Lecture 3

- Pointers *(pekare)* [sec. 5.4.1-5.4.4]
  - Basics
  - Pointers to constants and constant pointers
  - Pointers and arrays *(pekare och fält)* [sec. 5.4.3]
  - Pointer arithmetic (*pekararitmetik)*

Variables (constants)

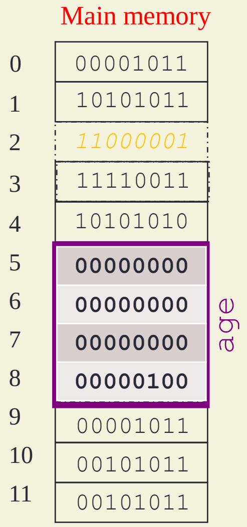
```
int age;
age = 4;
```

Each variable has some bytes in the memory reserved for it

same as

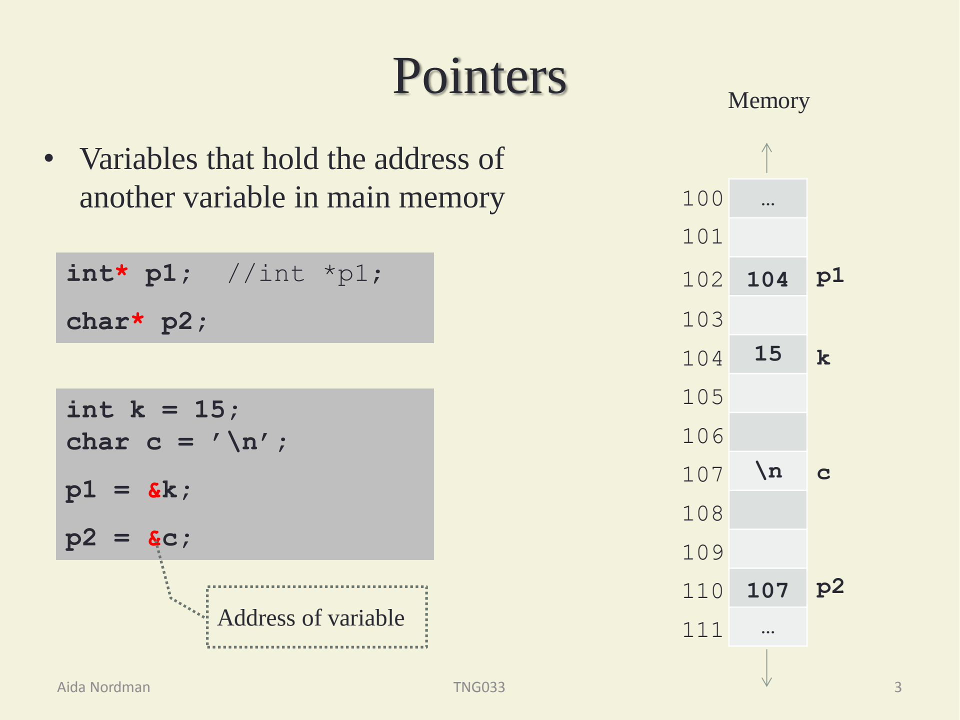
`int age = 4;` -- declare and initialize

- All variables have
  - A name age
  - A type int
  - A size in bytes 4 bytes
  - A memory address 5
  - A value 4



Main memory

0	00001011
1	10101011
2	*11000001*
3	11110011
4	10101010
5	**00000000**
6	**00000000**
7	**00000000**
8	**00000100**
9	00001011
10	00101011
11	00101011



Pointers

- Variables that hold the address of another variable in main memory

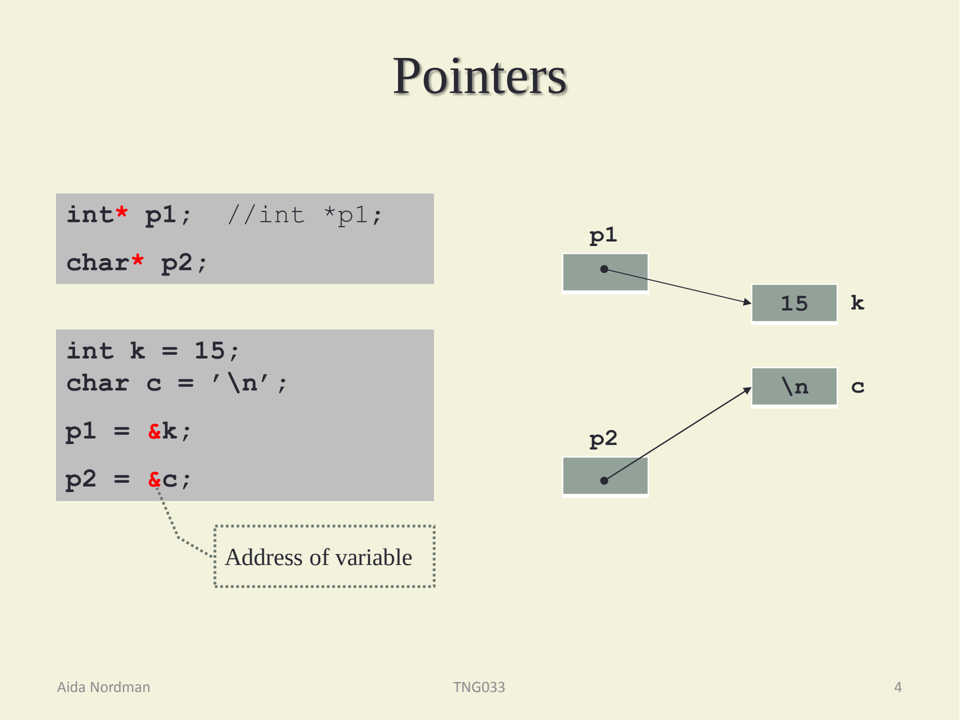
```
int* p1;   //int *p1;
char* p2;
```

```
int k = 15;
char c = '\n';
p1 = &k;
p2 = &c;
```

Address of variable

Memory

100	…	
101		
102	104	p1
103		
104	15	k
105		
106		
107	\n	c
108		
109		
110	107	p2
111	…	

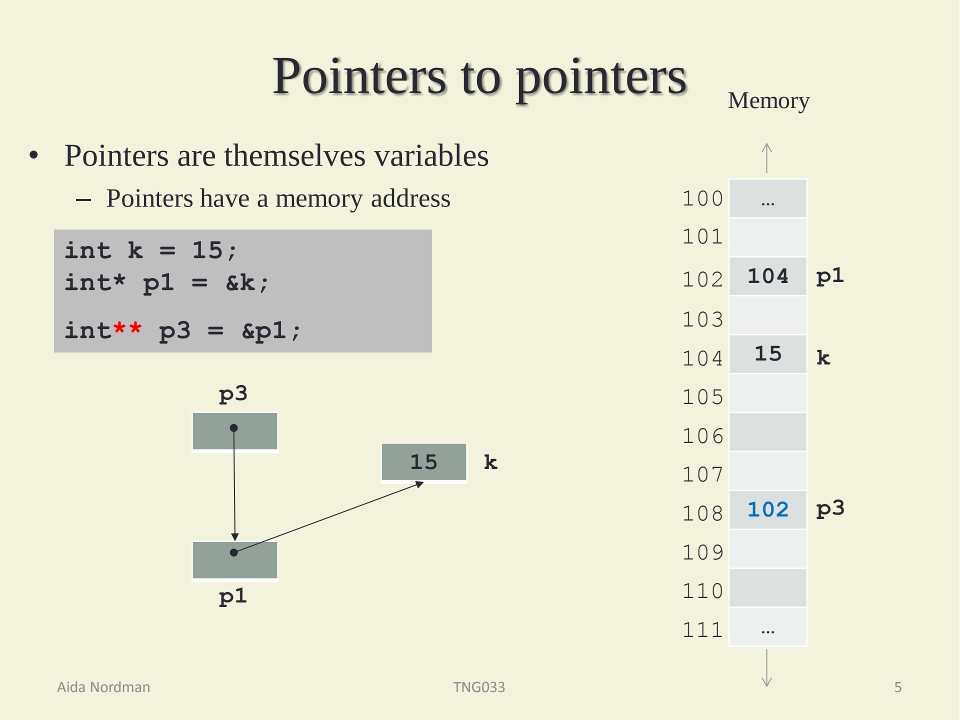


Pointers

```
int* p1;   //int *p1;
char* p2;
```

```
int k = 15;
char c = '\n';
p1 = &k;
p2 = &c;
```

Address of variable



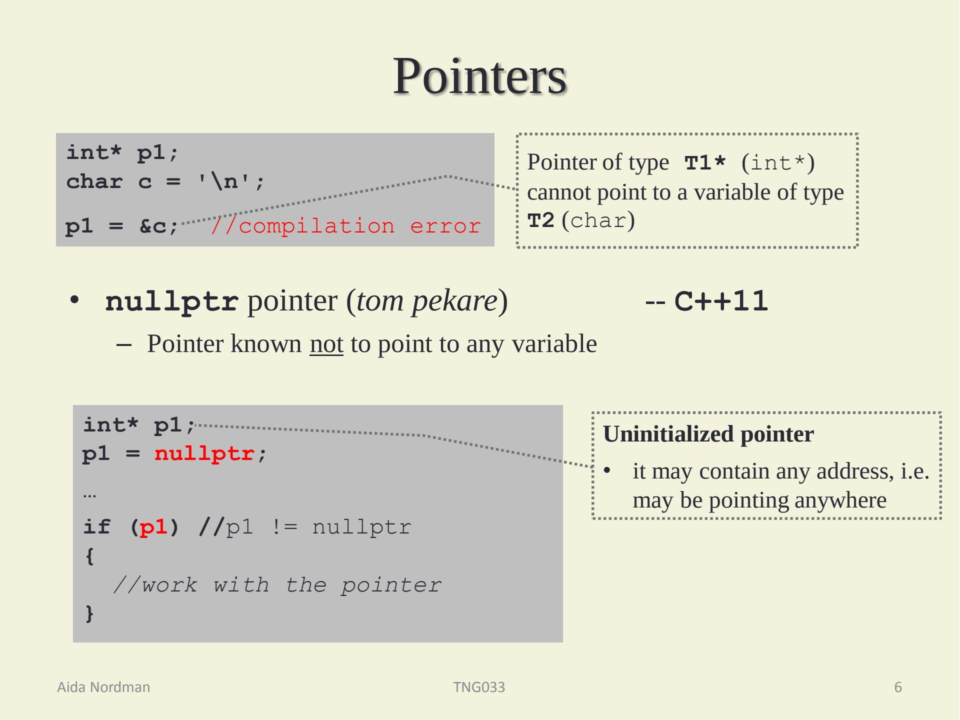
Pointers to pointers

- Pointers are themselves variables
  - Pointers have a memory address

```
int k = 15;
int* p1 = &k;

int** p3 = &p1;
```

Memory

100	…	
101		
102	104	p1
103		
104	15	k
105		
106		
107		
108	102	p3
109		
110		
111	…	



Pointers

```
int* p1;
char c = '\n';

p1 = &c;   //compilation error
```

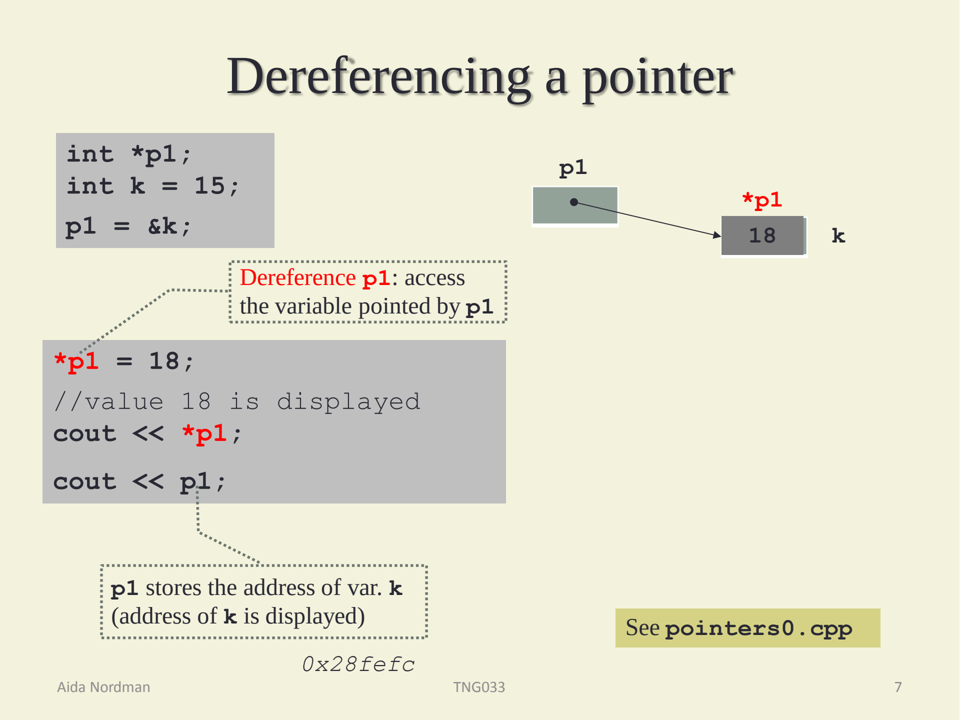
Pointer of type **T1*** (int*) cannot point to a variable of type **T2** (char)

- **nullptr** pointer (*tom pekare*) -- **C++11**
  - Pointer known not to point to any variable

```
int* p1;
p1 = nullptr;
…
if (p1) //p1 != nullptr
{
  //work with the pointer
}
```

Uninitialized pointer

- it may contain any address, i.e. may be pointing anywhere



Dereferencing a pointer

```
int *p1;
int k = 15;
p1 = &k;
```

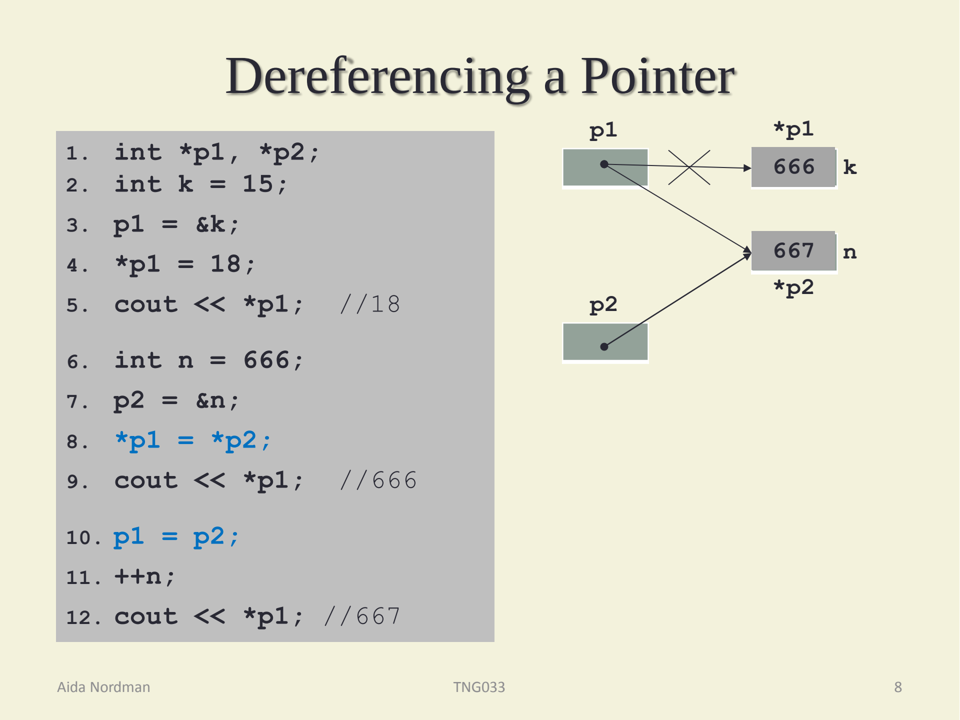
Dereference **p1**: access the variable pointed by **p1**

```
*p1 = 18;
//value 18 is displayed
cout << *p1;

cout << p1;
```

p1 stores the address of var. **k** (address of **k** is displayed)

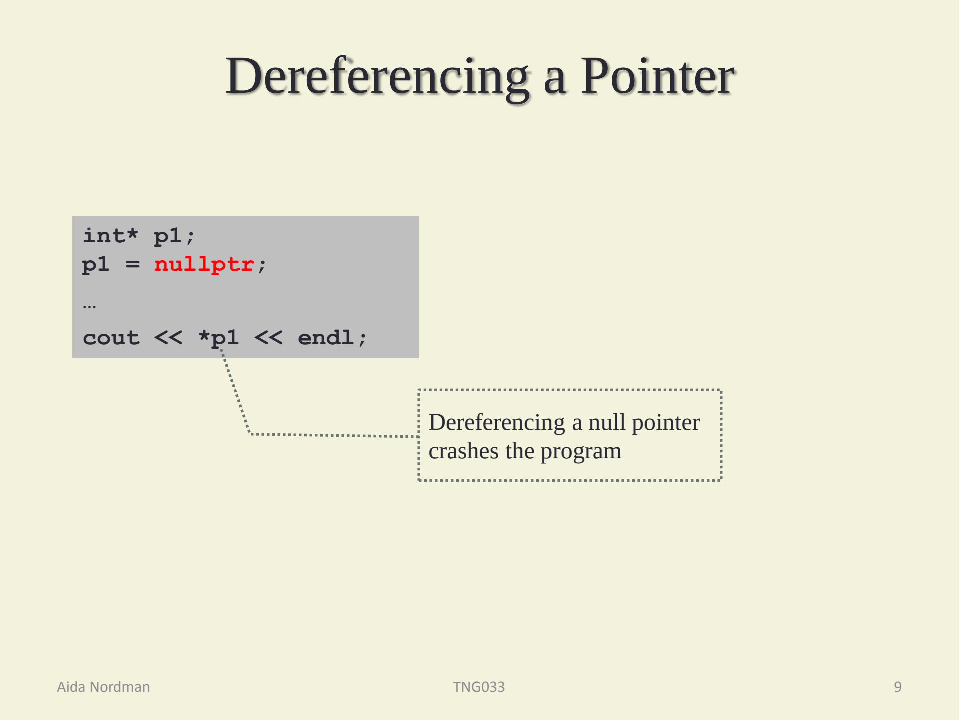
0x28fefc

See **pointers0.cpp**



Dereferencing a Pointer

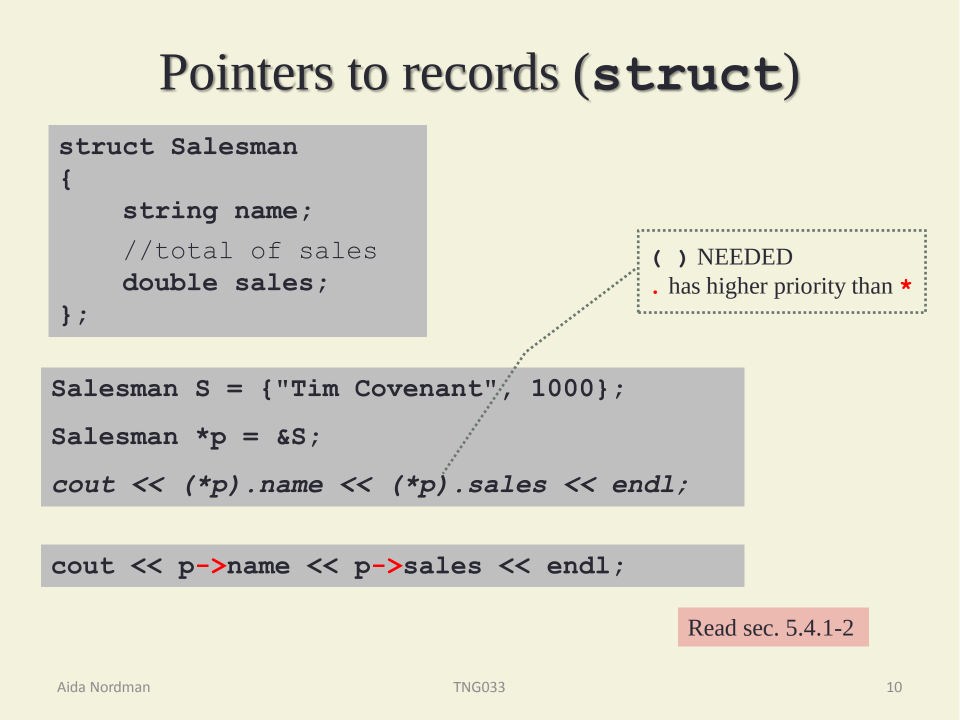
```
int *p1, *p2;
int k = 15;
p1 = &k;
*p1 = 18;
cout << *p1;   //18

int n = 666;
p2 = &n;
*p1 = *p2;
cout << *p1;   //666

p1 = p2;
++n;
cout << *p1;  //667
```



Dereferencing a Pointer

```
int* p1;
p1 = nullptr;
…
cout << *p1 << endl;
```

Dereferencing a null pointer crashes the program



Pointers to records (struct)

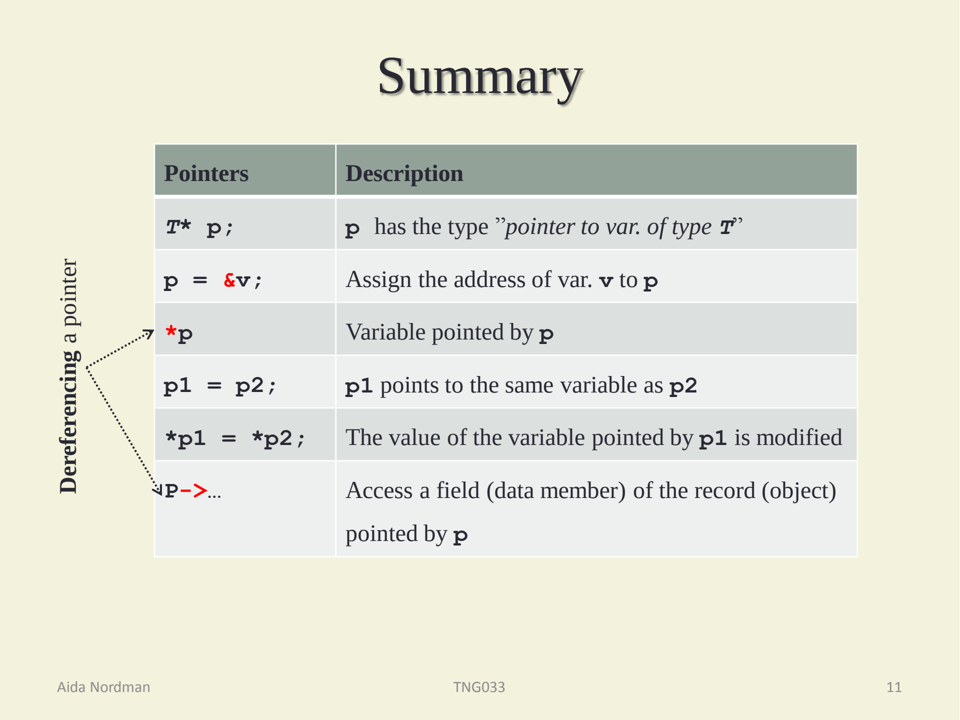
```
struct Salesman
{
    string name;
    //total of sales
    double sales;
};
```

```
Salesman S = {"Tim Covenant", 1000};
Salesman *p = &S;
cout << (*p).name << (*p).sales << endl;
```

( ) NEEDED
. has higher priority than *

```
cout << p->name << p->sales << endl;
```

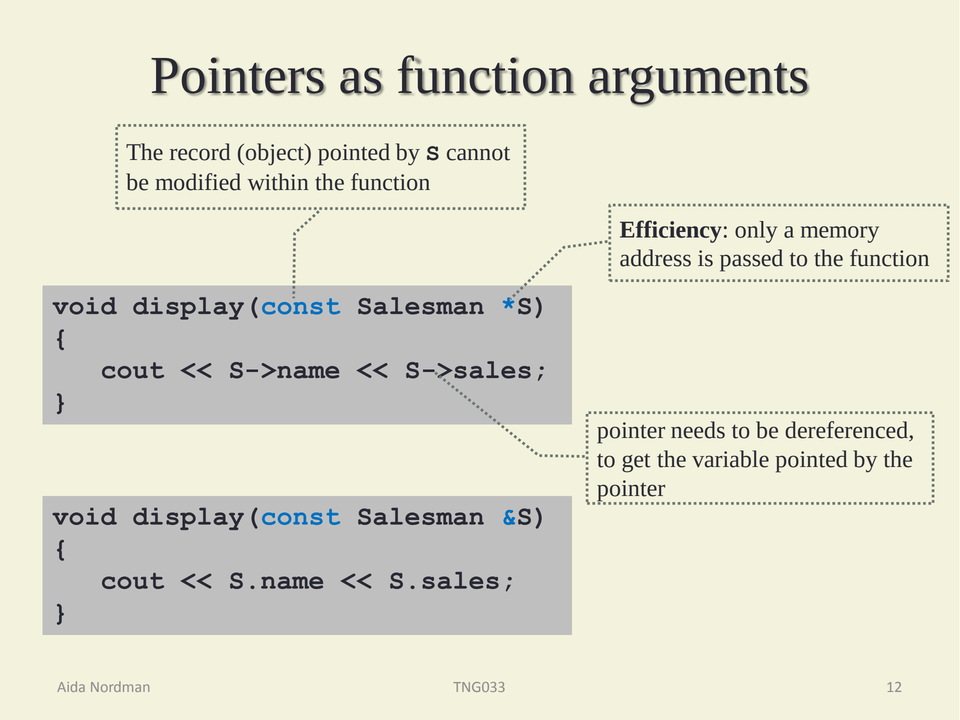
Read sec. 5.4.1-2



Summary

Pointers	Description
`T* p;`	`p` has the type ”*pointer to var. of type* `T`”
`p = &v;`	Assign the address of var. `v` to `p`
`*p`	Variable pointed by `p`
`p1 = p2;`	`p1` points to the same variable as `p2`
`*p1 = *p2;`	The value of the variable pointed by `p1` is modified
`p->`...	Access a field (data member) of the record (object) pointed by `p`

Dereferencing a pointer



Pointers as function arguments

The record (object) pointed by `S` cannot be modified within the function

Efficiency: only a memory address is passed to the function

```
void display(const Salesman *S)
{
   cout << S->name << S->sales;
}
```

pointer needs to be dereferenced, to get the variable pointed by the pointer

```
void display(const Salesman &S)
{
   cout << S.name << S.sales;
}
```

Pointers as function arguments

Version 1: call by reference

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

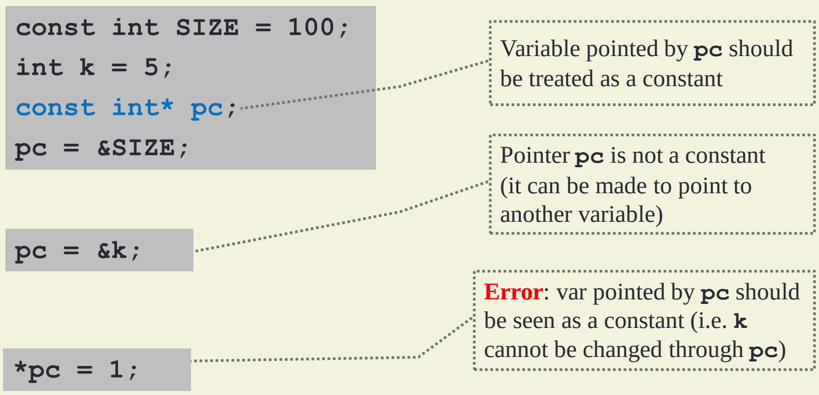
```
int main()
{
    int a = 3, b = 4;
    swap(a, b);
    return 0;
}
```

Version 2: call by reference using pointers

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

```
int main()
{
    int a = 3, b = 4;
    swap(&a, &b);
    return 0;
}
```

Pointers to constants



```
const int SIZE = 100;
int k = 5;
const int* pc;
pc = &SIZE;
```

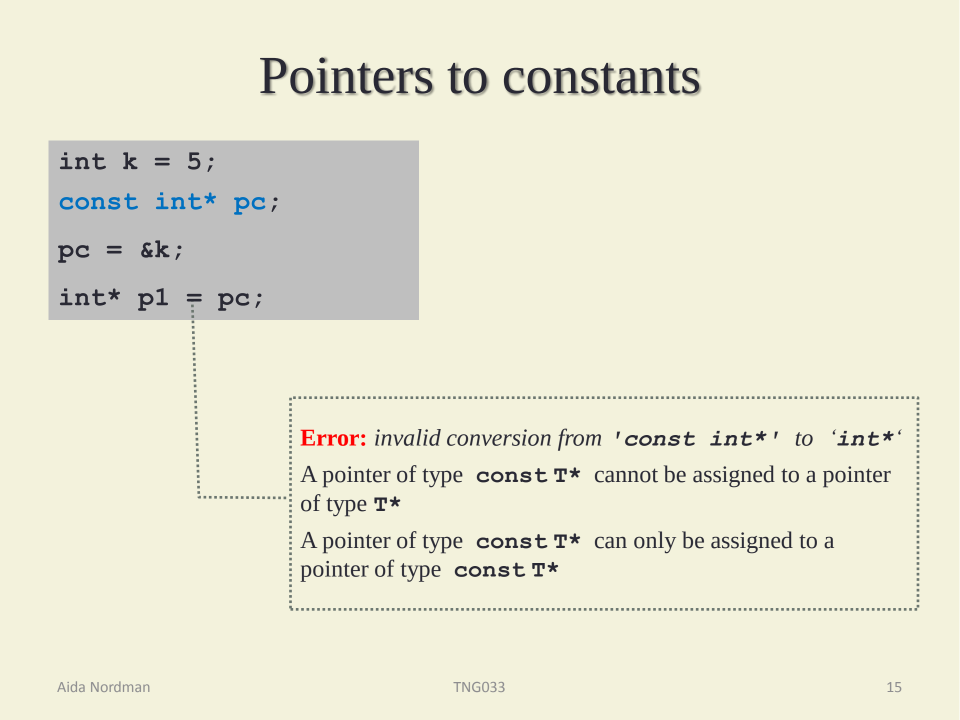
Variable pointed by **pc** should be treated as a constant

```
pc = &k;
```

Pointer **pc** is not a constant (it can be made to point to another variable)

```
*pc = 1;
```

Error: var pointed by **pc** should be seen as a constant (i.e. **k** cannot be changed through **pc**)



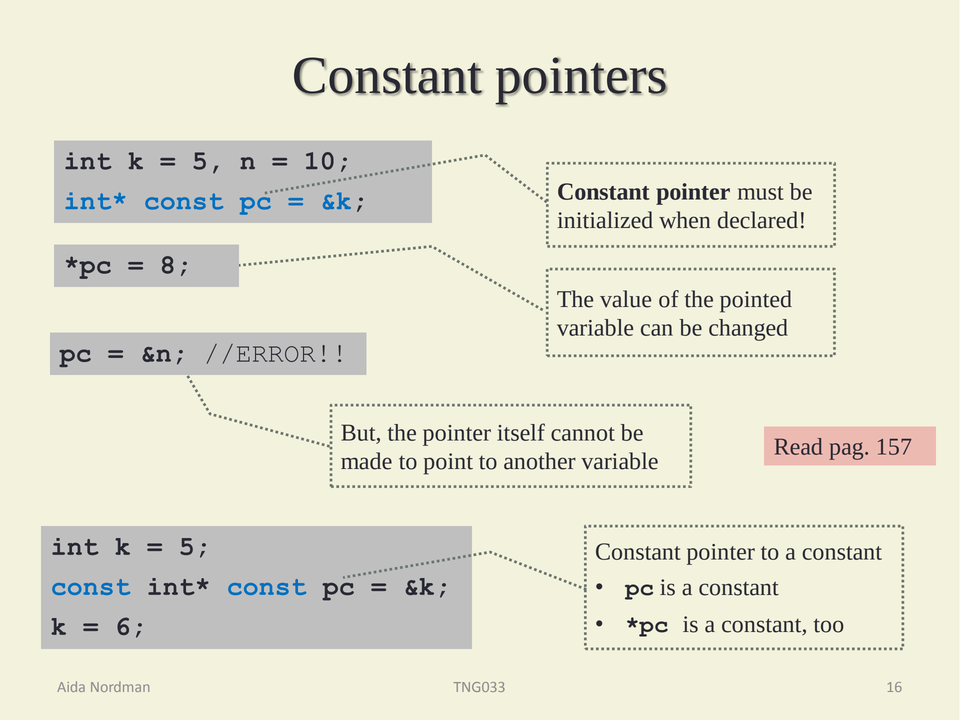
Pointers to constants

```
int k = 5;
const int* pc;
pc = &k;
int* p1 = pc;
```

Error: *invalid conversion from* `'const int*'` *to* `‘int*‘`

A pointer of type `const T*` cannot be assigned to a pointer of type `T*`

A pointer of type `const T*` can only be assigned to a pointer of type `const T*`



Constant pointers

```
int k = 5, n = 10;
int* const pc = &k;
```

Constant pointer must be initialized when declared!

```
*pc = 8;
```

The value of the pointed variable can be changed

```
pc = &n; //ERROR!!
```

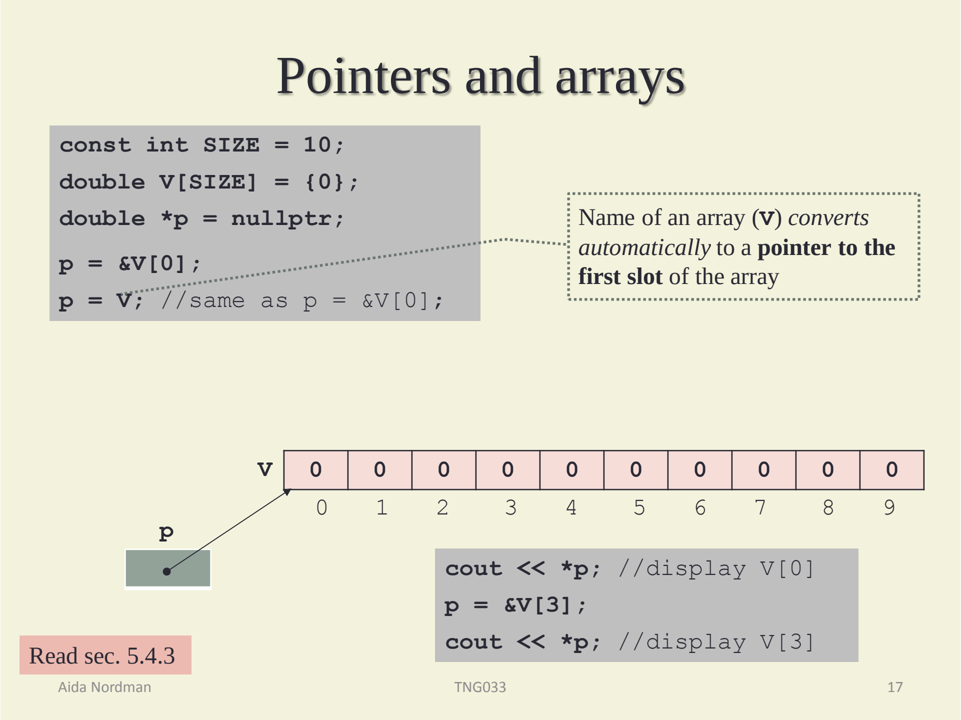
But, the pointer itself cannot be made to point to another variable

Read pag. 157

```
int k = 5;
const int* const pc = &k;
k = 6;
```

Constant pointer to a constant

- `pc` is a constant
- `*pc` is a constant, too



Pointers and arrays

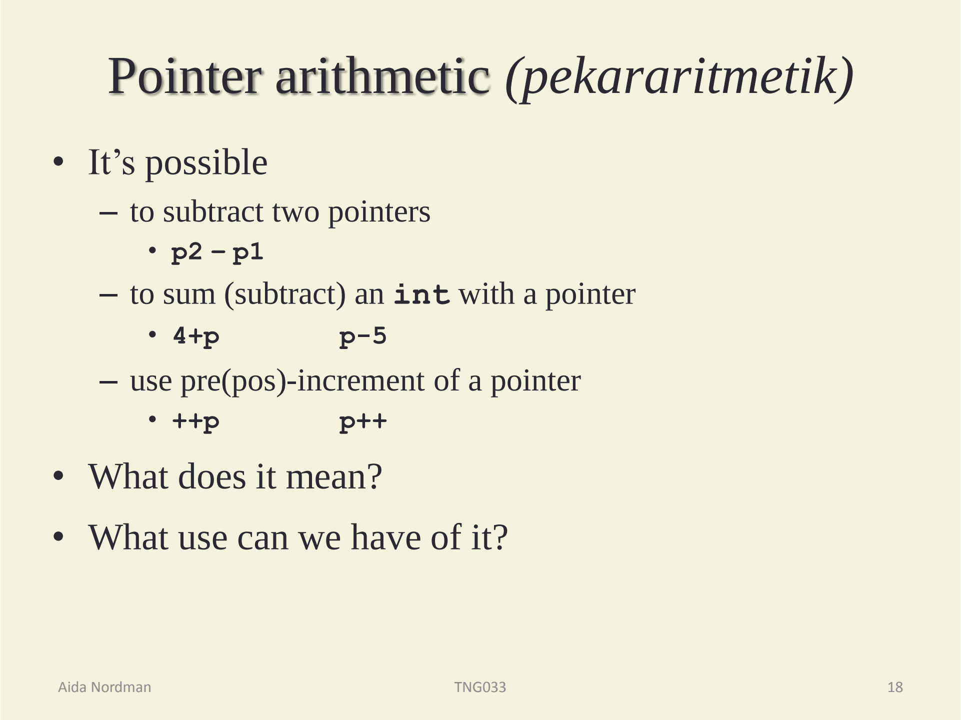
```
const int SIZE = 10;
double V[SIZE] = {0};
double *p = nullptr;

p = &V[0];
p = V; //same as p = &V[0];
```

Name of an array (**V**) *converts automatically* to a **pointer to the first slot** of the array

V	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9

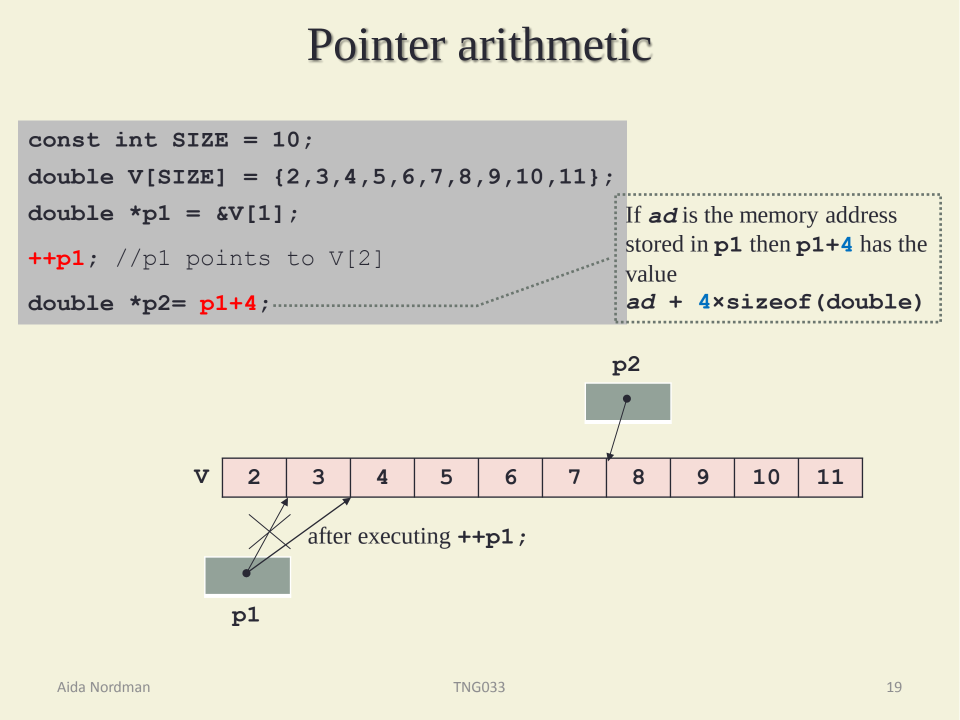
```
cout << *p; //display V[0]
p = &V[3];
cout << *p; //display V[3]
```

Read sec. 5.4.3



Pointer arithmetic *(pekararitmetik)*

- It’s possible
  - to subtract two pointers
    - **p2 – p1**
  - to sum (subtract) an **`int`** with a pointer
    - **4+p   p-5**
  - use pre(pos)-increment of a pointer
    - **++p   p++**
- What does it mean?
- What use can we have of it?

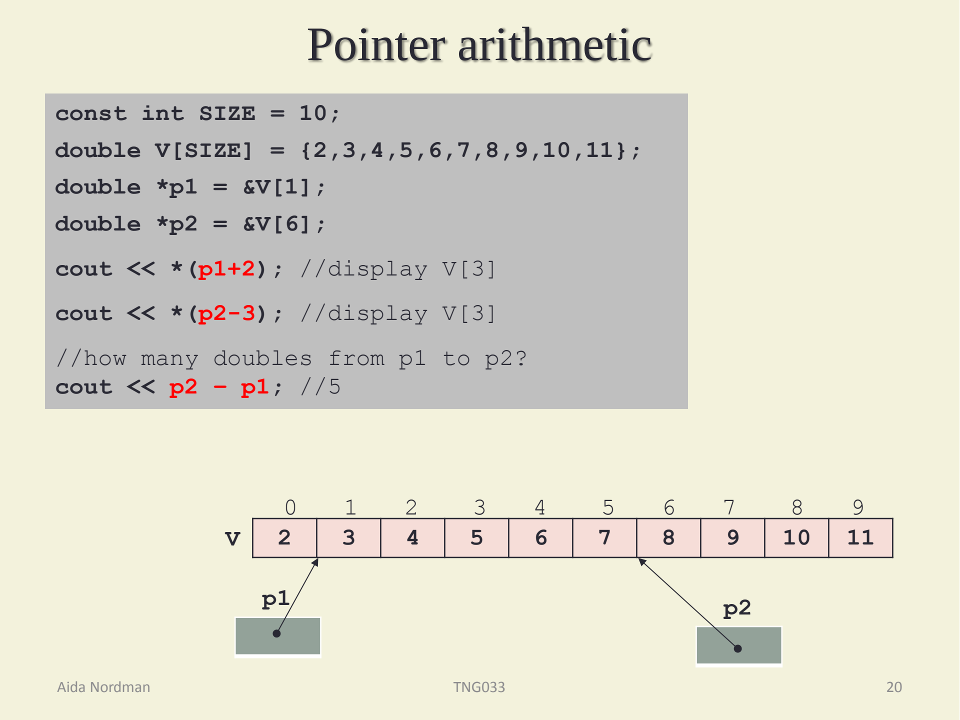


Pointer arithmetic

```
const int SIZE = 10;
double V[SIZE] = {2,3,4,5,6,7,8,9,10,11};
double *p1 = &V[1];
++p1; //p1 points to V[2]
double *p2= p1+4;
```

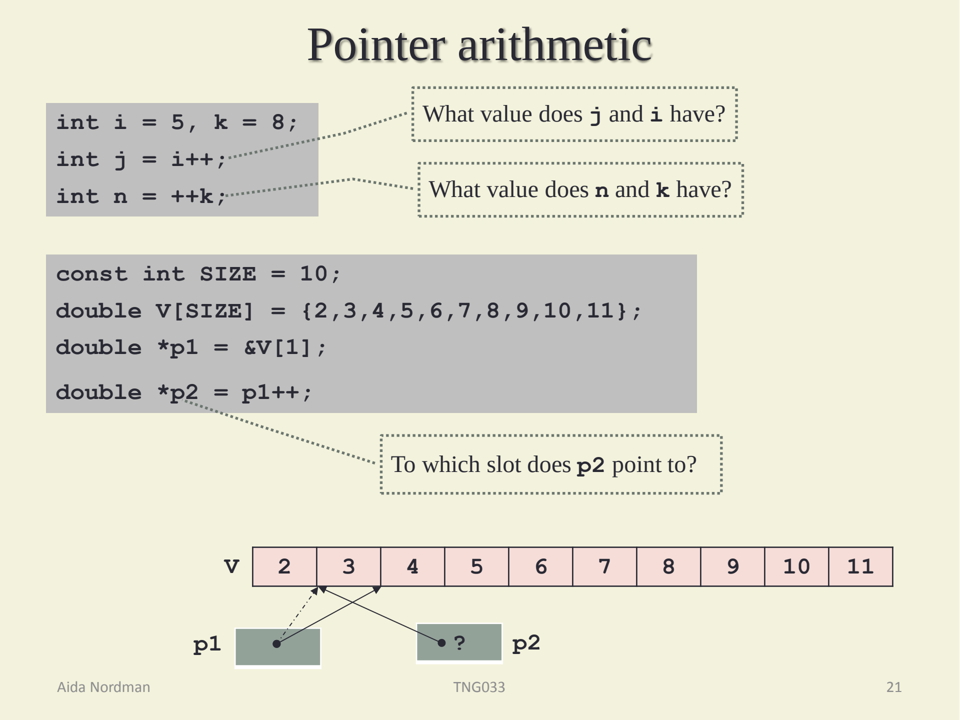
If *ad* is the memory address stored in **p1** then **p1+4** has the value
ad **+ 4×sizeof(double)**

after executing **++p1;**



Pointer arithmetic

```
const int SIZE = 10;
double V[SIZE] = {2,3,4,5,6,7,8,9,10,11};
double *p1 = &V[1];
double *p2 = &V[6];
cout << *(p1+2); //display V[3]
cout << *(p2-3); //display V[3]
//how many doubles from p1 to p2?
cout << p2 – p1; //5
```



Pointer arithmetic

```
int i = 5, k = 8;
int j = i++;
int n = ++k;
```

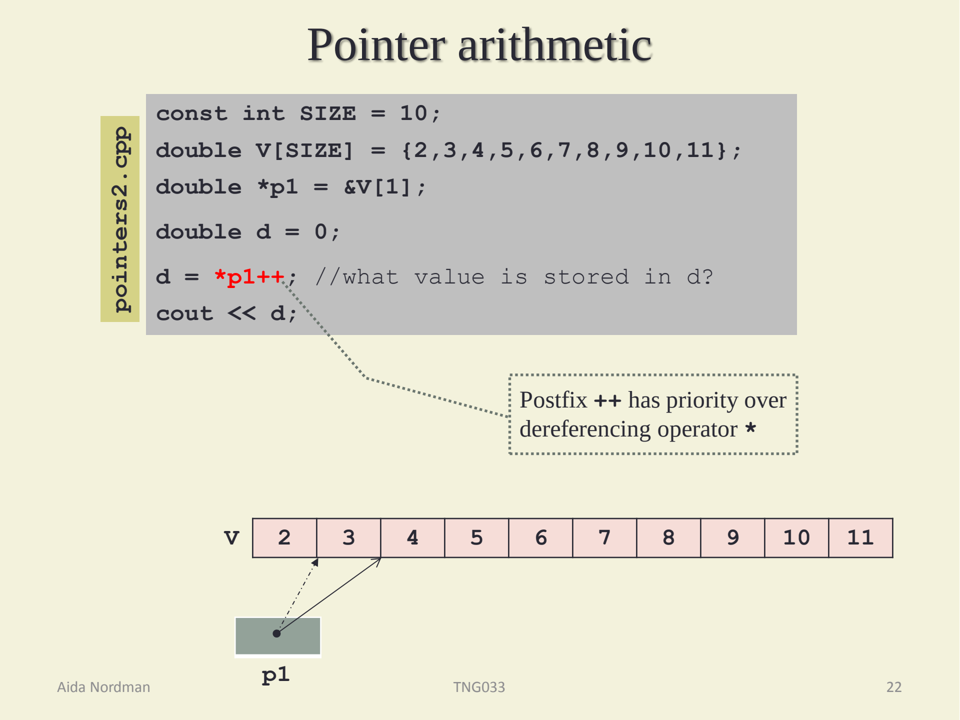
What value does **j** and **i** have?

What value does **n** and **k** have?

```
const int SIZE = 10;
double V[SIZE] = {2,3,4,5,6,7,8,9,10,11};
double *p1 = &V[1];

double *p2 = p1++;
```

To which slot does **p2** point to?

V	2	3	4	5	6	7	8	9	10	11



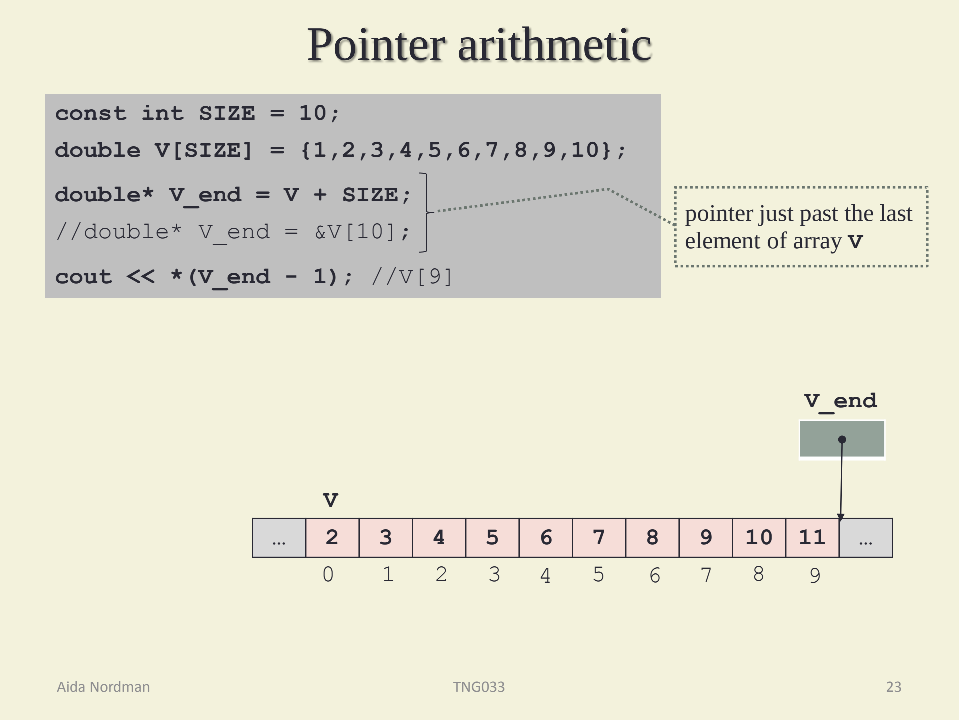
Pointer arithmetic

pointers2.cpp

```
const int SIZE = 10;
double V[SIZE] = {2,3,4,5,6,7,8,9,10,11};
double *p1 = &V[1];

double d = 0;

d = *p1++; //what value is stored in d?
cout << d;
```

Postfix **++** has priority over dereferencing operator *****

V	2	3	4	5	6	7	8	9	10	11



Pointer arithmetic

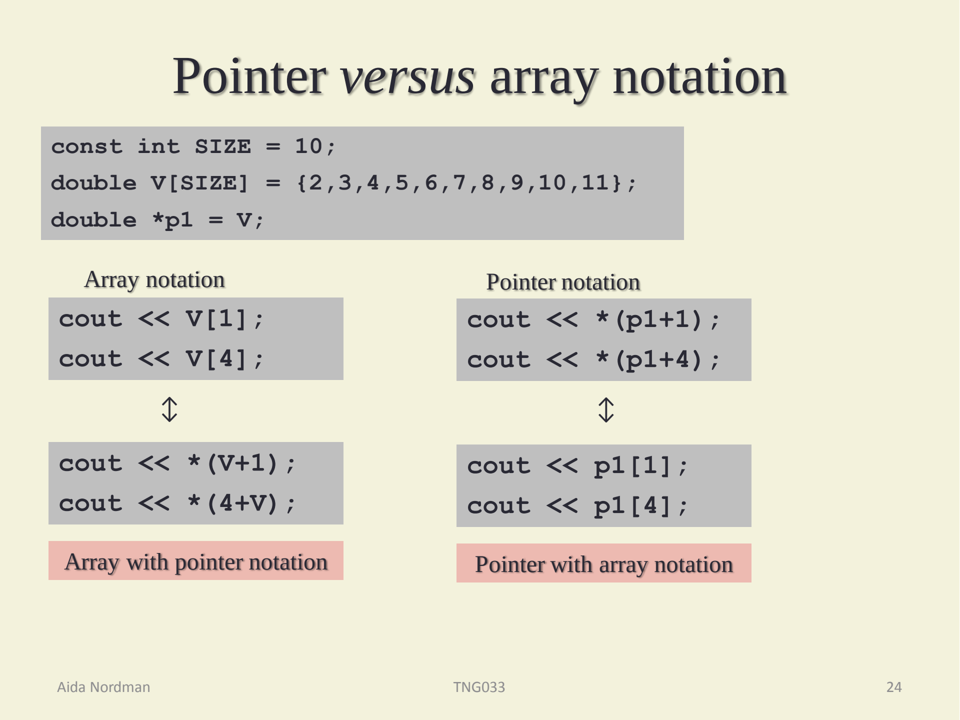
```
const int SIZE = 10;
double V[SIZE] = {1,2,3,4,5,6,7,8,9,10};
double* V_end = V + SIZE;
//double* V_end = &V[10];
cout << *(V_end - 1); //V[9]
```

pointer just past the last element of array **V**

V_end

V

…	2	3	4	5	6	7	8	9	10	11	…
	0	1	2	3	4	5	6	7	8	9	



Pointer *versus* array notation

```
const int SIZE = 10;
double V[SIZE] = {2,3,4,5,6,7,8,9,10,11};
double *p1 = V;
```

Array notation

```
cout << V[1];
cout << V[4];
```

↕

```
cout << *(V+1);
cout << *(4+V);
```

Array with pointer notation

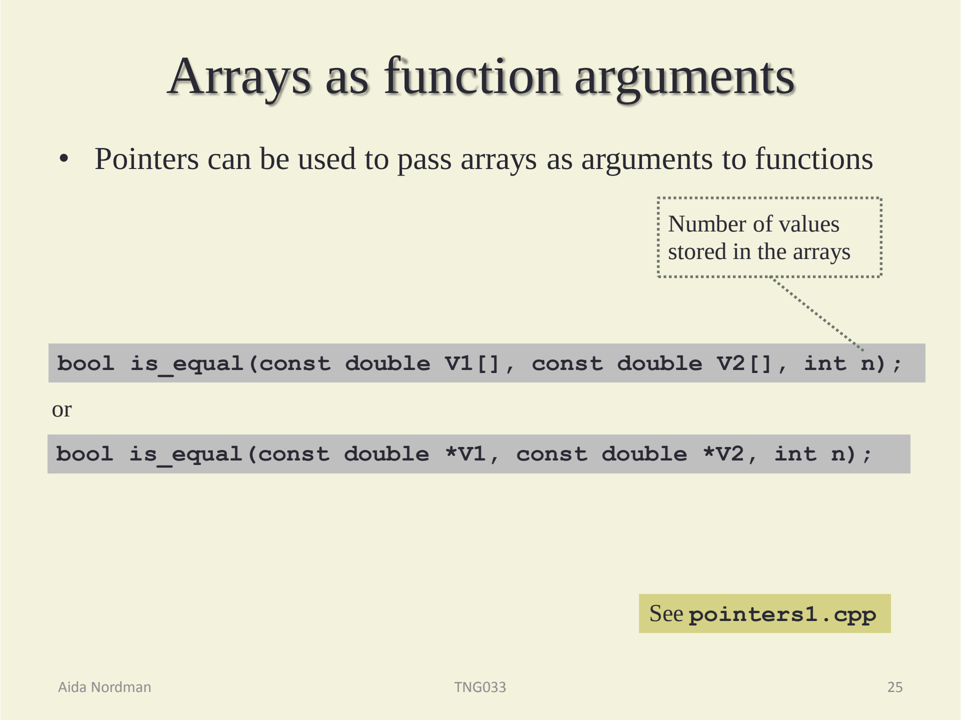
Pointer notation

```
cout << *(p1+1);
cout << *(p1+4);
```

↕

```
cout << p1[1];
cout << p1[4];
```

Pointer with array notation



Arrays as function arguments

- Pointers can be used to pass arrays as arguments to functions

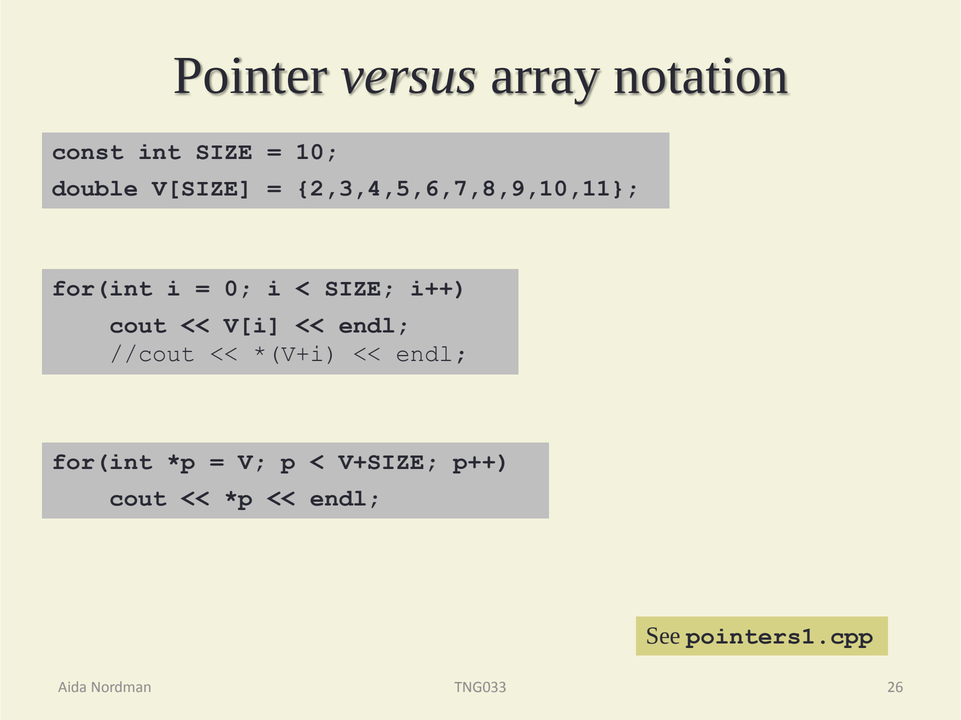
Number of values stored in the arrays

```
bool is_equal(const double V1[], const double V2[], int n);
```

or

```
bool is_equal(const double *V1, const double *V2, int n);
```

See `pointers1.cpp`



Pointer *versus* array notation

```
const int SIZE = 10;
double V[SIZE] = {2,3,4,5,6,7,8,9,10,11};
```

```
for(int i = 0; i < SIZE; i++)
    cout << V[i] << endl;
    //cout << *(V+i) << endl;
```

```
for(int *p = V; p < V+SIZE; p++)
    cout << *p << endl;
```

See `pointers1.cpp`

Exercise

1. Write a function that given a vector $\vec{v} = [v_1, v_2, \ldots, v_n]$, with n>0, calculates the vector length. The length of a vector is given by the formula

$$L = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}$$

   - Vector's coordinates are stored in an array
   - Use pointers
   - Use array with pointer notation
   - The function's declaration is

```
double length(const double *V, int n);
```

2. Write a program that reads a vector's coordinates $[v_1, v_2, \ldots, v_n]$ and then calculates the length of the vector
   - Call the function above

But, why!??

- Pointers are variables that point to other variables
- So,
  1. first, one declares a variabel **k** `int k = 0;`
  2. then, one declares a pointer **p** to **k** `int *p = &k;`
  3. and then, one accesses to **k** through **p** `cout << *p;`
  4. *But, why to access **k** through the pointer if one can access **k**?!*
- Indeed, that's not the reason to use pointers
- Pointers are used to point to **dynamically allocated memory**
  - More the coming lecture …

Next …

- Fö 4
  - Memory allocation/deallocation [5.4.5] *(Minnesallokering)*
  - Common pitfalls [5.4.6]
- Fö 3 is important for understanding the coming Fö 4