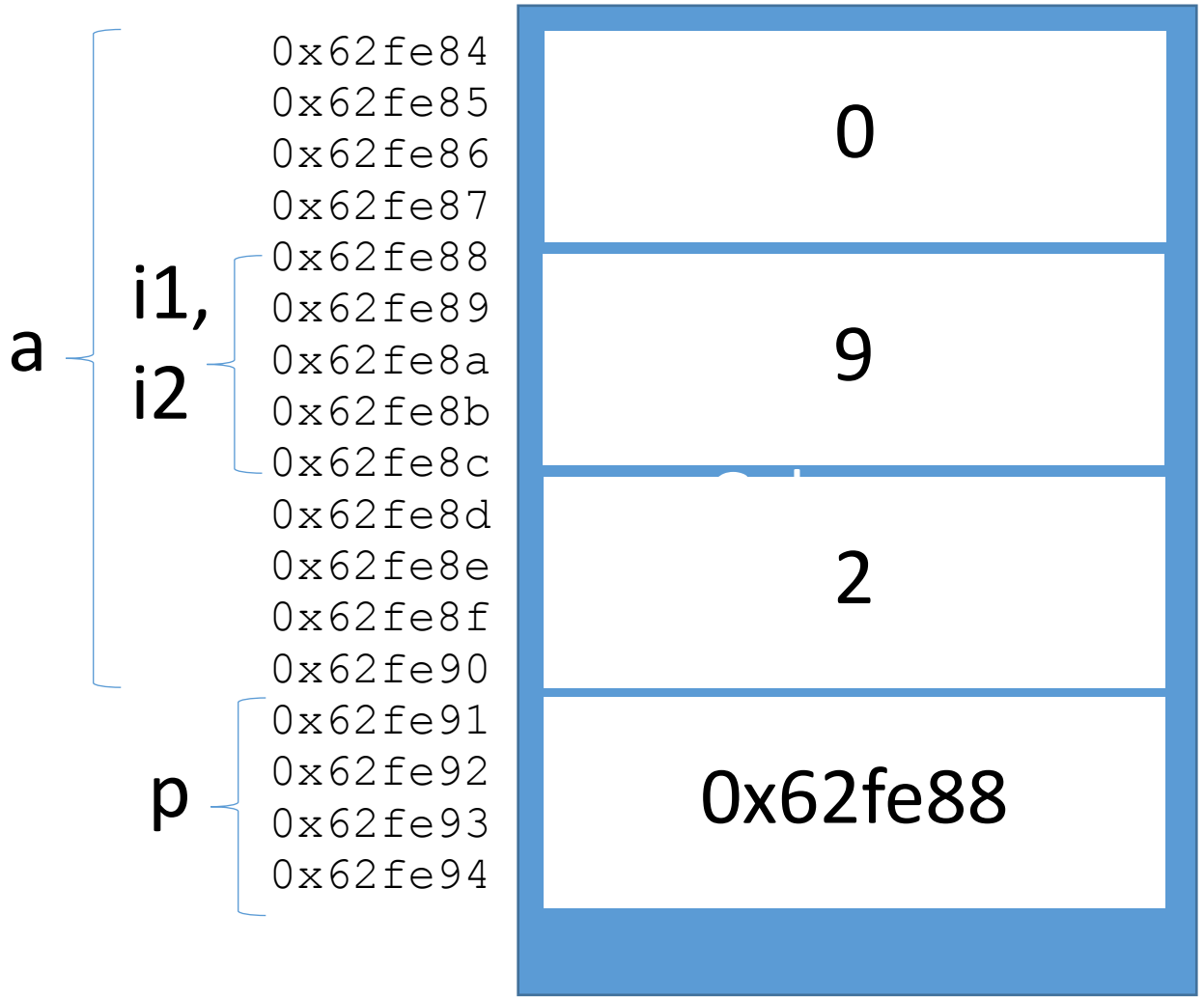
!!



Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
int* p = &i1;
*p = 8;
int &i2 = *p;
i2 = 9;
```

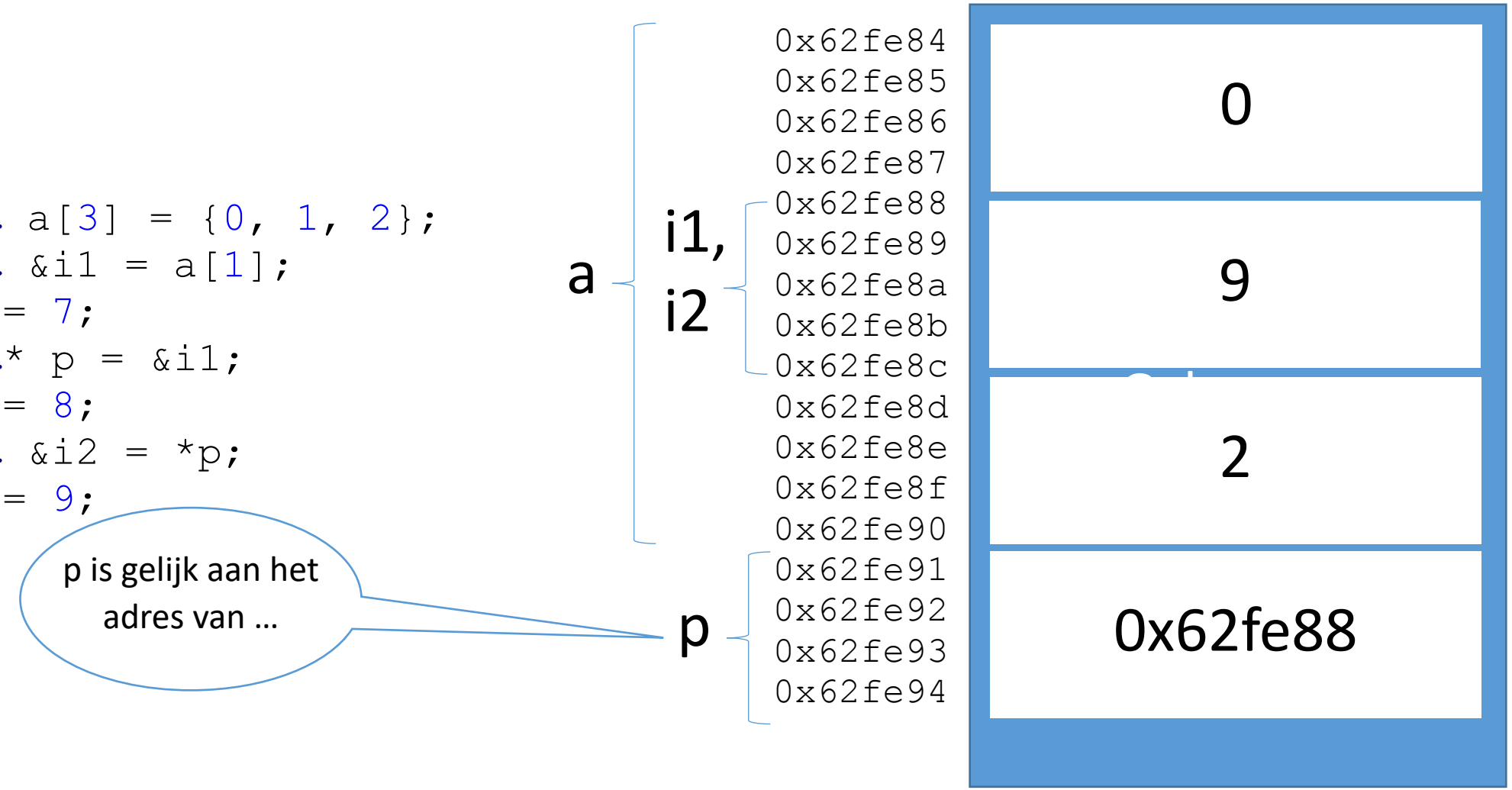
wat is er allemaal 9?



Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
int* p = &i1;
*p = 8;
int &i2 = *p;
i2 = 9;
```

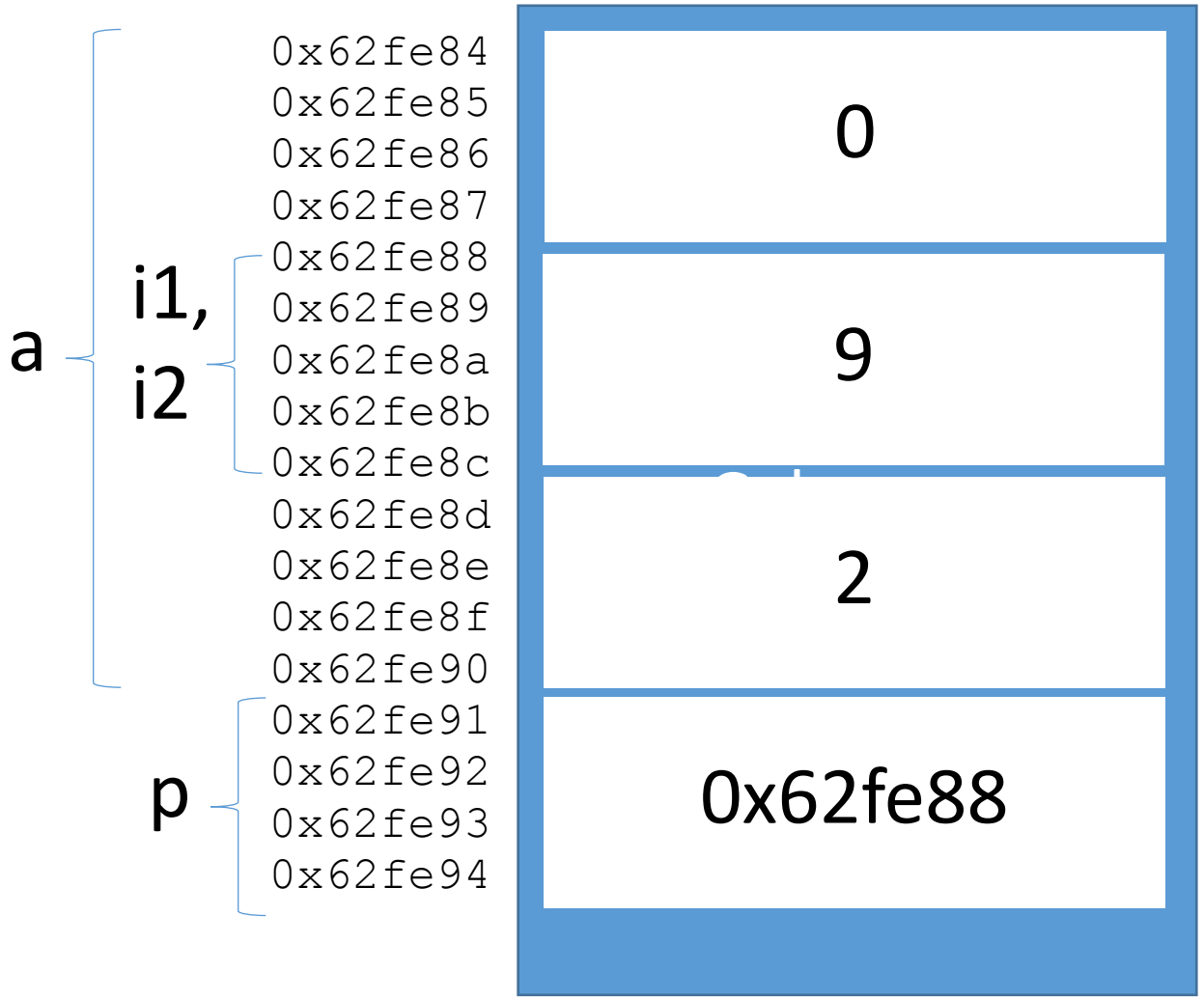
p is gelijk aan het adres van …



Referentie variabelen

```
int a[3] = {0, 1, 2};
int &i1 = a[1];
i1 = 7;
int* p = &i1;
*p = 8;
int &i2 = *p;
i2 = 9;
```

!!



a[1] == i1 == i2 == *p

p == &a[1] == &i1 == &i2

Universiteit Antwerpen

Deze les

- Pointers
  - Variabelen als alias voor geheugenlocaties
  - Pointer type
  - Referencing en dereferencing
  - Rekenen met pointers
- Referentie variabelen
- Parameters bij functieaanroepen
  - Call by value / Call by reference
- Call stack

Bekijk ook volgende uitstekende tutorial:

- http://www.cplusplus.com/doc/tutorial/pointers/

Universiteit Antwerpen

Call by Value

- Als we argumenten doorgeven bij een functie, gebeurt dat standaard *by value*; d.w.z. : er wordt een *kopie* gemaakt van de inhoud van de variabele en dat wordt aan de parameter toegekend.
  - (standaard: shallow copy tenzij anders gespecifieerd; voor de STL klassen is dat het geval)

- In de functie kunnen we de waarde van een parameter veranderen, maar dat verandert enkel de *kopie*.

Call by Value

```
void f(int a, vector<int> v, vector<vector<int>> w) {
    a=5;
    v[0]=3;
    w[0][0]=7;
    cout << "Inside f: " << a << " "
         << v[0] << " " << w[0][0] << endl;
}

int main() {
    int a=3;
    vector<int> v={1,2,3};
    vector<vector<int>> w={{2,5,6},{3,8,9}};

    f(a,v,w);
    cout << "Outside f: " << a << " "
         << v[0] << " " << w[0][0] << endl;
```

in python werd er nooit een kopie gemaakt van een lijst dus dit kan niet in python

komt er nu geheugen bij?

```
Inside f: 5 3 7
Outside f: 3 1 2
```

Call by Reference

- Soms willen we echter argumenten doorgeven via een *referentie* i.p.v. via een kopie van hun waarde
  - Als we de inhoud willen wijzigen
  - Als kopiëren erg kostelijk en onnodig is

- *By reference* betekent dat het argument wordt meegegeven aan de functie door de parameter als *alias* te gebruiken voor het argument

Call by Reference

```
void f_by_ref(int &a, vector<int> &v, vector<vector<int>> &w) {
    a=5;
    v[0]=3;
    w[0][0]=7;
    cout << "Inside f_by_ref: " << a << " "
         << v[0] << " " << w[0][0] << endl;
}

int main() {
    int a=3;
    vector<int> v={1,2,3};
    vector<vector<int>> w={{2,5,6},{3,8,9}};

    f_by_ref(a,v,w);
    cout << "Outside f_by_ref: " << a << " "
         << v[0] << " " << w[0][0] << endl;
```

beide verwijzen naar DEZELFDE geheugenplaats

komt er nu geheugen bij?

```
Inside f_by_ref: 5 3 7
Outside f_by_ref: 5 3 7
```

Call by Reference

```
void f_by_ref(int &a, vector<int> &v, vector<vector<int>> &w) {
     a=5;
     v[0]=3;
     w[0][0]=7;
     cout << "Inside f_by_ref: " << a << " "
          << v[0] << " " << w[0][0] << endl;
}

int main() {
     int a=3;
     vector<int> v={1,2,3};
     vector<vector<int>> w={{2,5,6},{3,8,9}};

     f_by_ref(a,v,w);
     cout << "Outside f_by_ref: " << a << " "
          << v[0] << " " << w[0][0] << endl;
```

je kan dus enkel lvalues doorgeven!
f_by_ref(3,…) werkt dus niet
aangezien de parameter a
geen nieuwe geheugenplaats heeft

Simulatie call-by-reference met pointers

```
void swap(int&x, int&y) {
    int z=x;
    x=y;
    y=z;
}
```

wat doet deze functie?

Call: **swap(3,4)**

Simulatie call-by-reference met pointers

```
void swap(int&x, int&y) {
    int z=x;
    x=y;
    y=z;
}
```

Call: **swap(3,4)**

Simulatie call-by-reference met pointers

```
void swap(int&x, int&y) {
    int z=x;
    x=y;
    y=z;
}
```

Call:

int a = 3;

int b = 4;

swap(a,b);

Simulatie call-by-reference met pointers

```
void swap(int&x, int&y) {
    int z=x;
    x=y;
    y=z;
}
```

call by value of call by ref?

Call: **swap(a,b)**

call by ref

==> ampersands

```
void swap_p(int*x, int*y) {
    int z=*x;
    *x=*y;
    *y=z;
}
```

call by value of call by ref?

call by value

Call: **swap_p(&a,&b)**

Return “by reference”

- We kunnen variabelen niet enkel by reference doorgeven, maar ook by reference *teruggeven*

```
int& max(vector<int>& v) {
    int max=v[0];
    int maxpos=0;
    for (int j=1;j<v.size();j++) {
        if (v[j]>max) {
            max=v[j];
            maxpos=j;
        }
    }
    return v[maxpos];
}
```

Referentie variabelen

- Wanneer is dit handig?
  - Als we een complexe lvalue hebben
  - Als return waarde van een functie die we direct willen manipuleren

```
int& max(vector<int>& v) {
    int max=v[0];
    int maxpos=0;
    for (int j=1;j<v.size();j++) {
        if (v[j]>max) {
            max=v[j];
            maxpos=j;
        }
    }
    return v[maxpos];
}
```

```
int main() {
    vector<int> v={0,1,4,7,2,3,6};
    max(v)=-1;
    for(int i=0;i<v.size();i++)
        cout << v[i] << " ";
    cout << endl;
}
```

0 1 4 -1 2 3 6

Referentie variabelen

- Zijn in principe equivalent:

```
int& max(vector<int>& v) {
    int max=v[0];
    int maxpos=0;
    for (int j=1;j<v.size();j++) {
        if (v[j]>max) {
            max=v[j];
            maxpos=j;
        }
    }
    return v[maxpos];
}

max(v)=-1;
```

```
int* max2(vector<int>& v) {
    int max=v[0];
    int maxpos=0;
    for (int j=1;j<v.size();j++) {
        if (v[j]>max) {
            max=v[j];
            maxpos=j;
        }
    }
    return &v[maxpos];
}

*max(v)=-1;
```

Universiteit Antwerpen

Wat móet ik onthouden?

- Variabele = een alias voor een geheugenlocatie.
  - Jij schrijft: “int a=5;”,
    De compiler leest “Reserveer een plaats in het geheugen groot genoeg om een int te bevatten en sla in deze locatie 5 op”
  - Jij schrijft: “c=2*a;”
    De compiler leest “Lees geheugenplaats a uit, doe maal twee en sla het resultaat op in geheugenplaats c.”
- Lokale variabelen staan op de stack en verdwijnen zodra de functie afgelopen is.
- Parameter *by reference*: parameter is een alias voor de geheugenlocatie van het argument; blijft dus wél bestaan nadat de functie afgelopen is.
  - Geïmplementeerd met behulp van pointer

Universiteit Antwerpen

Deze les

- Pointers
  - Variabelen als alias voor geheugenlocaties
  - Pointer type
  - Referencing en dereferencing
  - Rekenen met pointers
- Referentie variabelen
- Parameters bij functieaanroepen
  - Call by value / Call by reference
- Call stack

Bekijk ook volgende uitstekende tutorial:

- http://www.cplusplus.com/doc/tutorial/pointers/

Universiteit Antwerpen

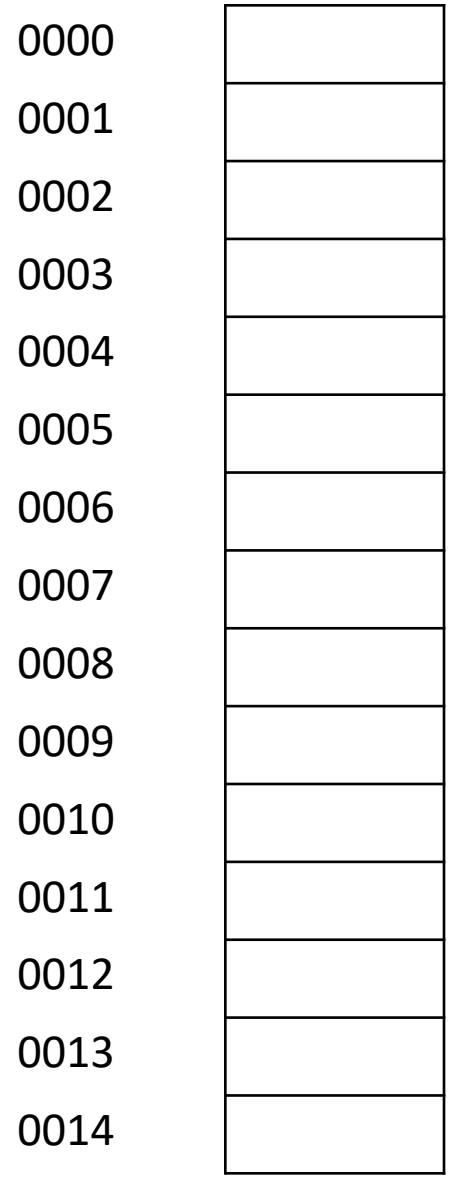
Geheugenbeheer in C++

- Geheugen is onderverdeeld in verschillende segmenten; de belangrijkste:
  - Code segment : read-only; bevat de programma-code
  - Free storage (heap): voor dynamisch toegekend geheugen
  - **Stack: lokale variabelen van alle actieve functies**

- We beschouwen in wat volgt enkel de stack

Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```

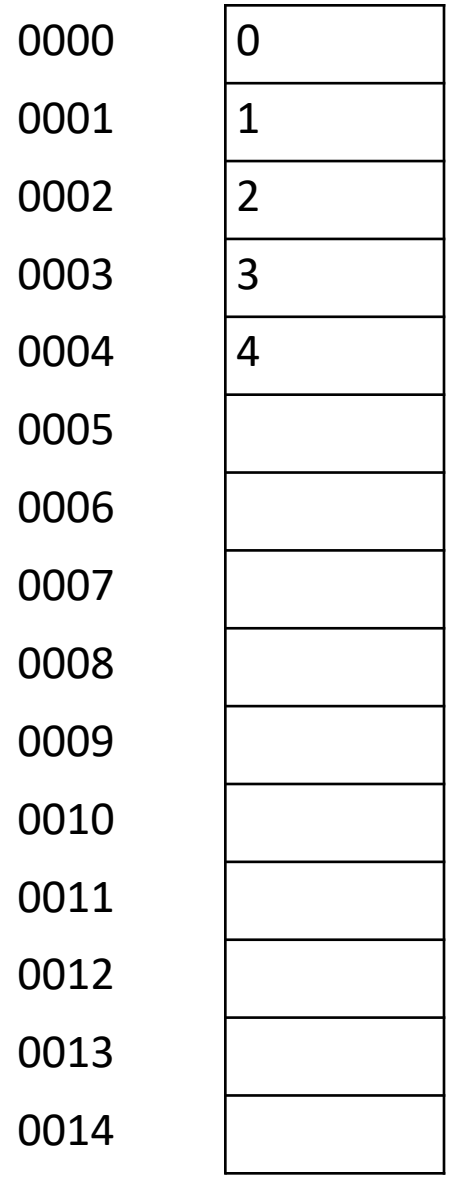


Geheugenbeheer

```
void swap(int&x, int& y) {
     int z=x;
     x=y; y=z;
}

void reverse(in
     for (int i=
          swap(a[
}

int main() {
     int v[5]={0,1,2,3,4};
     reverse(v,5);
     return 0;
}
```

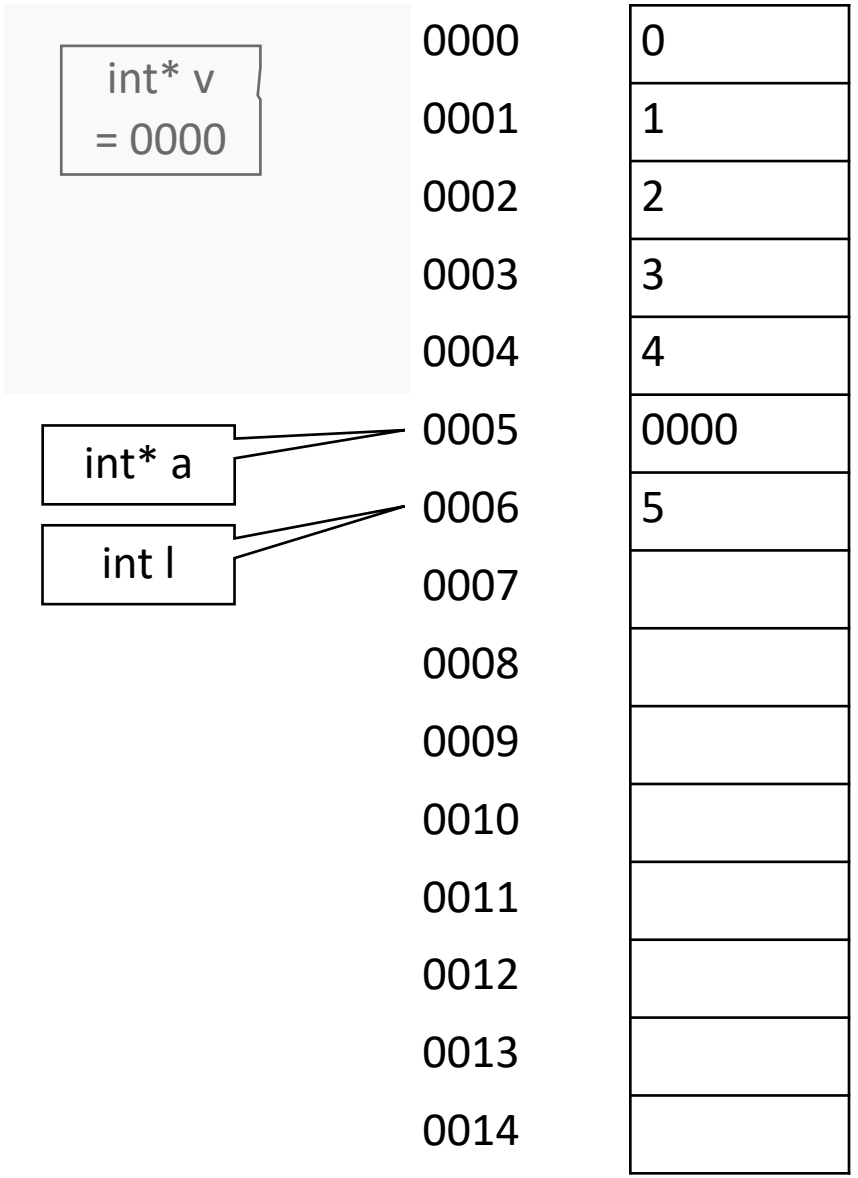
int* v
= 0000

Reminder:
statische array in C++
=
rvalue pointer naar eerste element



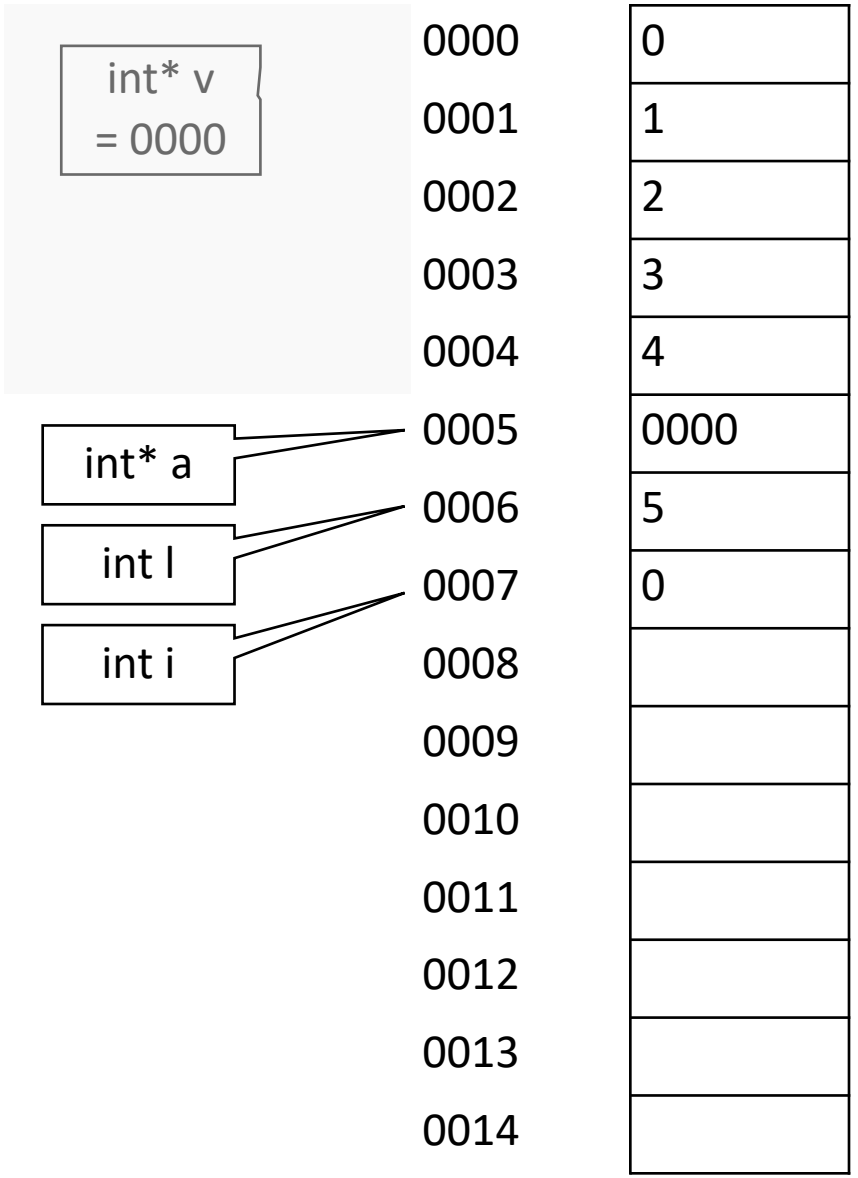
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



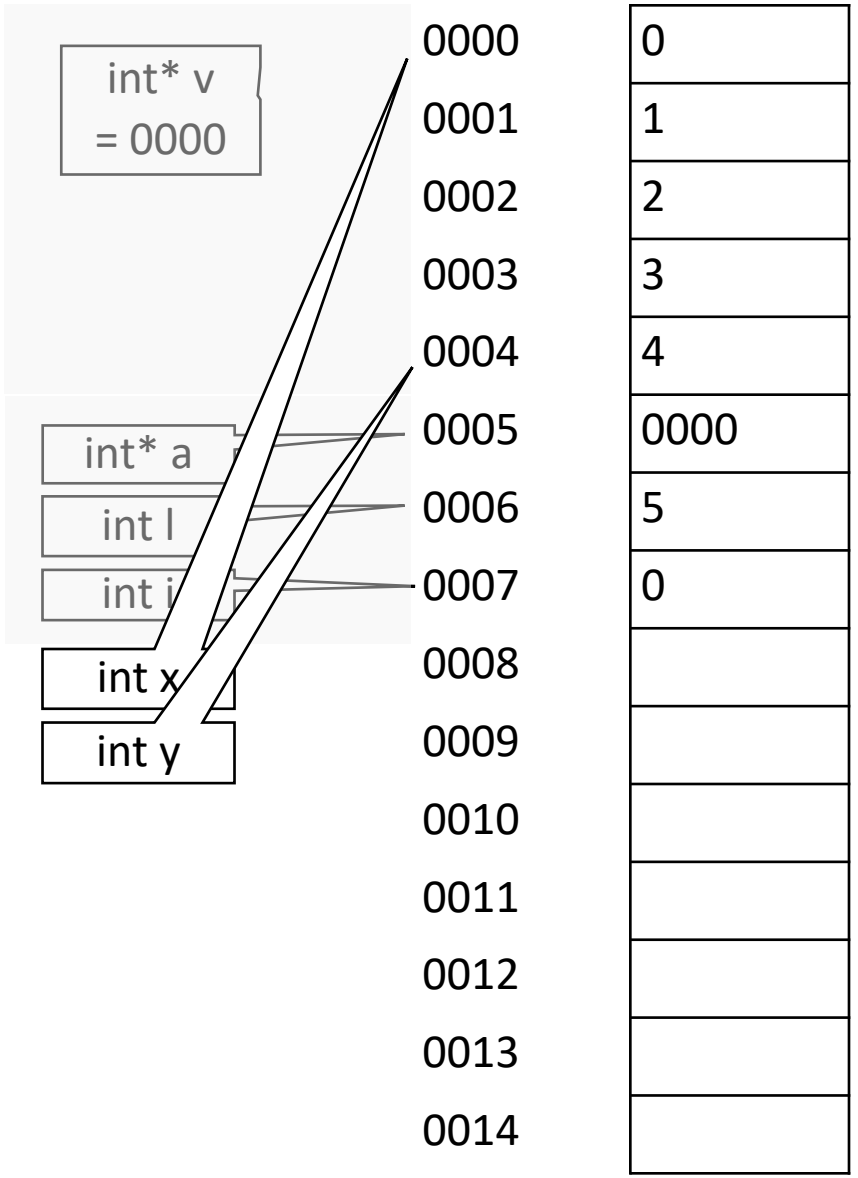
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



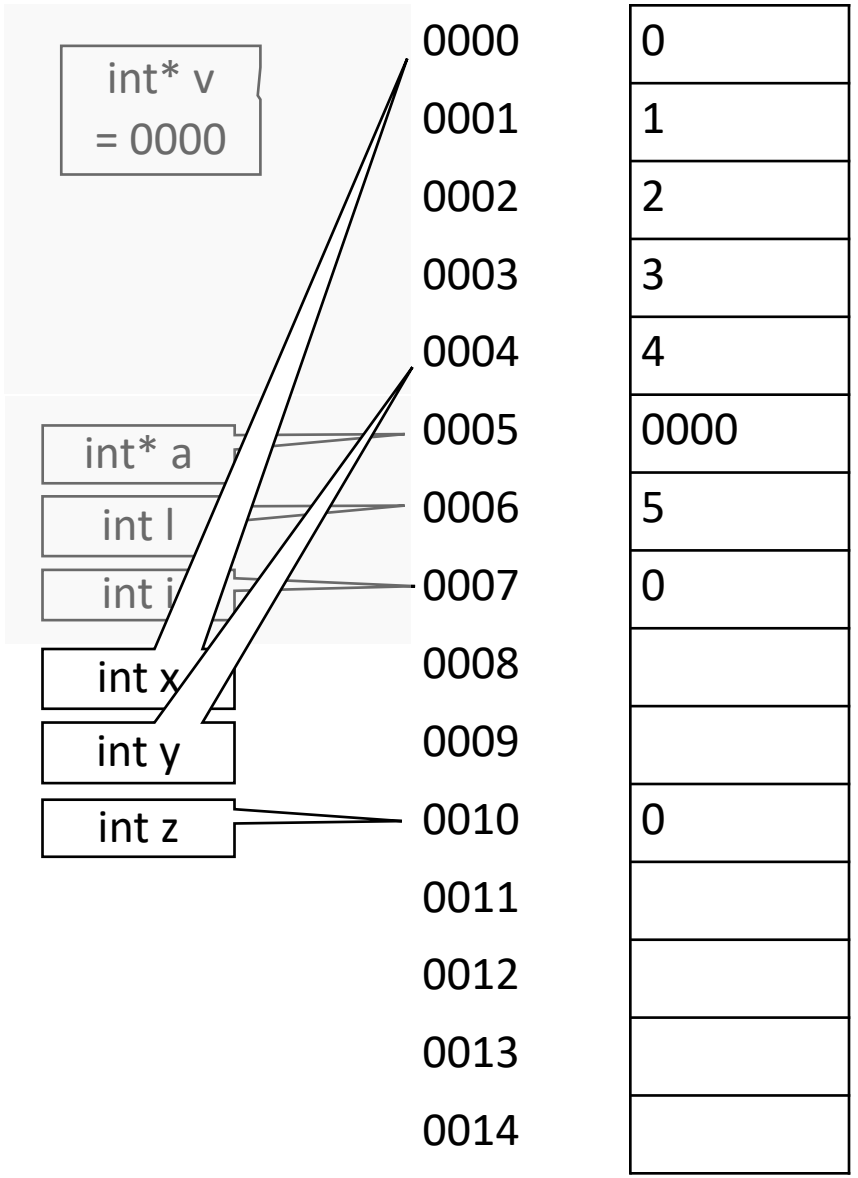
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



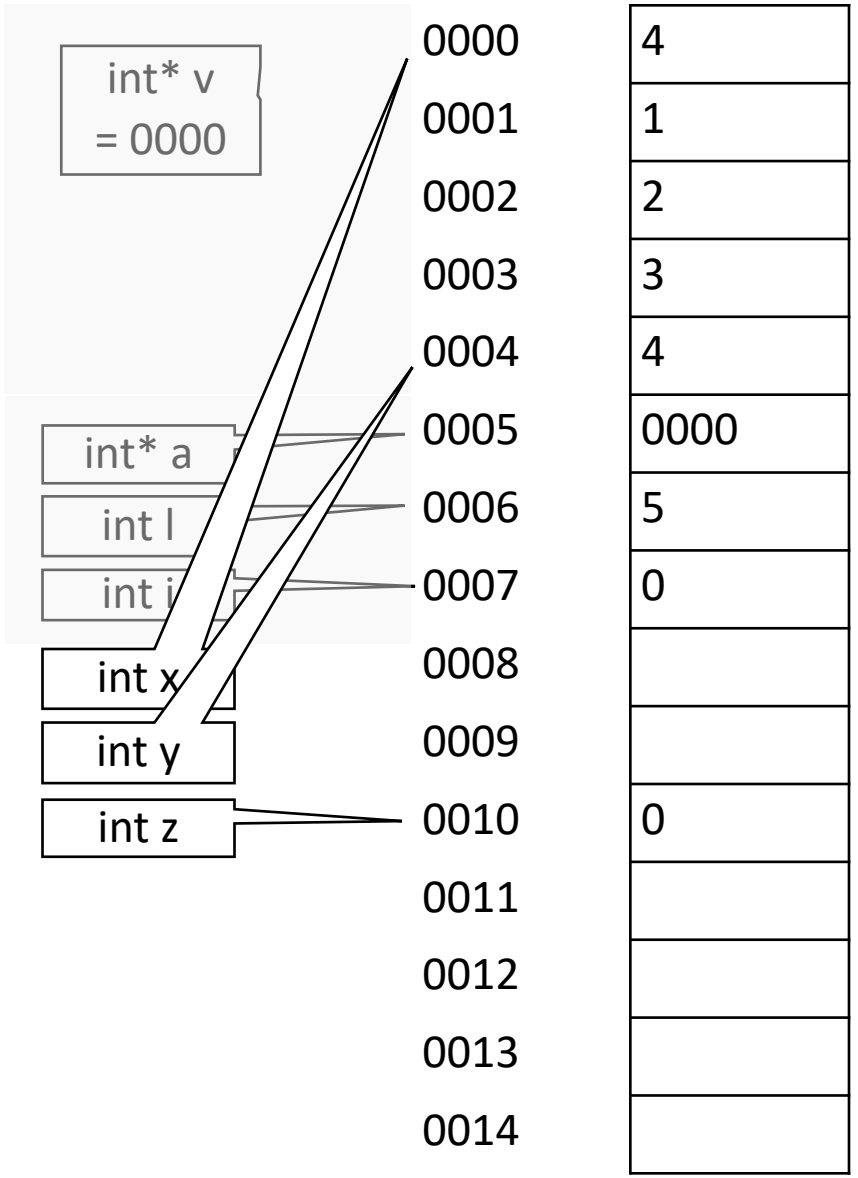
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



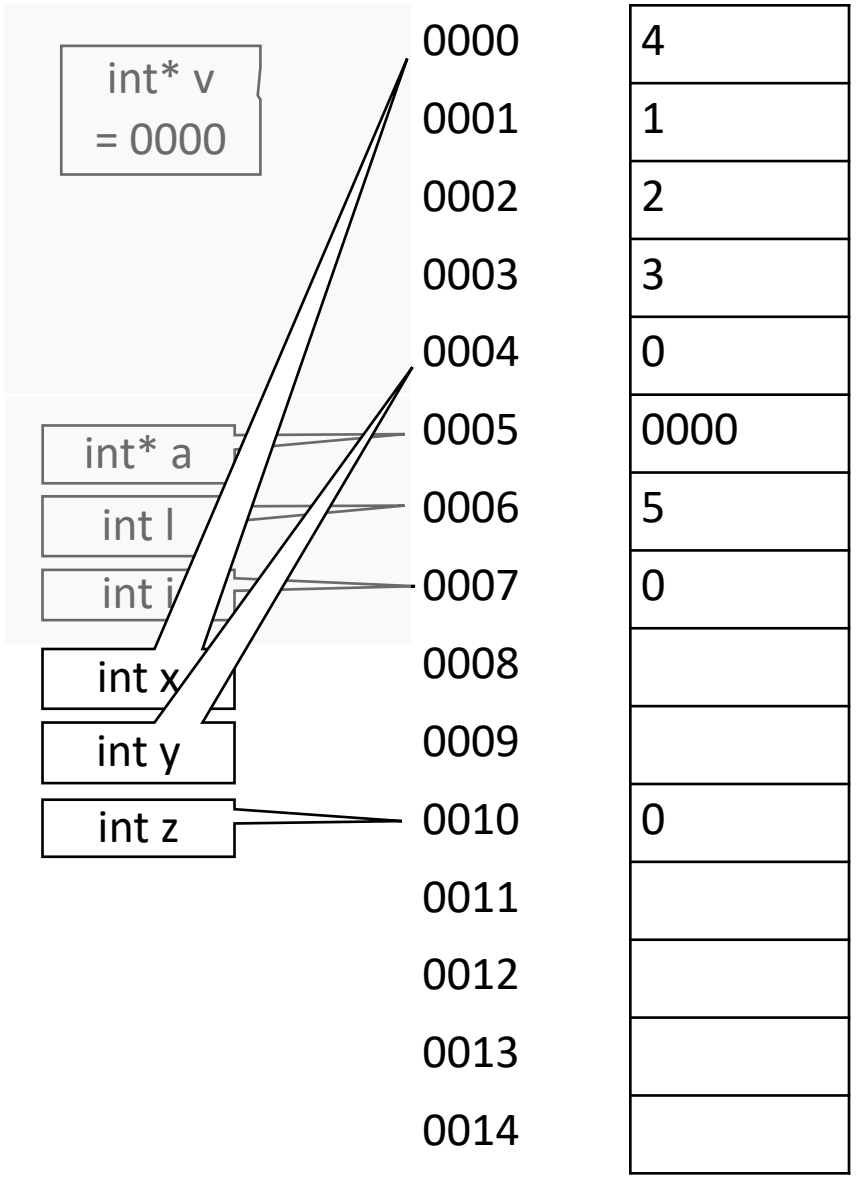
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



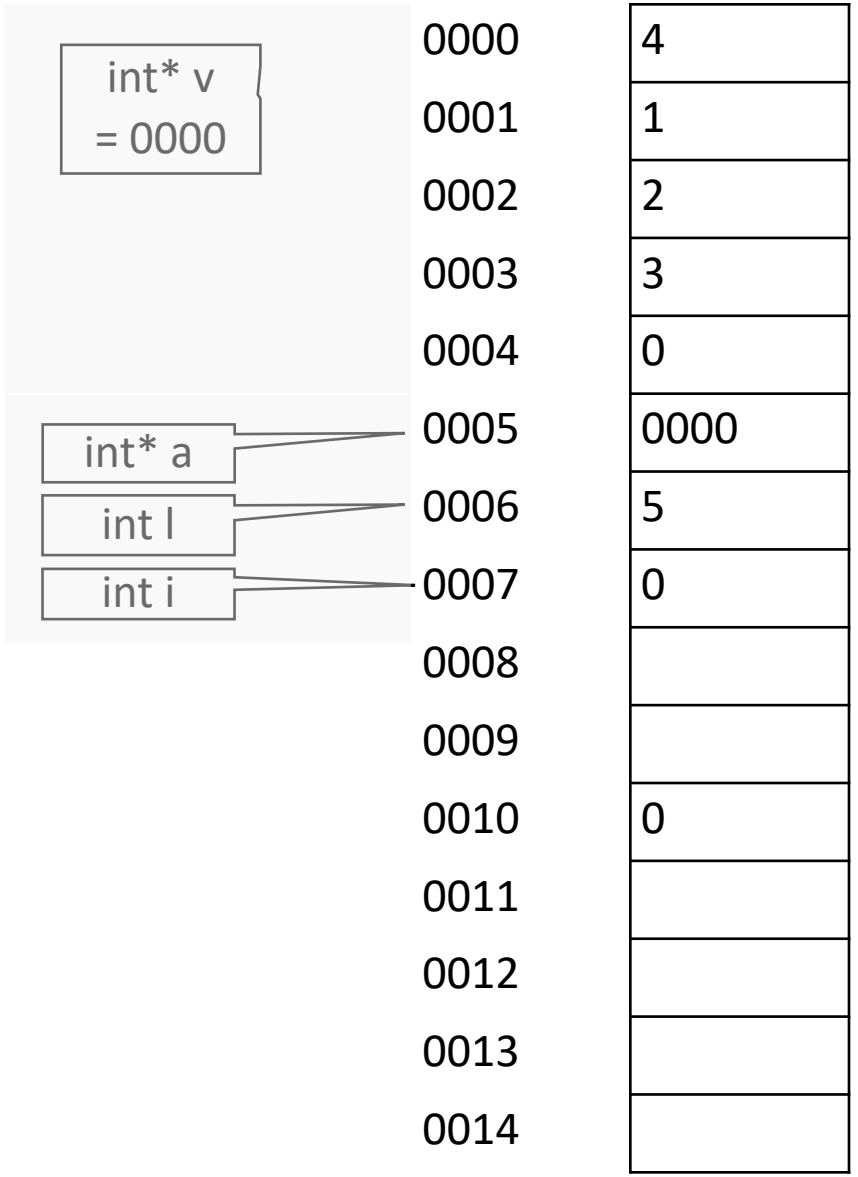
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



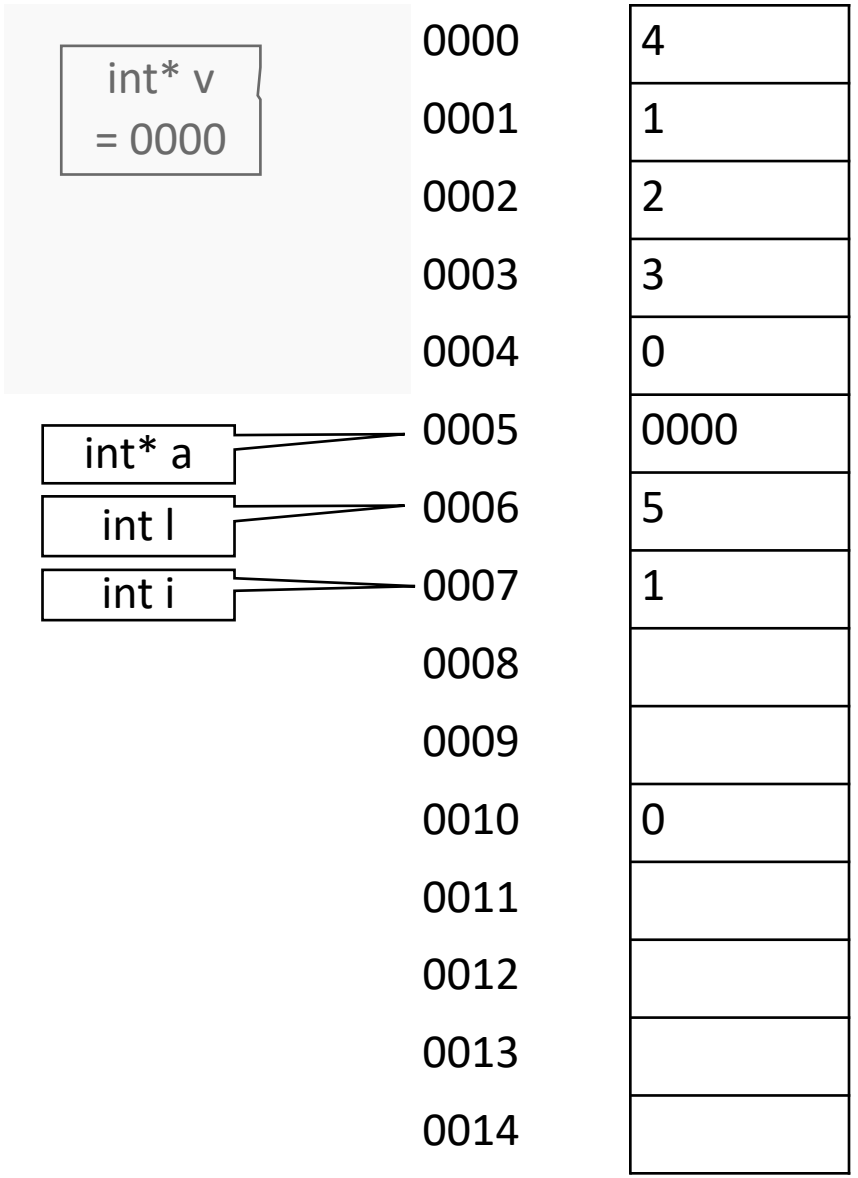
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



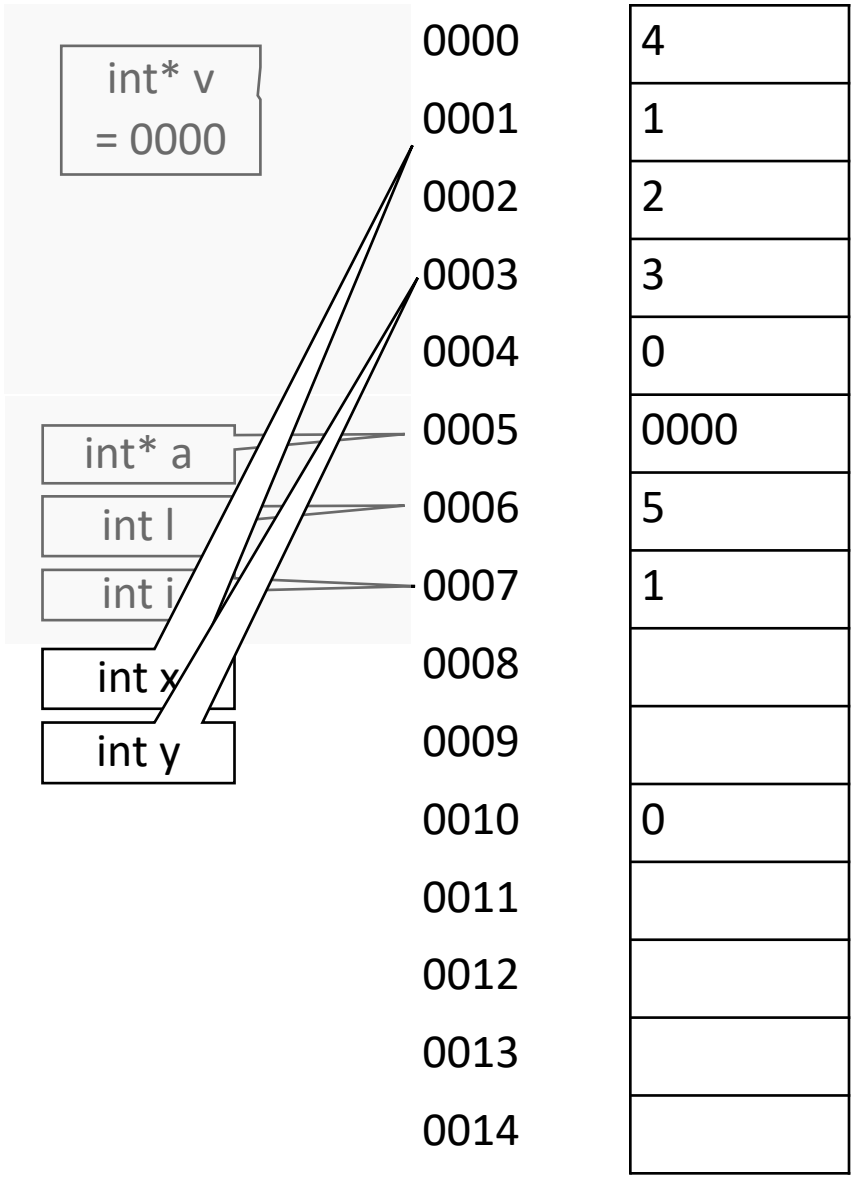
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



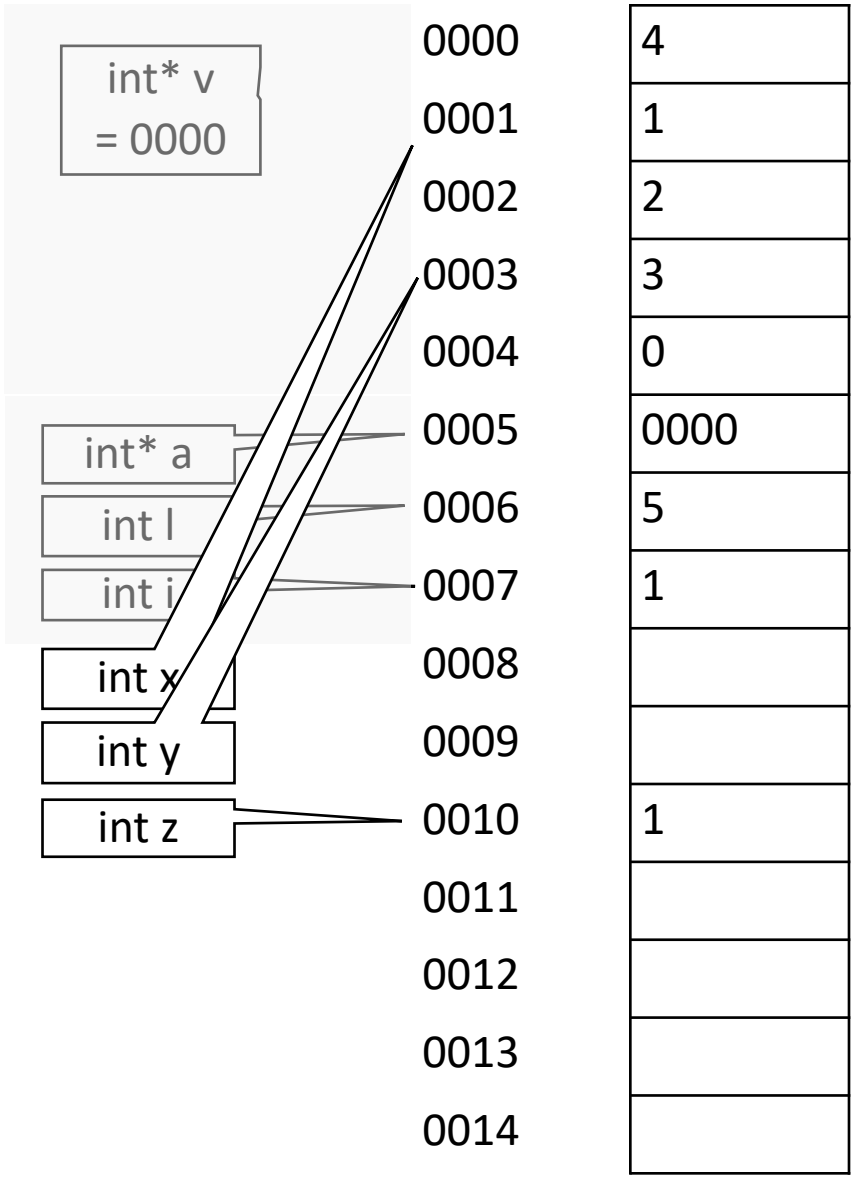
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



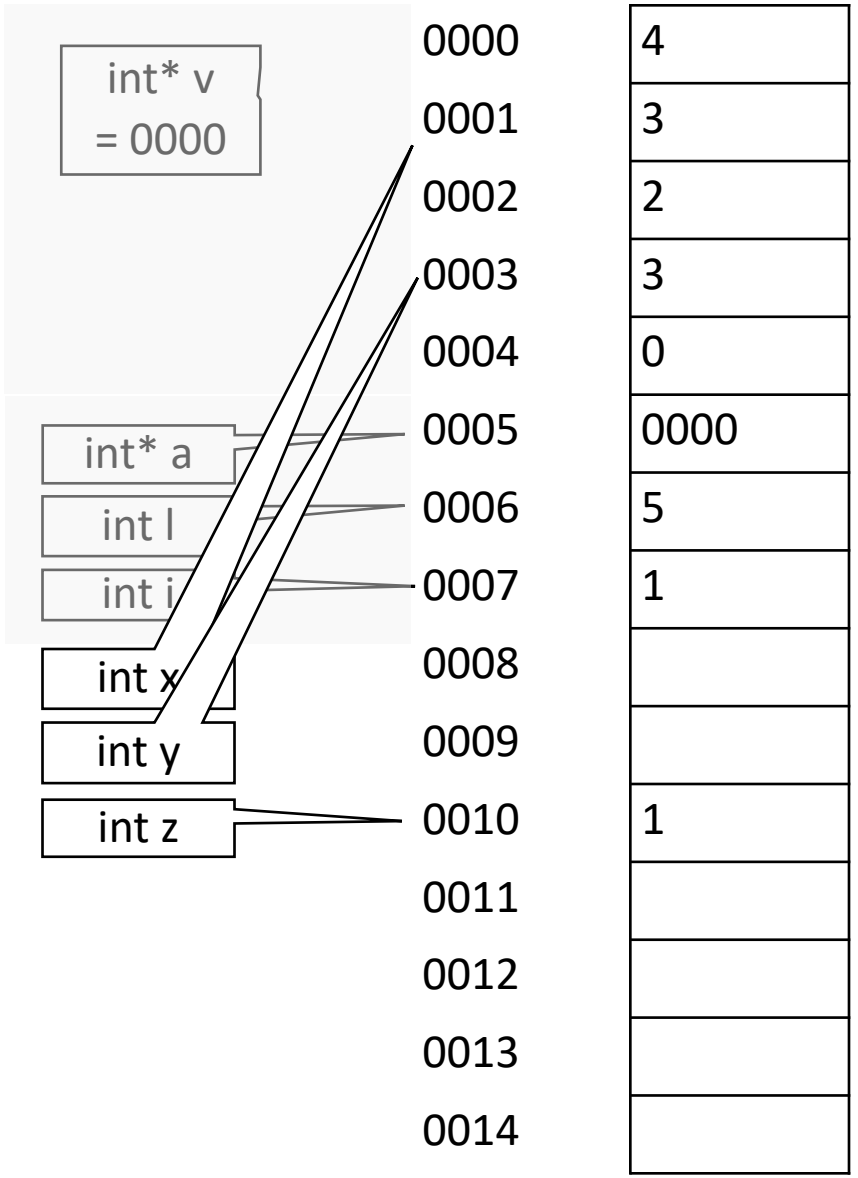
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



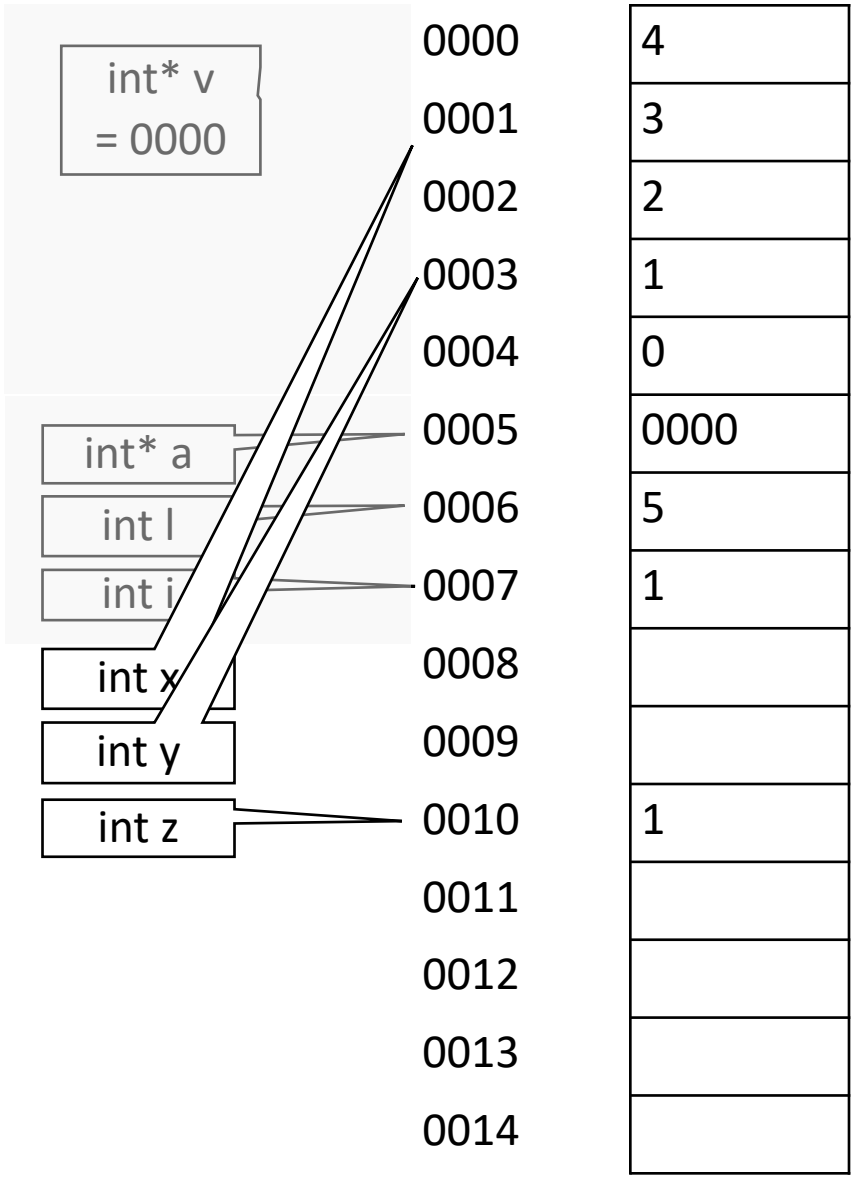
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



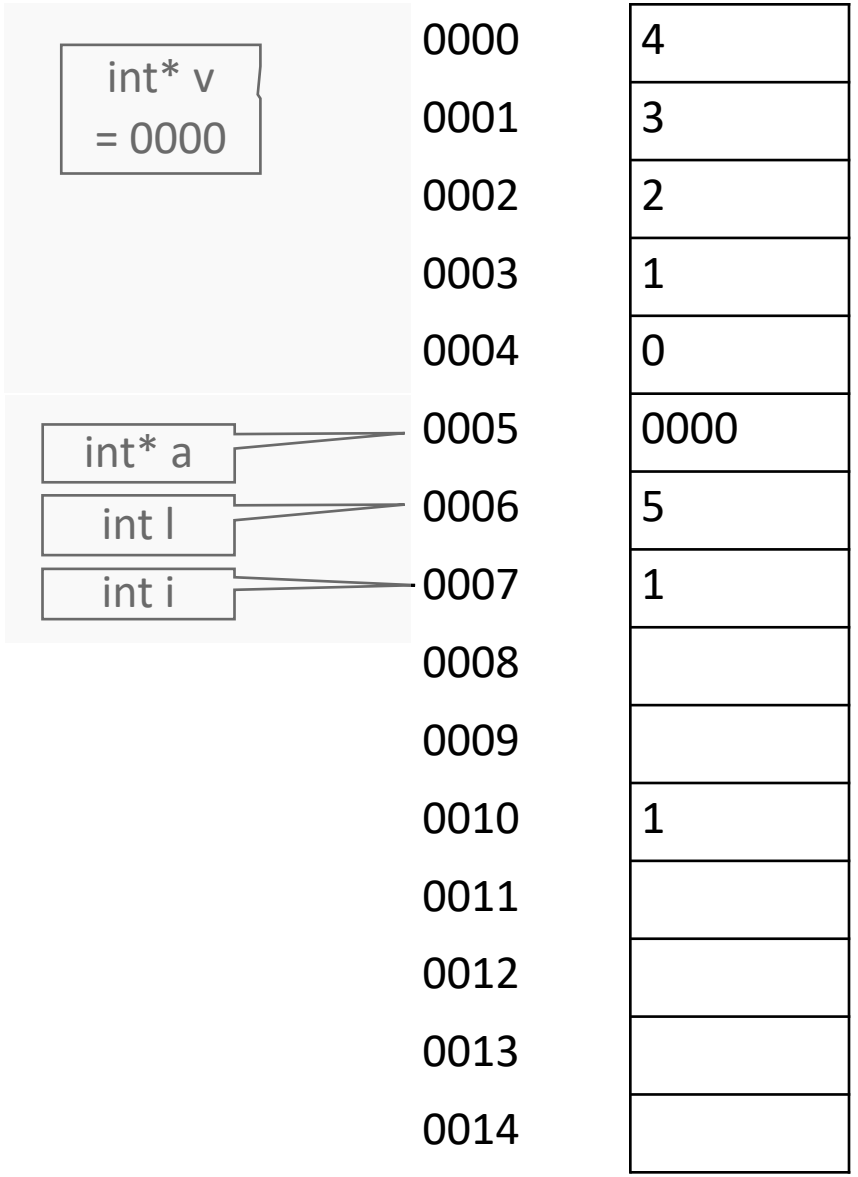
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



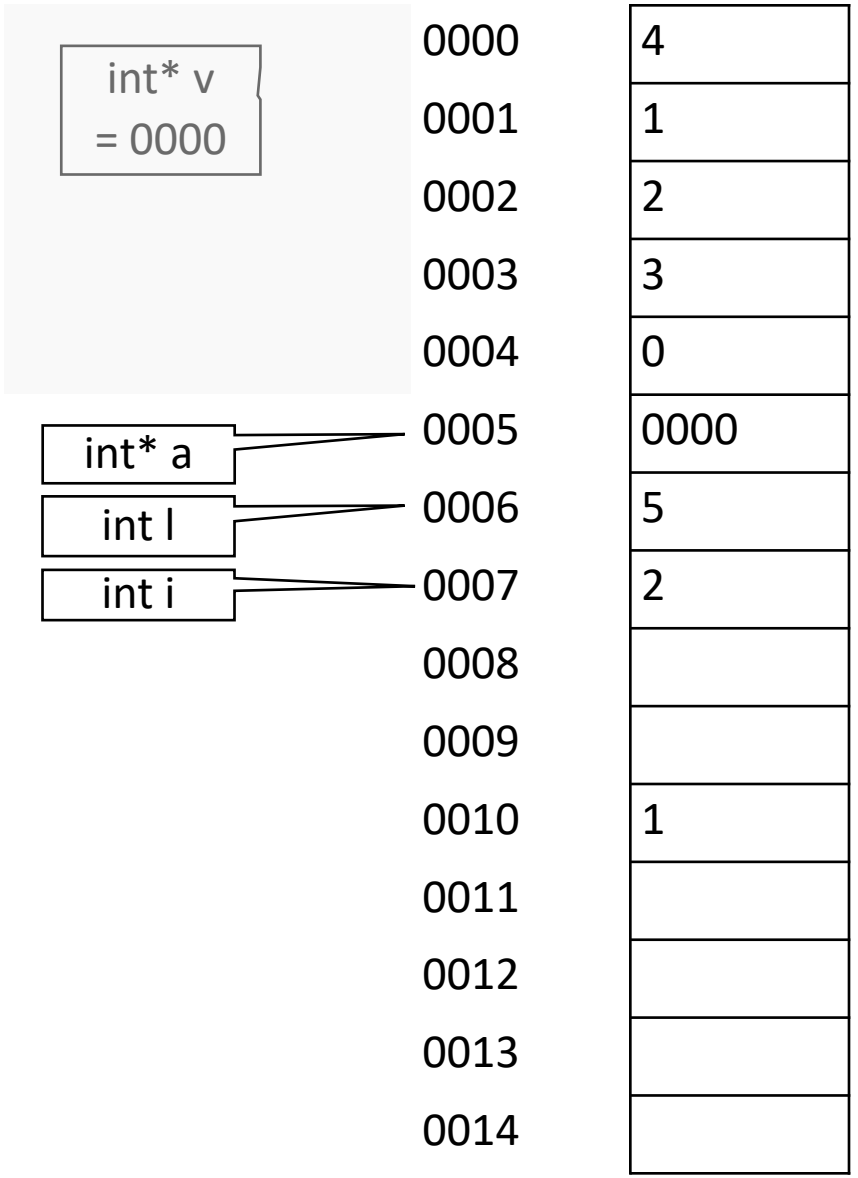
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



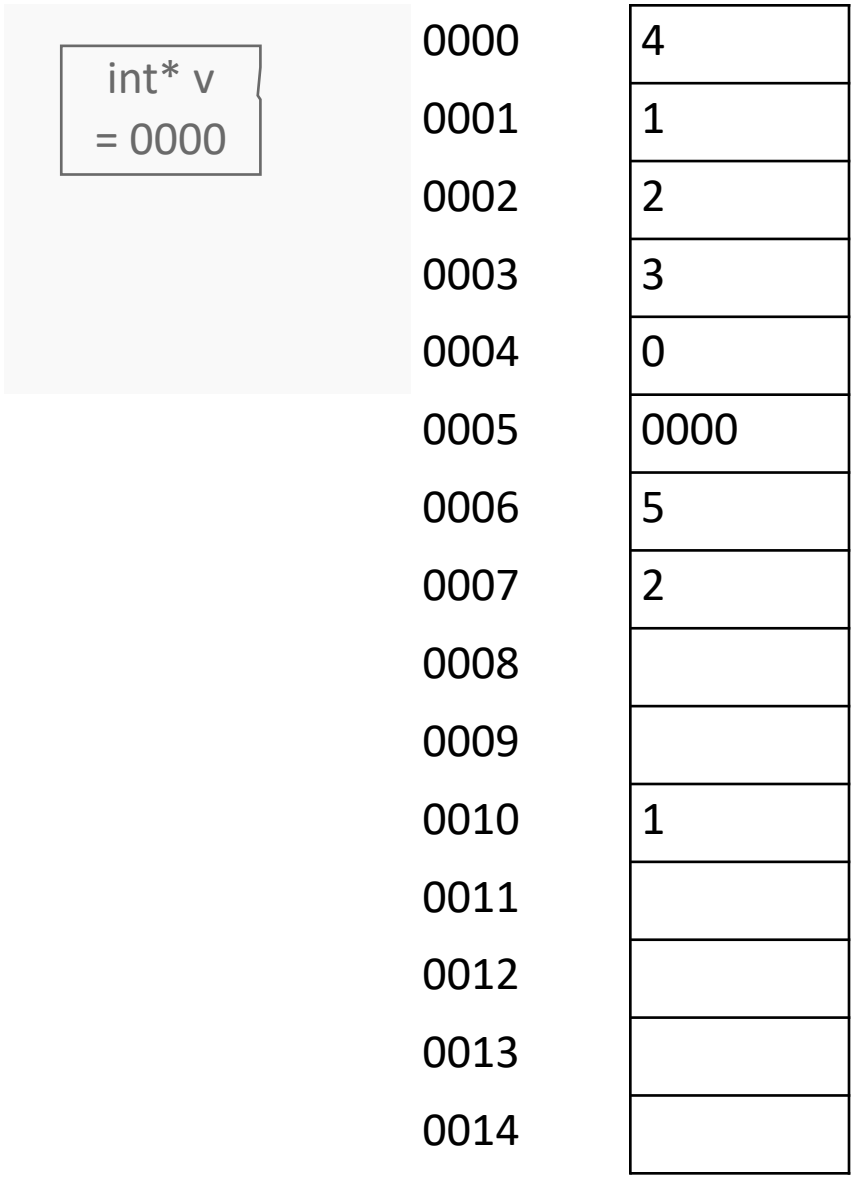
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



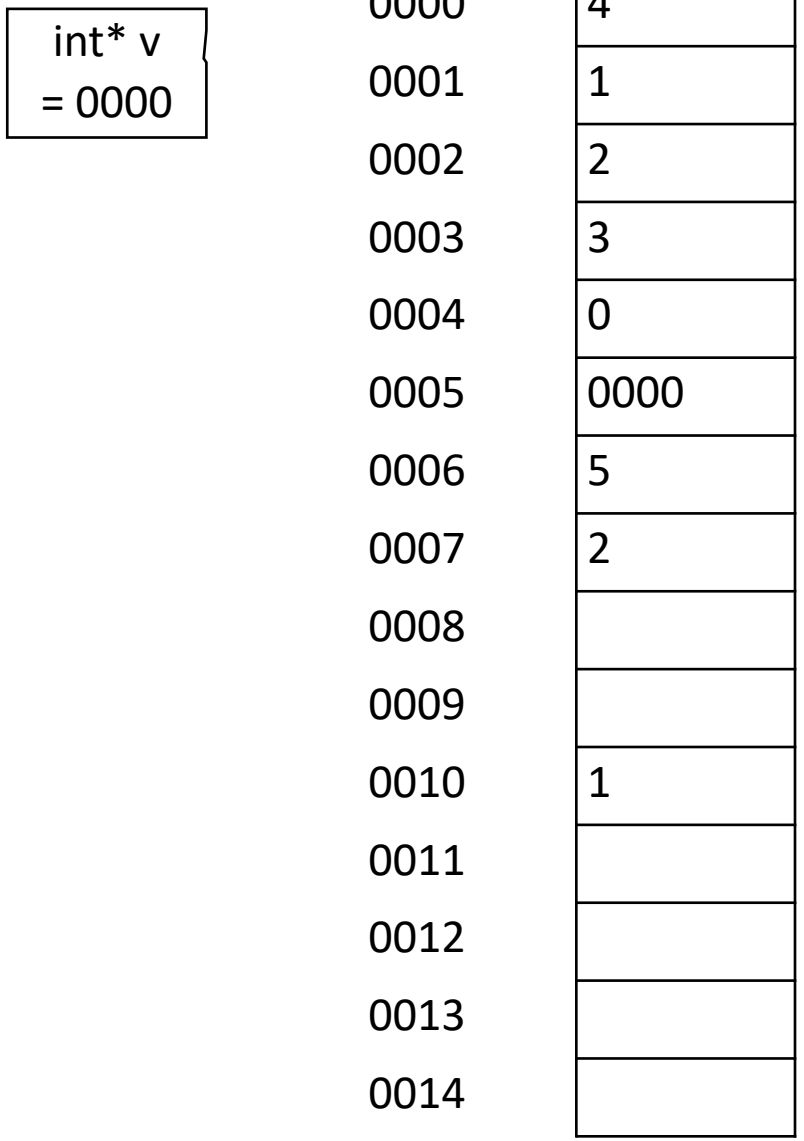
Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



Geheugenbeheer

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



Interne Implementatie van By Reference

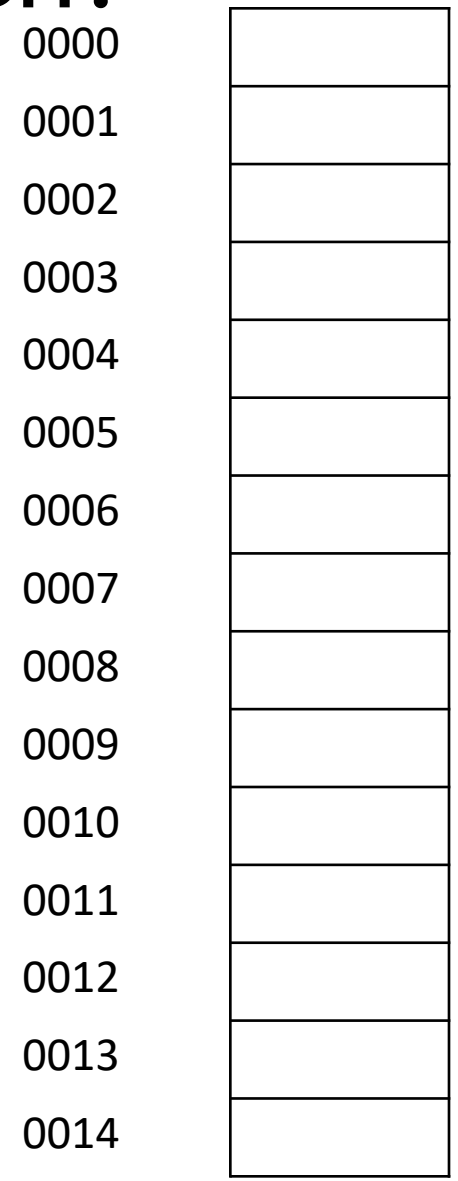
- By-reference variabele *kan* intern geïmplementeerd worden met pointers
  - int& x : achter de schermen wordt een pointer (int*) p doorgegeven
  - Argument lvalue wordt &lvalue
  - x=5 wordt dan *p=5

```
void swap(int &x, int &y) {
    int z=x;
    x=y; y=z;
}
```

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}
```

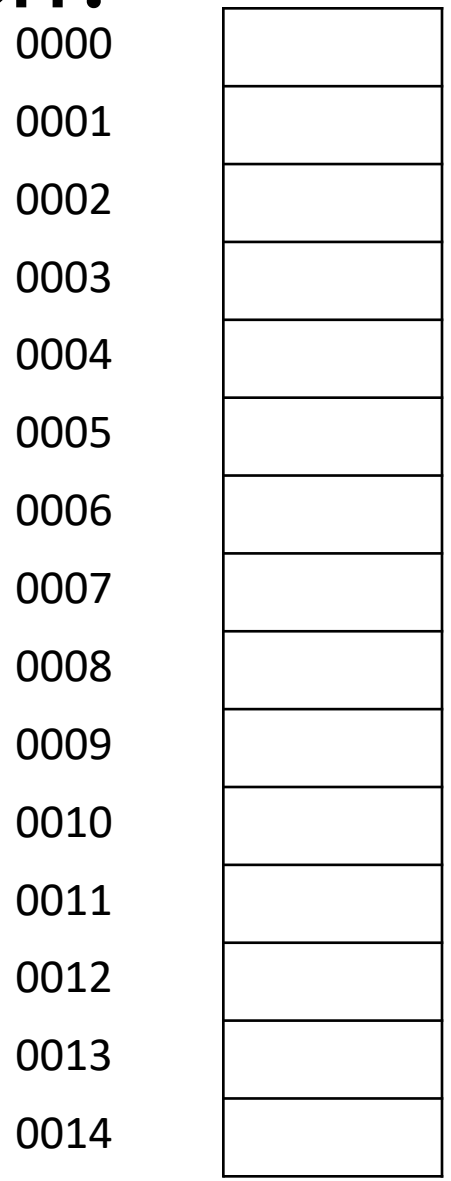
Wat met de by reference variabelen?

```
void swap(int&x, int& y) {
    int z=x;
    x=y; y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(a[i],a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



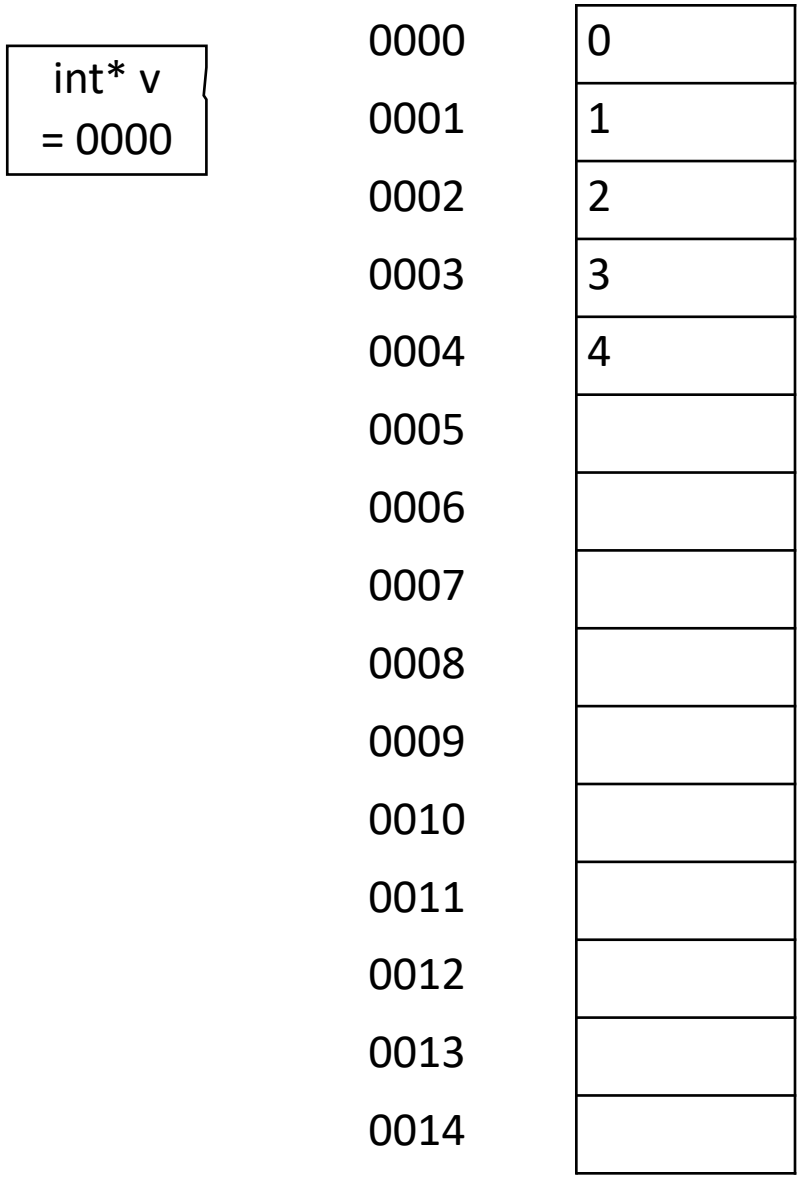
Wat met de by reference variabelen?

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



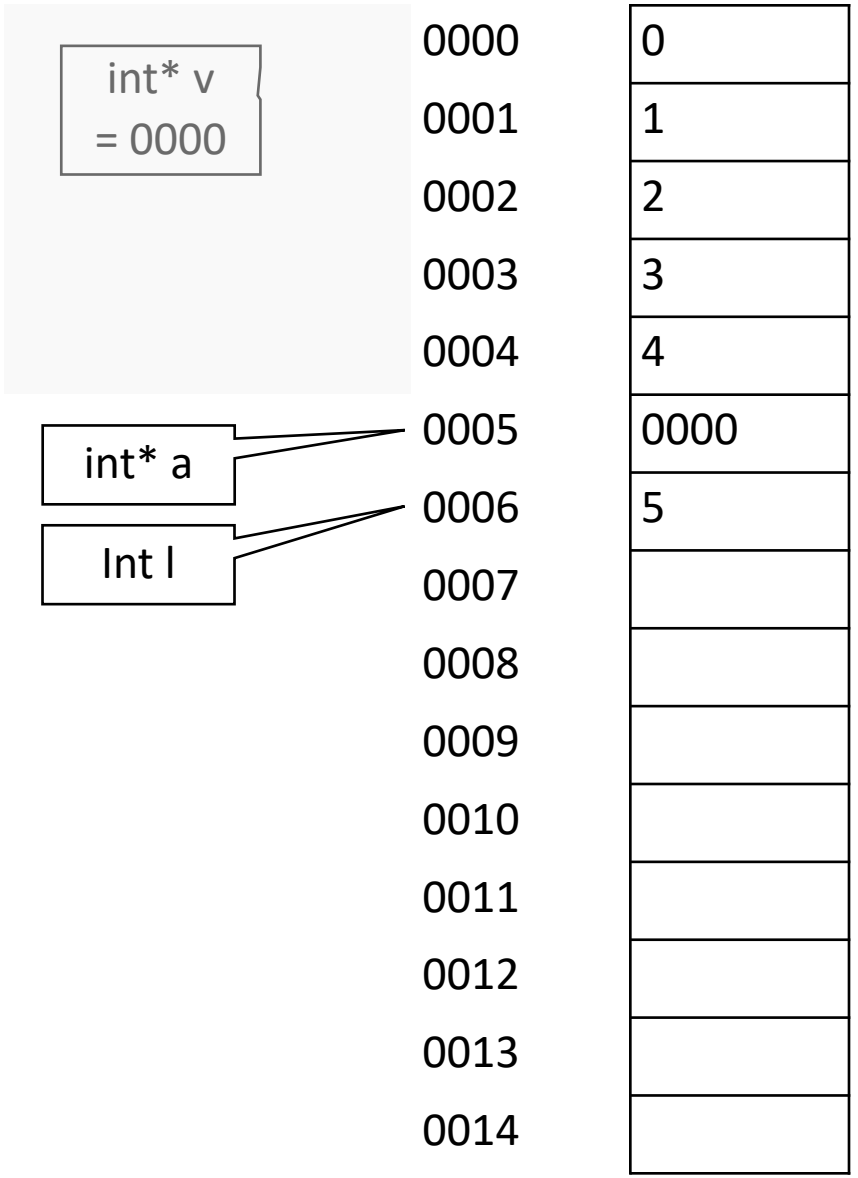
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



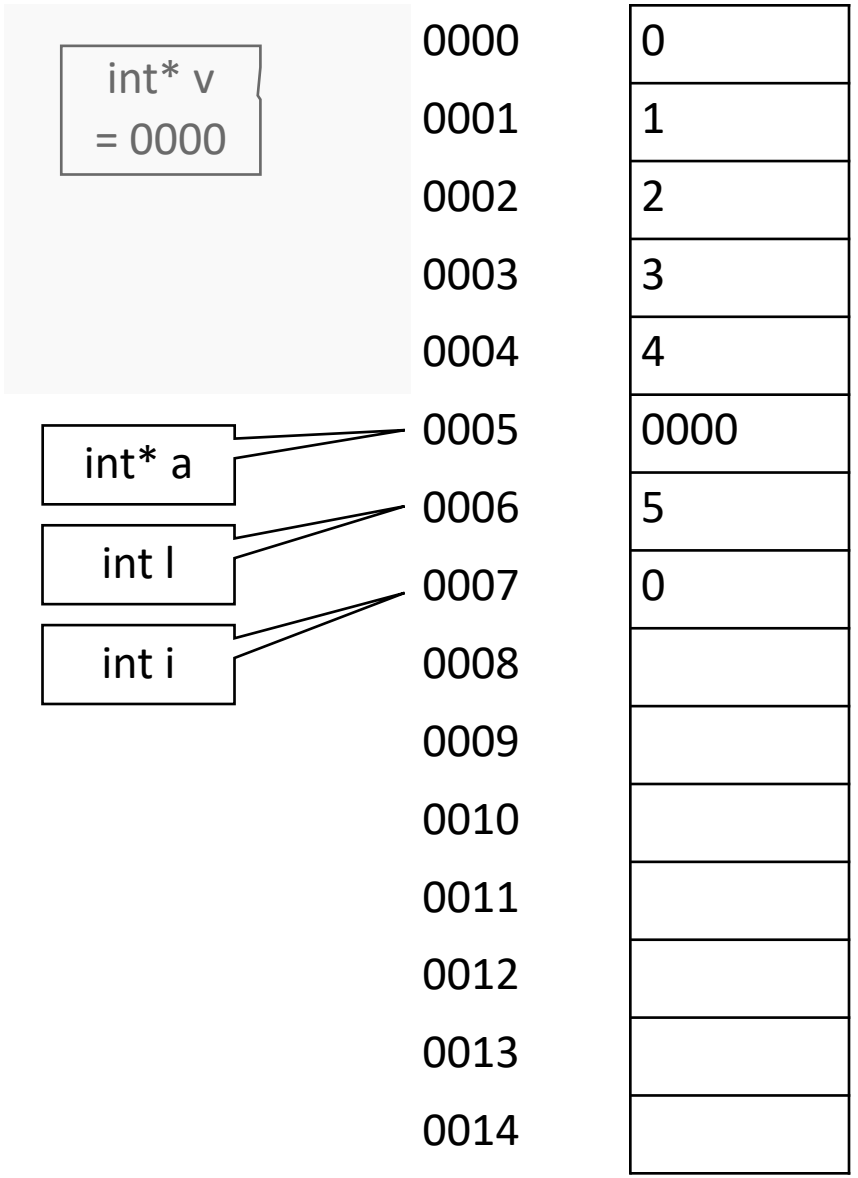
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



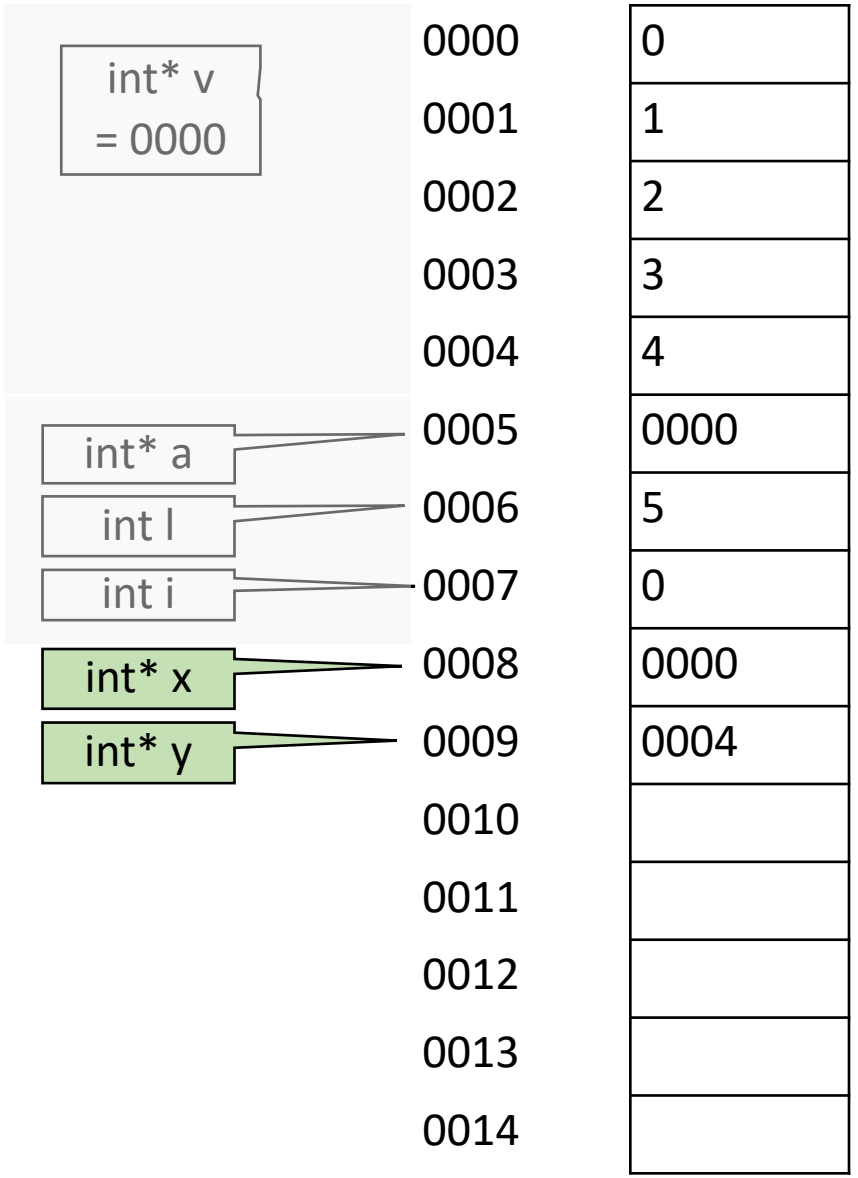
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



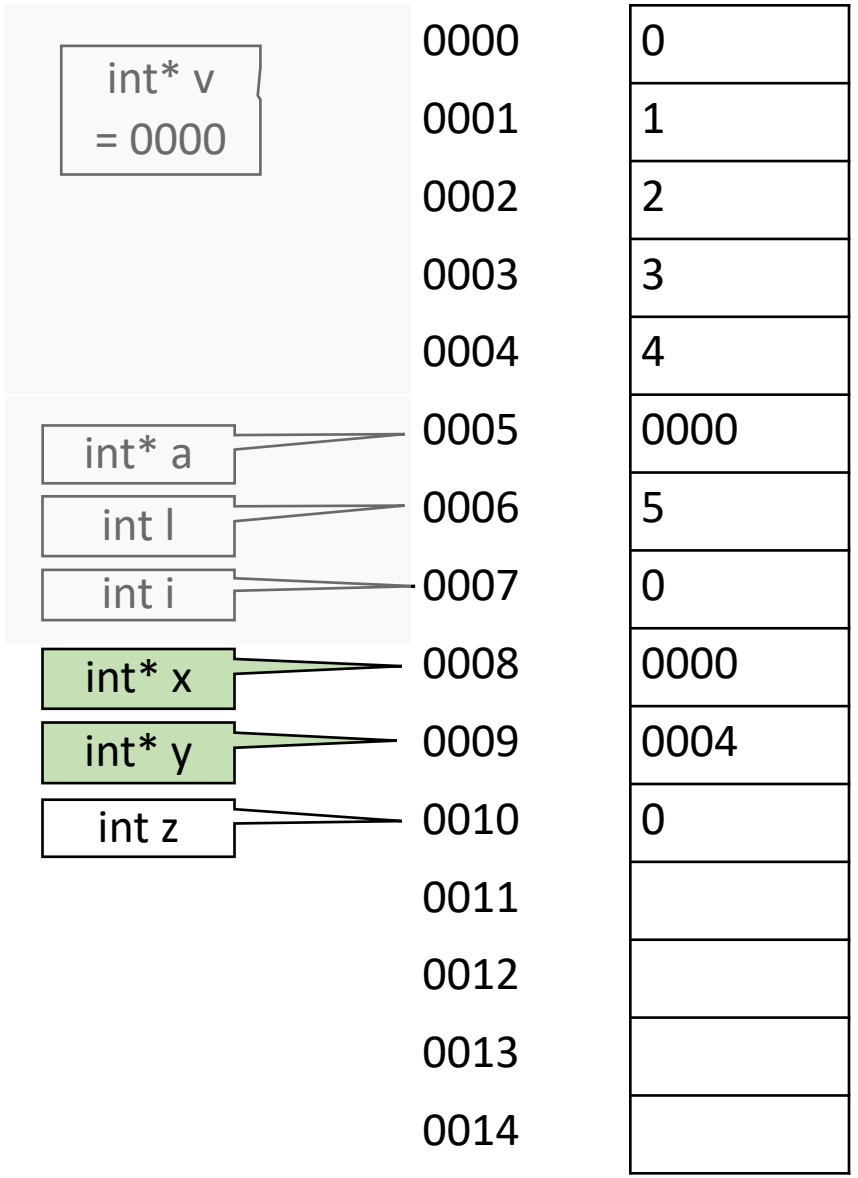
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



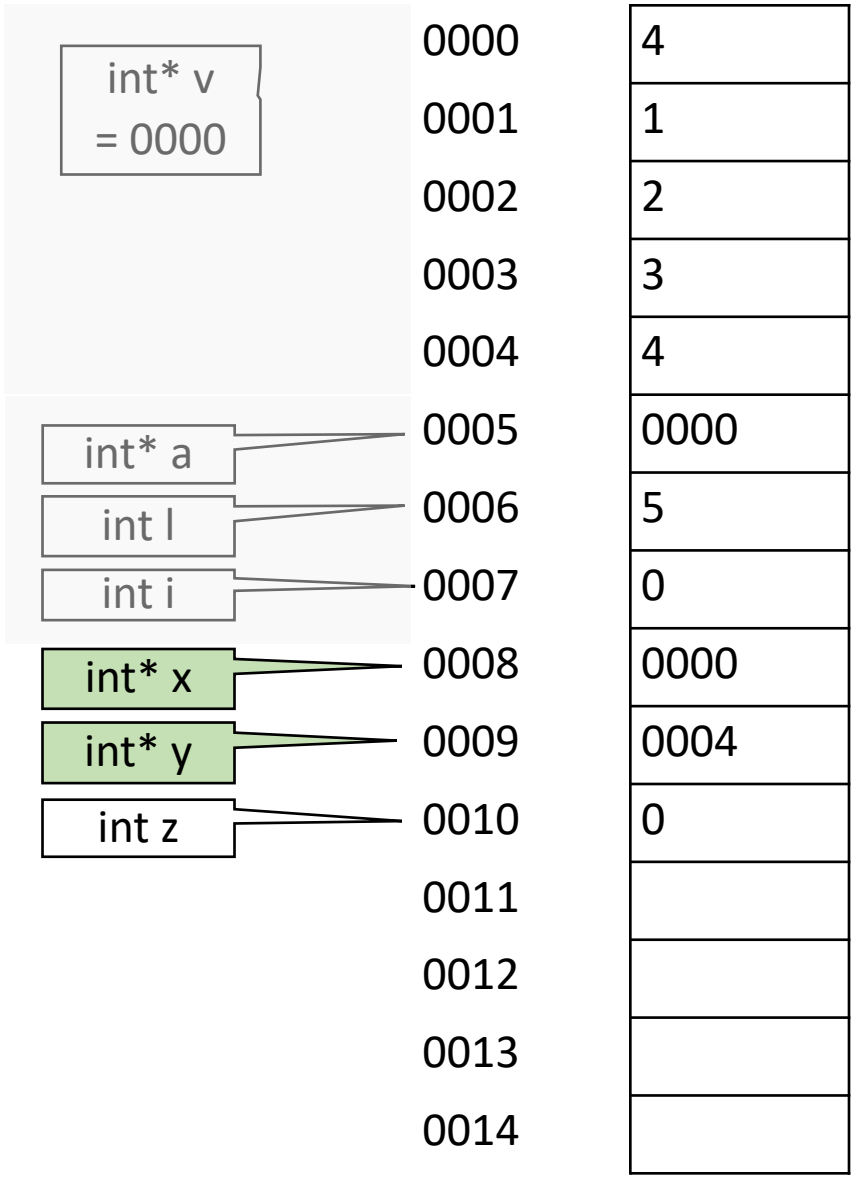
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



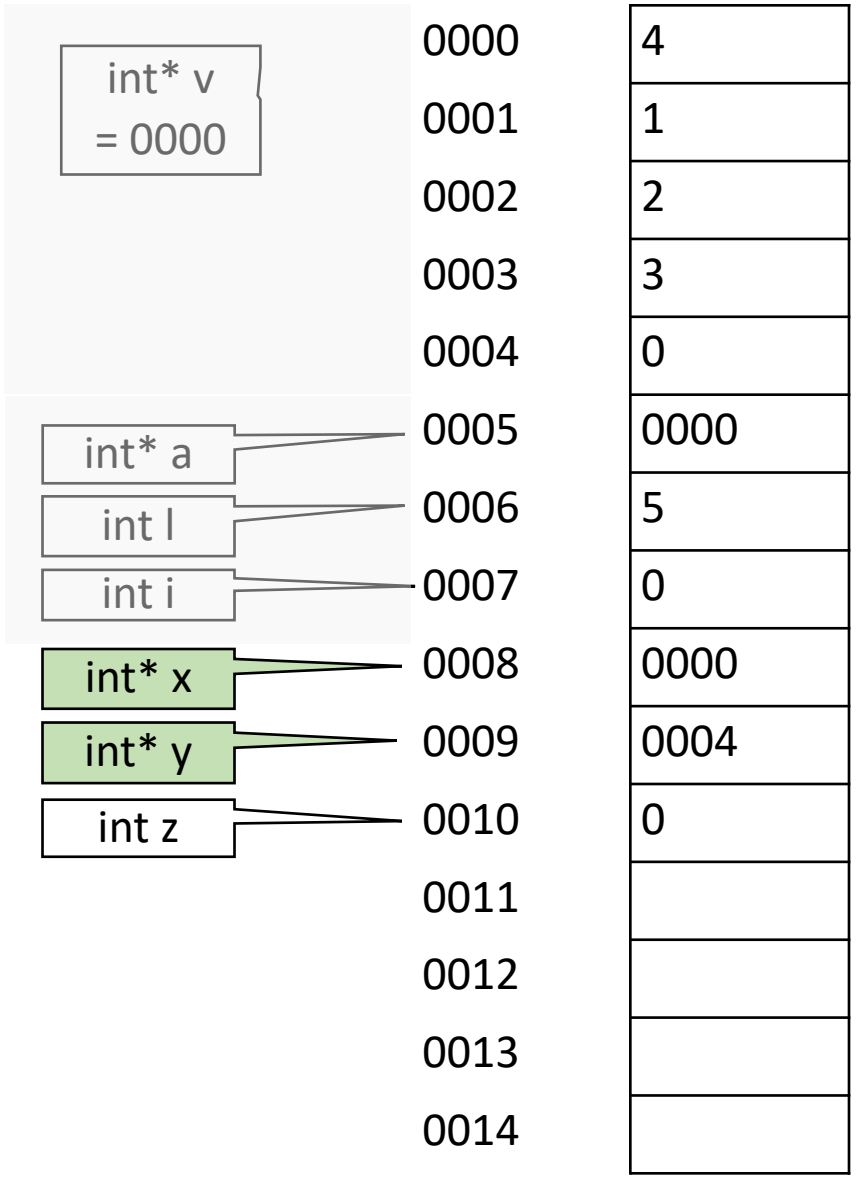
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



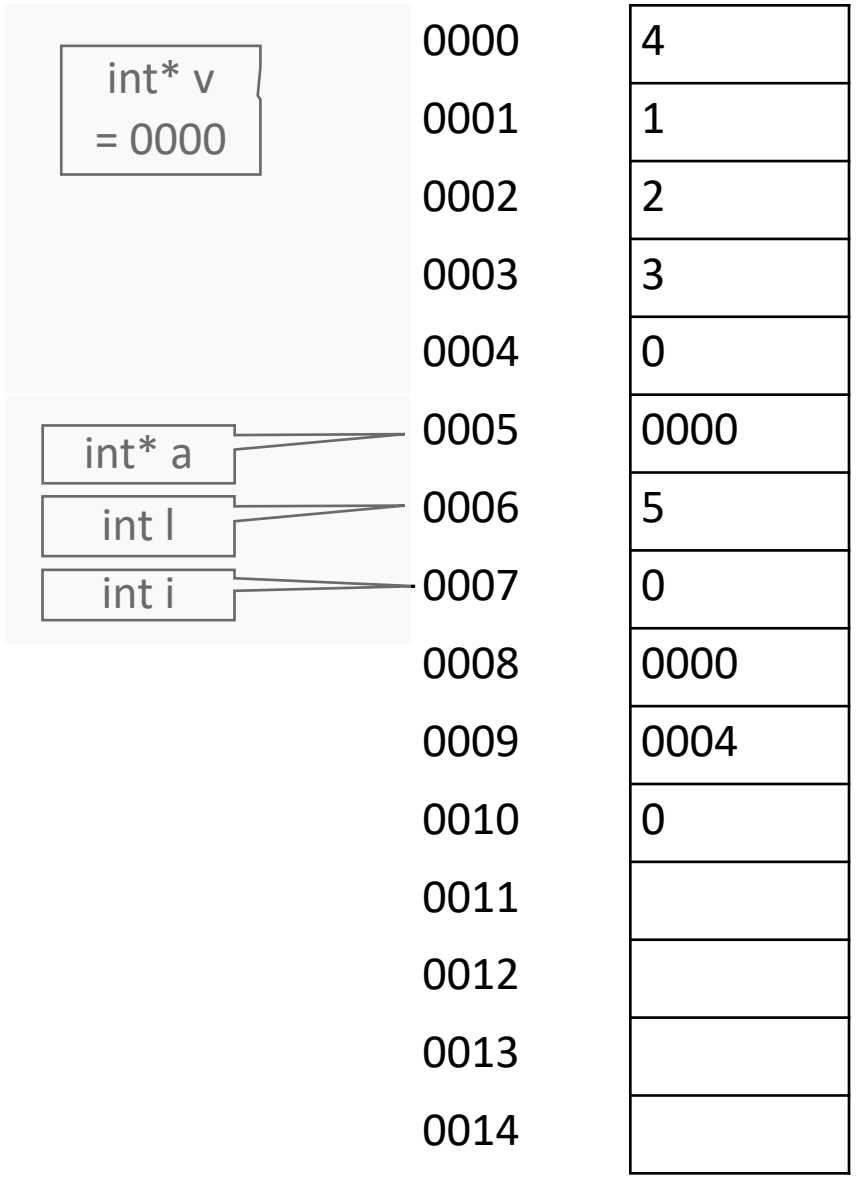
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



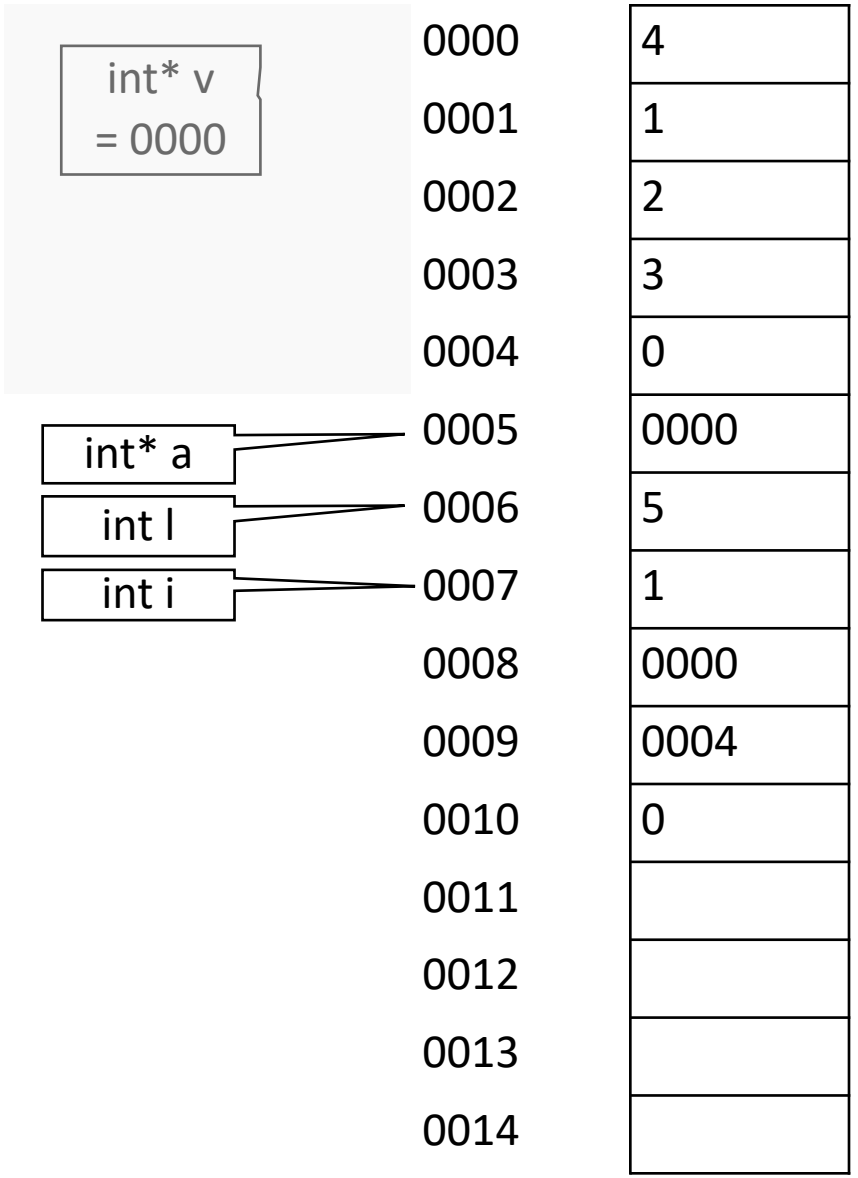
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



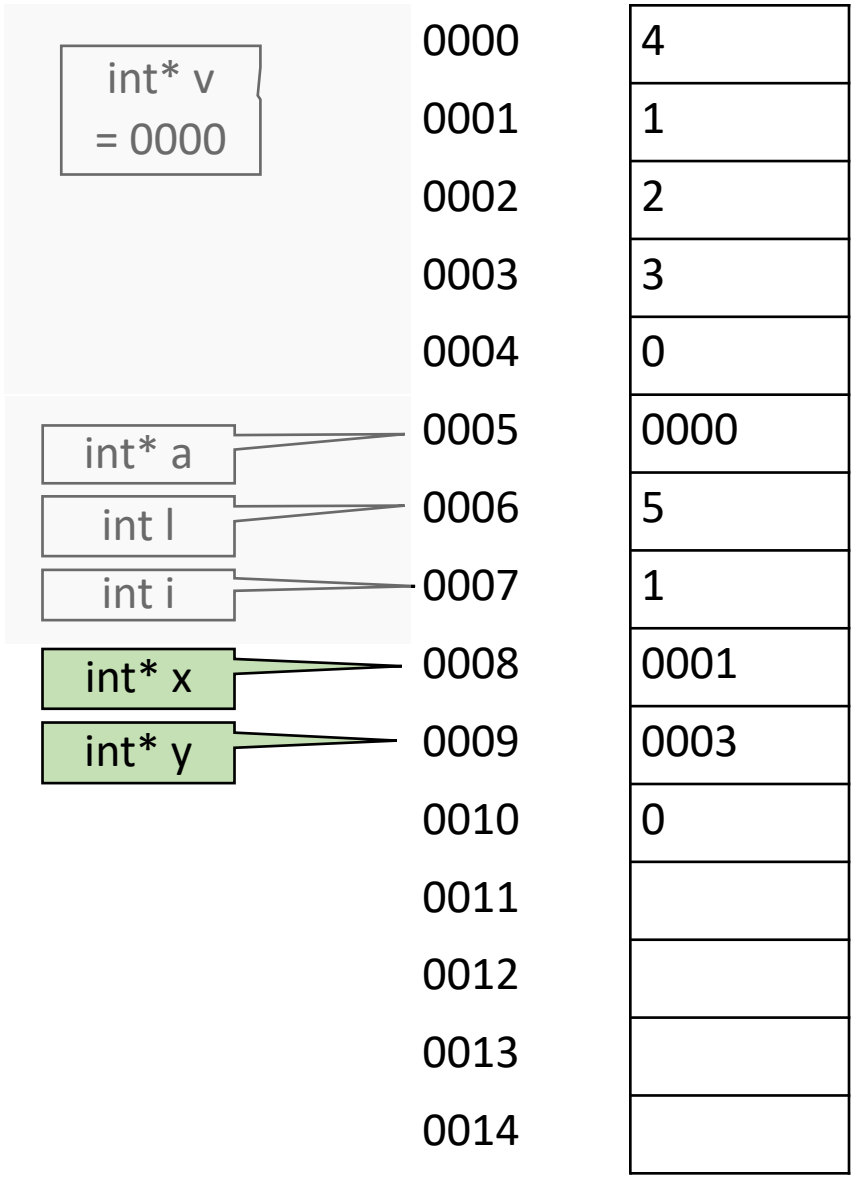
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



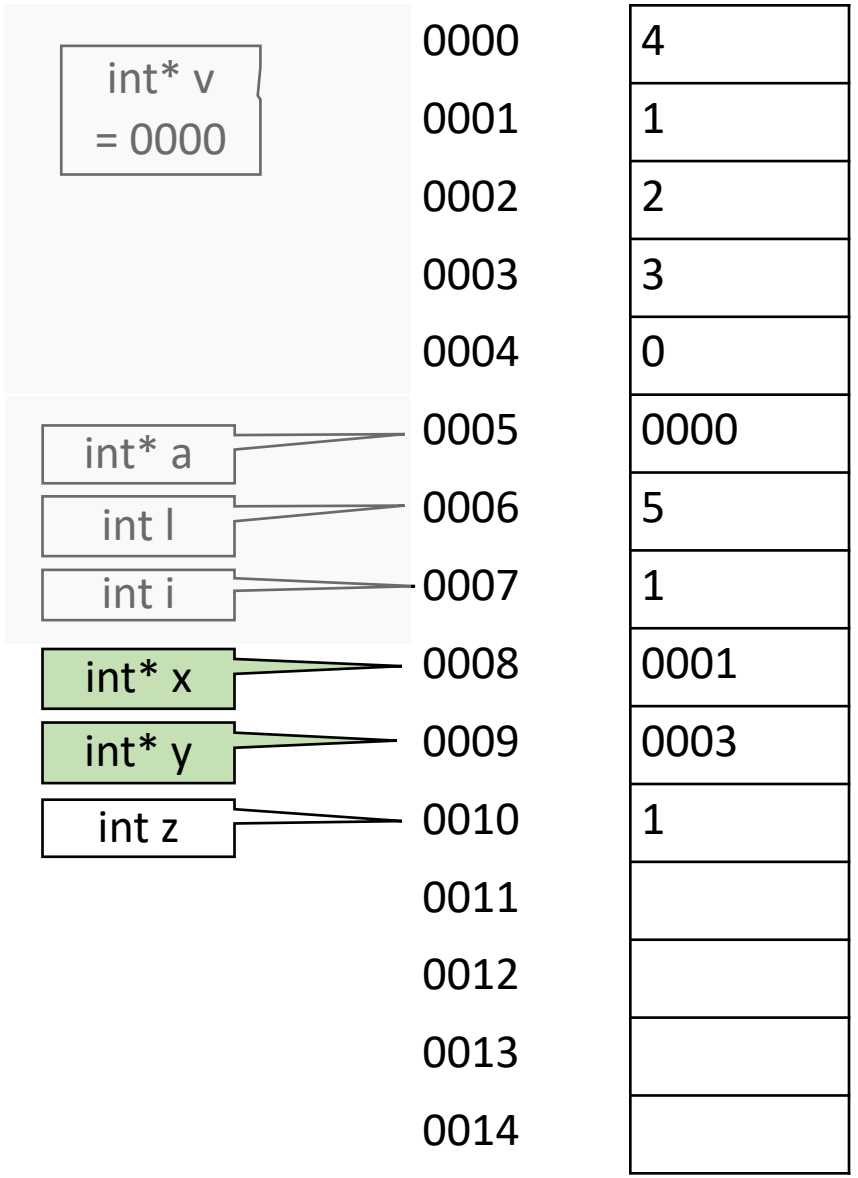
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



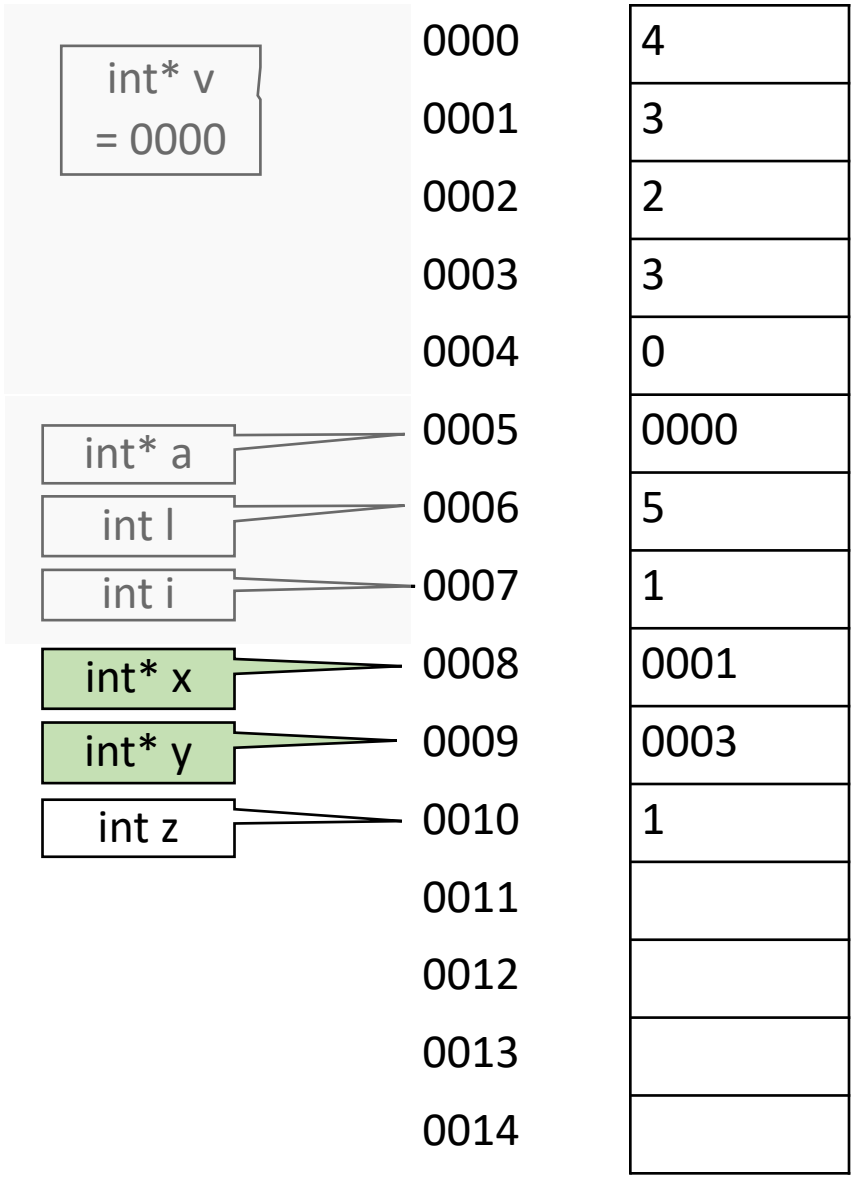
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



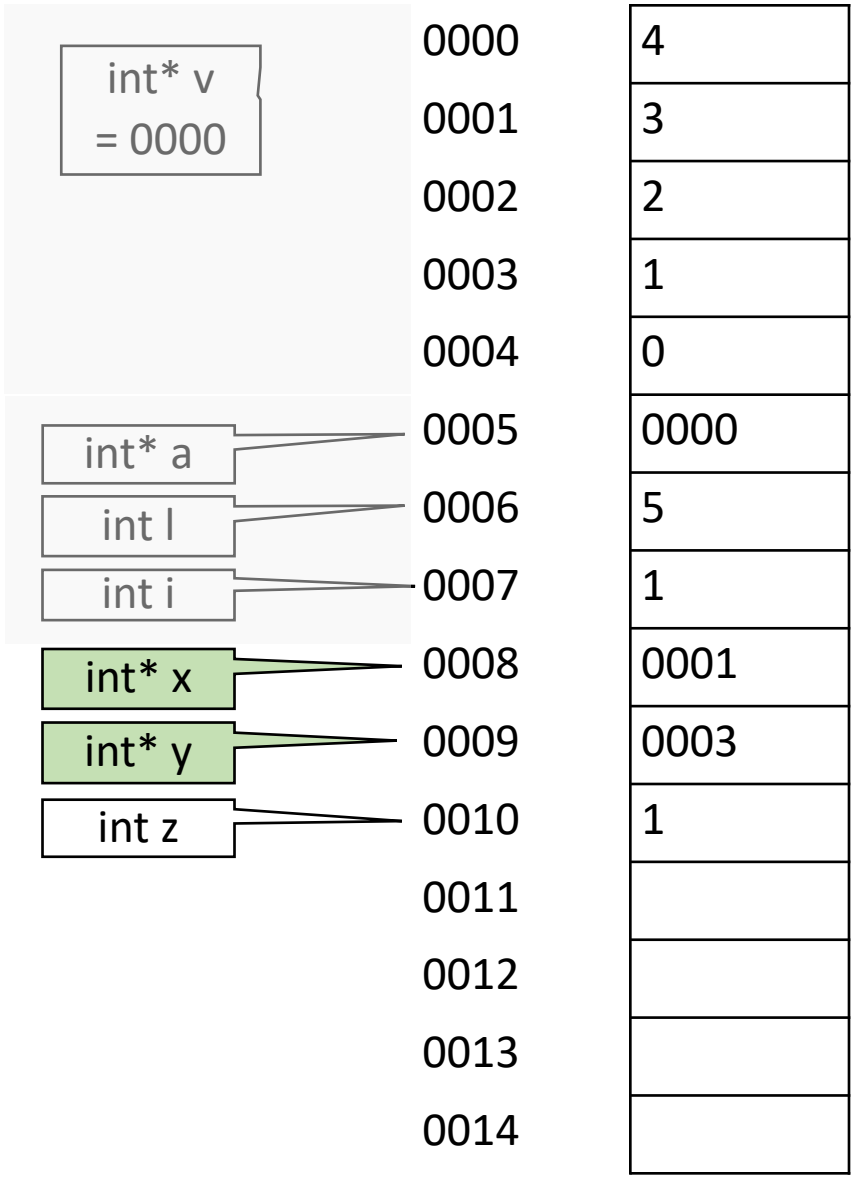
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



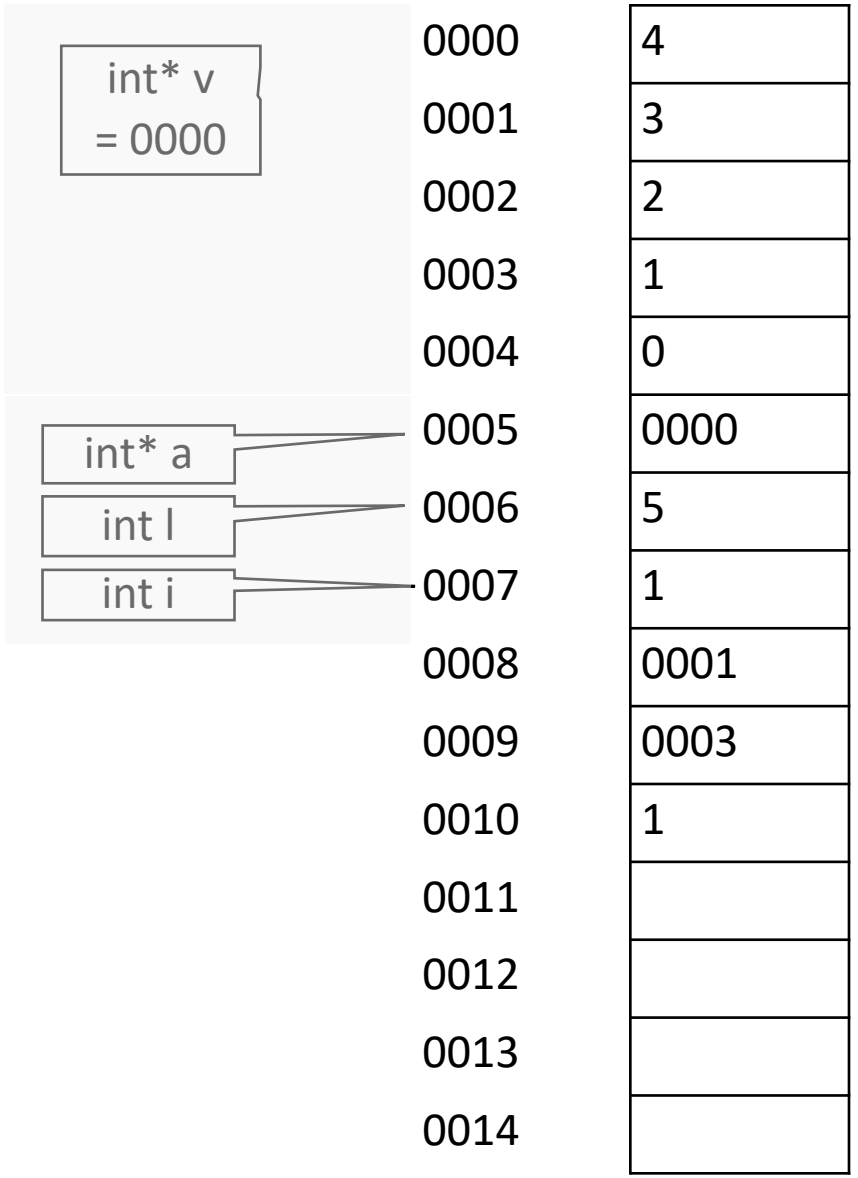
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



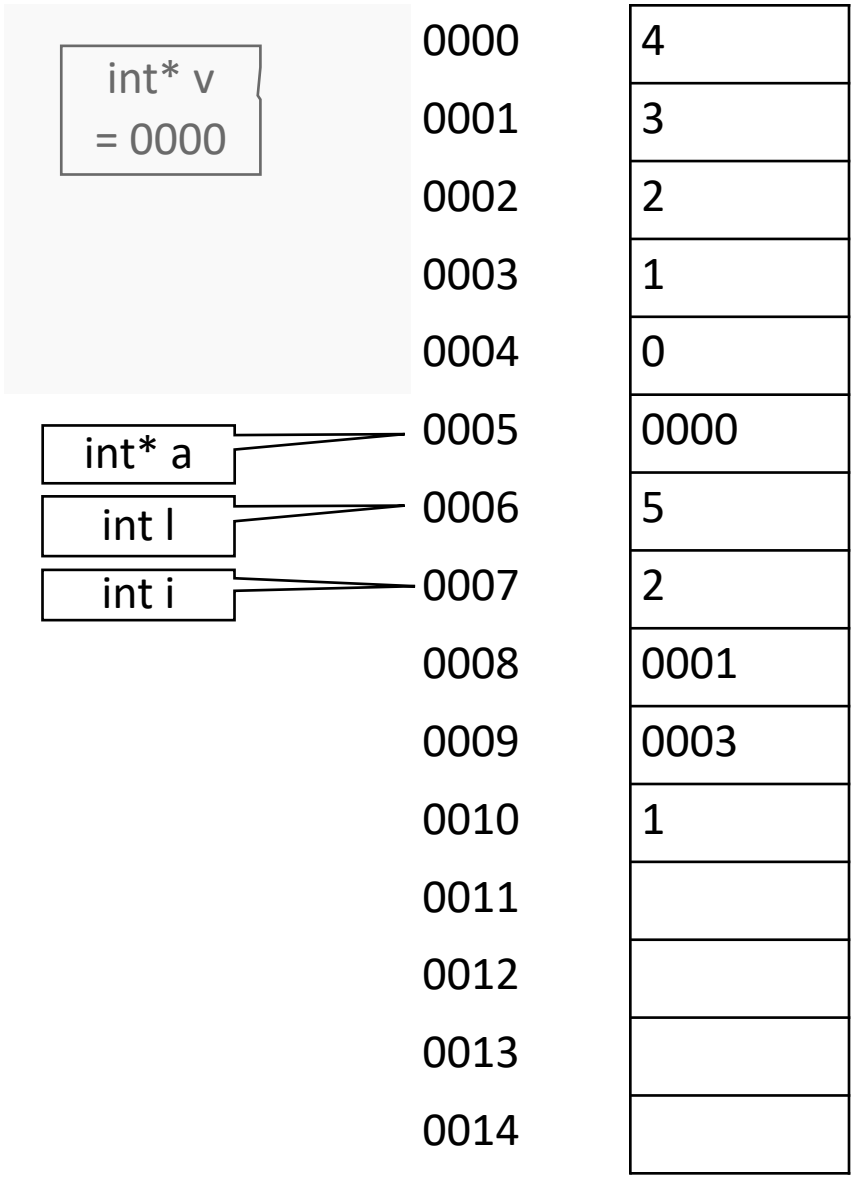
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



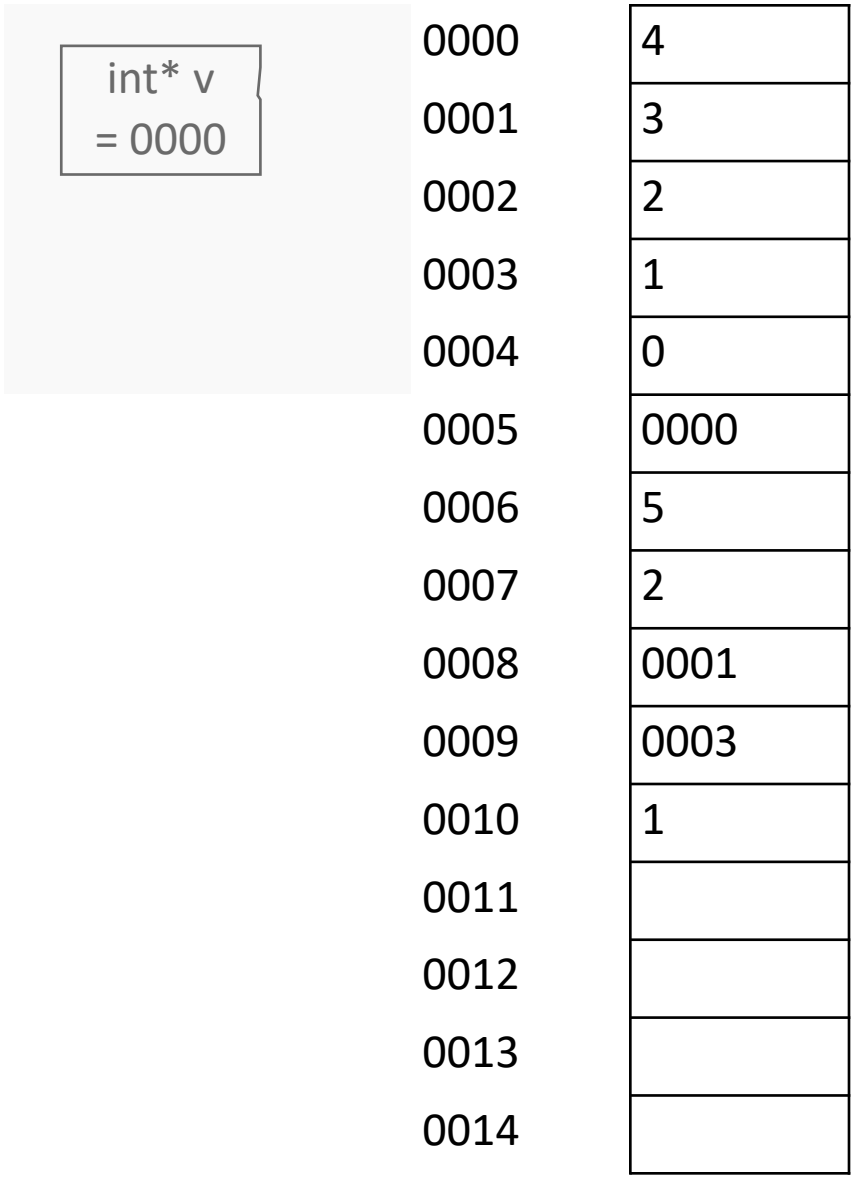
Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```



Geheugenbeheer

```
void swap(int* x, int* y) {
    int z=*x;
    *x=*y; *y=z;
}

void reverse(int* a, int l) {
    for (int i=0;i<l/2;i++)
        swap(&a[i],&a[l-1-i]);
}

int main() {
    int v[5]={0,1,2,3,4};
    reverse(v,5);
    return 0;
}
```

int* v
= 0000

0000	4
0001	3
0002	2
0003	1
0004	0
0005	0000
0006	5
0007	2
0008	0001
0009	0003
0010	1
0011	
0012	
0013	
0014	

Universiteit Antwerpen

Meer weten?

- Op stack wordt veel meer bewaard dan enkel lokale variabelen
  - Return value
  - “Book keeping” information:
    Pointers naar vorige stack pointer, base pointer, return address
- Stack pointer, base pointer: om at runtime te weten waar de lokale variabelen te vinden zijn

https://manybutfinite.com/post/journey-to-the-stack/
http://www.slideshare.net/saumilshah/how-functions-work-7776073