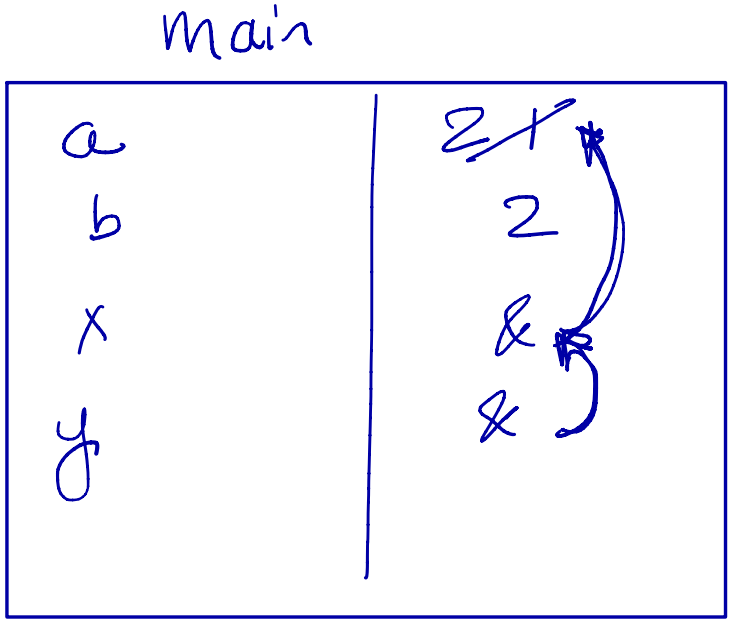
Informatik
Exercise Session

PVK → VMP
→ AMIV
→ VIS

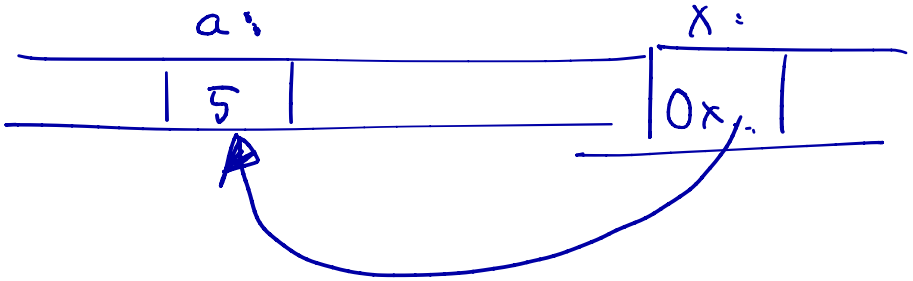
References Recap

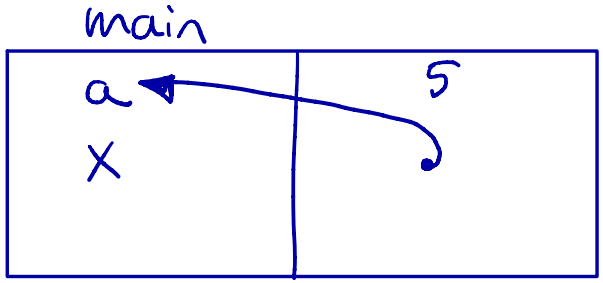



```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << std::endl;
std::cout << x << " " << y << std::endl;
```

Basic Pointers

- Ein Pointer "zeigt" auf eine Adresse. D.h. ein Pointer ist nichts weiteres als eine Variable, deren Wert eine Adresse ist.






```
int a = 5;
int* x = &a;
*x = 6;
```

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)
3. to *take the address* of a variable (address operator) (eg. `int *ptr_a = &a;`)

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)
3. to *take the address* of a variable (address operator) (eg. `int *ptr_a = &a;`)

Similarly, the symbol `*` can be used:

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)
3. to *take the address* of a variable (address operator) (eg. `int *ptr_a = &a;`)

Similarly, the symbol `*` can be used:

1. as the arithmetic multiplication operator (e.g. `z = x * y;`)

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)
3. to *take the address* of a variable (address operator) (eg. `int *ptr_a = &a;`)

Similarly, the symbol `*` can be used:

1. as the arithmetic multiplication operator (e.g. `z = x * y;`)
2. to *declare* a pointer variable (e.g. `int *ptr_a = &a;`)

Meanings of & and *

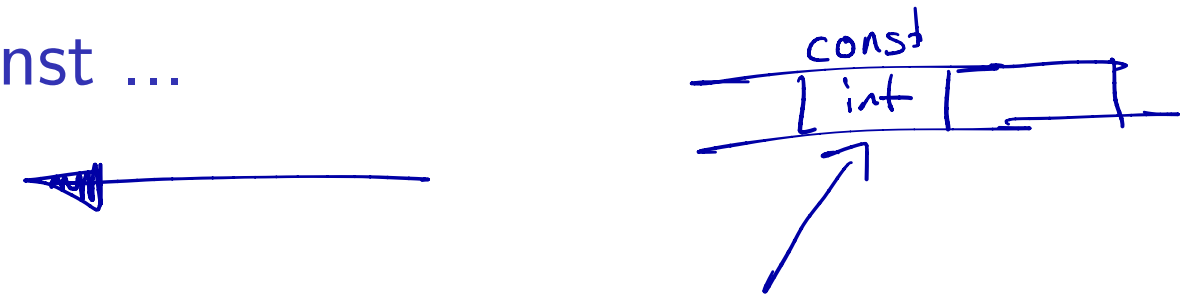
The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)
3. to *take the address* of a variable (address operator) (eg. `int *ptr_a = &a;`)

Similarly, the symbol `*` can be used:

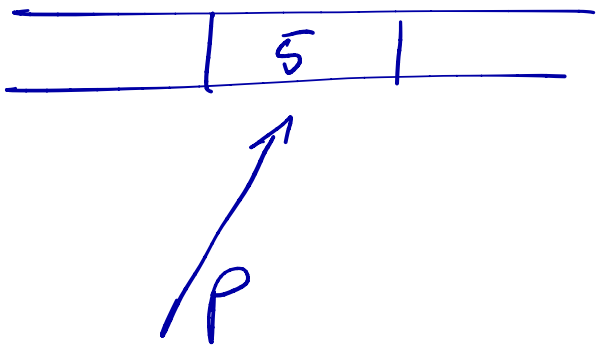
1. as the arithmetic multiplication operator (e.g. `z = x * y;`)
2. to *declare* a pointer variable (e.g. `int *ptr_a = &a;`)
3. to *take the content* of a variable via its pointer (dereference operator) (e.g. `int a = *ptr_a;`)

Noch mehr const ...



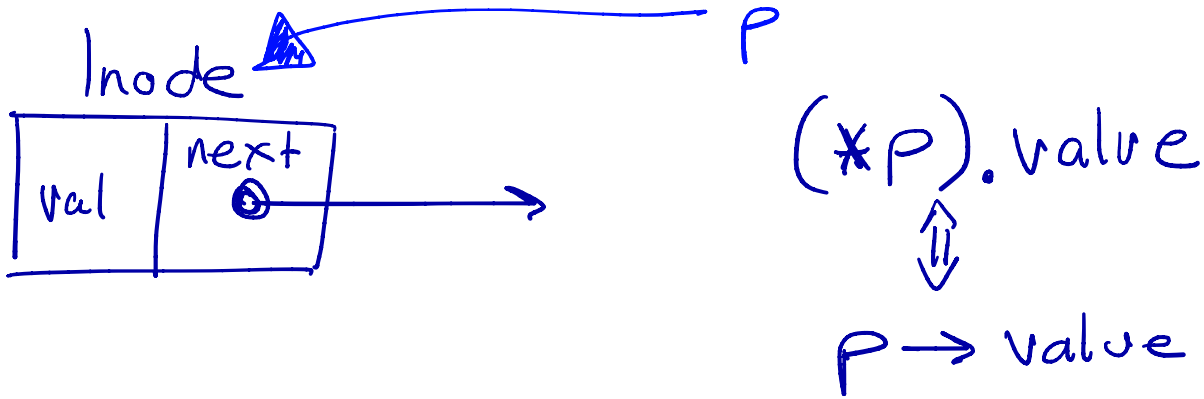

```
int const * p ...
```

```
int * const p ...
```



Pointer Syntax: `(*ptr).member` ist `ptr->member`

- Wir dereferenzieren ptr mittels *-Operator (d.h. wir gehen dorthin, wo ptr hin zeigt) und greifen dort auf member zu mittels .-Operator.



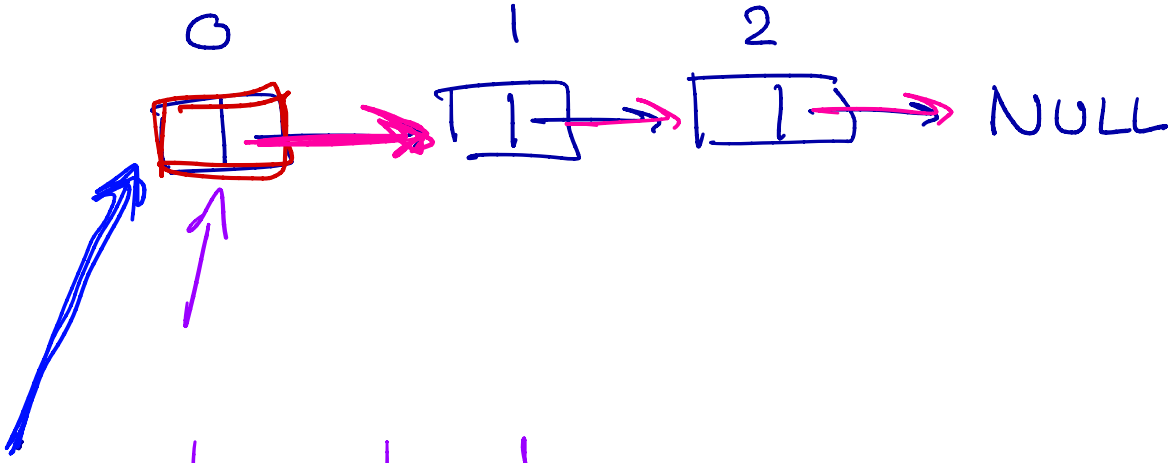

```
struct lnode {
    int value;
    lnode* next;
};

lnode *node_name = some_lnode_ptr

int lnode_value1 = (*node_name).value

int lnode_value2 = node_name->value
```

our_list




```
// in our_list.h
class our_list {

  struct lnode {
    int value;
    lnode* next;
  };

  lnode* head;

public:
    // ...
}
```

it = head

loop: it != NULL

next_it = (*it).next

our_list name = our_list();

// . . .

this pointer

std::vector<int> v ...

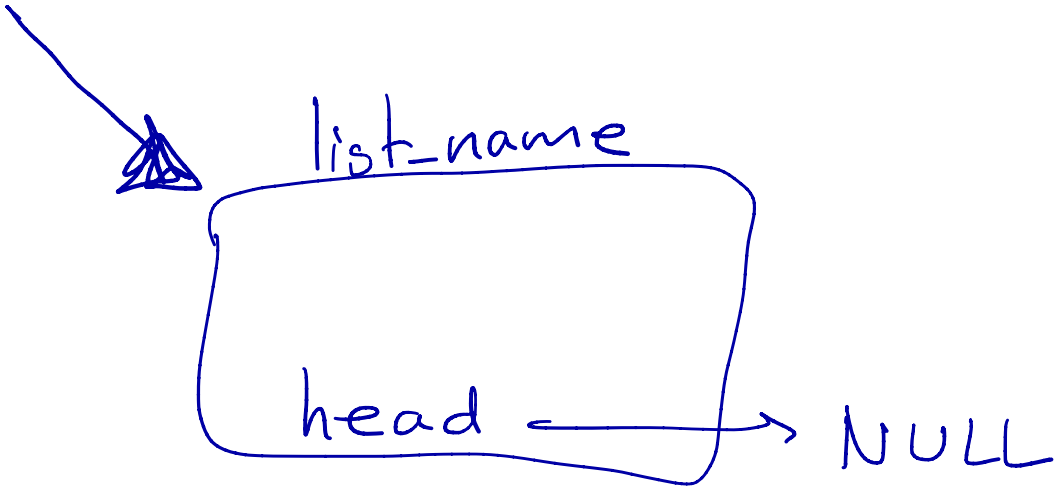
v.size()

```
// in our_list.h
class our_list {
  struct lnode {
    int value;
    lnode* next;
  };
  lnode* head;
public:
    // ...
}

// default constructor in our_list.cpp:
our_list::our_list() {
    // this ptr is a hidden argument in all member functions
    this->head = nullptr;

}

// in main: make a new list called list_name
our_list list_name = our_list();
```



Dynamic Memory

stack vs. heap
↙ ↘
local var etc dynamic mem.

Normalerweise werden all unsere Objekte am Ende ihres scopes "gelöscht", d.h. der Speicher den sie benutzt haben, wird wieder freigegeben.

Wir wollen aber unsere Objekte etwas länger benutzen ...

```
{
  int a = 5;


}
```

Dynamic Memory Allocation: new

The new operator denotes a request for memory allocation. If sufficient memory is available, the new operator default-initializes the memory and returns the address of the newly allocated memory.

- Heisst: `new` gibt uns immer einen Pointer zurück!

Weiterführende Links:

Allg. zu new: `https://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/`

Bzgl. Initialisierung des Speichers: `https://stackoverflow.com/questions/7546620/operator-new-initializes-memory-to-zero`

Dynamic Memory Allocation: new

Syntax allgemein:

```
pointer-variable = new data-type;
```

Syntax Beispiel:

```
int *p = NULL;   // initialize ptr with NULL
p = new int;     // request memory for one int

int *p = new int;   // request directly

// request and initialize with int_value
int *p = new int(int_value);
```

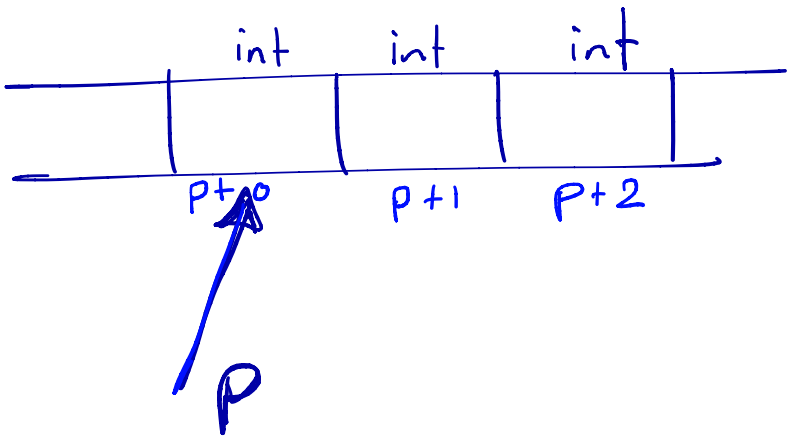
Dynamic Memory Allocation: new

```
// runde Klammern:
// - wir wollen Speicher für einen einzigen Integer
// - p zeigt auf diesen int
int *p = new int(int_value);
```

int(10)

```
// eckige Klammern:
// - wir wollen Speicher für num_ints Integer
// - p zeigt auf den ersten int
int *p = new int[num_ints];
```

int[3]



p[-1] ≙ *(p+1)

(linked list: https://www.geeksforgeeks.org/data-structures/linked-list/)

↳ our_list ist eine singly linked list

our_list.h

```
class our_list {

  struct lnode {
    int value;
    lnode* next;
  };

  lnode* head;

public:
  class const_iterator {
    const lnode* node;

  public:
    const_iterator(const lnode* const n);
    // PRE: Iterator does not point to the element beyond the last one.
    // POST: Iterator points to the next element.
    const_iterator& operator++(); // Pre-increment
    // POST: Return the reference to the number at which the iterator is
    //       currently pointing.
    const int& operator*() const;
    // True if iterators are pointing to different elements.
    bool operator!=(const const_iterator& other) const;
    // True if iterators are pointing to the same element.
    bool operator==(const const_iterator& other) const;
  };

  our_list();
  // PRE: begin and end are iterators pointing to the same vector
  //      and begin is before end.
  // POST: The constructed our_list contains all elements between begin and end.
  our_list(const_iterator begin, const_iterator end);
  // POST: e is appended at the beginning of the vector.
  void push_front(int e);
  // POST: Returns an iterator that points to the first element.
  const_iterator begin() const;
  // POST: Returns an iterator that points after the last element.
  const_iterator end() const;
};

// POST: Outputs the vector into output stream.
std::ostream& operator<<(std::ostream& sink, const our_list& vec);
```

constructor (example init)

```
5  our_list::our_list(our_list::const_iterator begin, our_list::const_iterator end) {
6    this->head = nullptr;
7    if (begin == end) {
8      return;
9    }
10   // Let's add the first element from the iterator.
11   our_list::const_iterator it = begin;
12   this->head = new lnode { *it, nullptr };
13   ++it;
14   lnode *node = this->head;
15   // Let's add all the remaining elements.
16   for (; it != end; ++it) {
17     node->next = new lnode { *it, nullptr };
18     node = node->next;
19   }
20 }
21
22 our_list::our_list() {
23   this->head = nullptr;
24 }
25
26 std::ostream& operator<<(std::ostream& sink, const our_list& vec) {
27   sink << '[';
28   for (our_list::const_iterator it = vec.begin(); it != vec.end(); ++it) {
29     sink << *it << ' ';
30   }
31   sink << ']';
32   return sink;
33 }
34
```

```
35  void our_list::push_front(int e) {
36    this->head = new lnode { e, this->head };
37  }
38
39  our_list::const_iterator our_list::begin() const {
40    return our_list::const_iterator(this->head);
41  }
42
43  our_list::const_iterator our_list::end() const {
44    return our_list::const_iterator(nullptr);
45  }
46
47
48  our_list::const_iterator::const_iterator(const lnode* const n): node(n) {}
49
50  our_list::const_iterator& our_list::const_iterator::operator++() {
51    assert(this->node != nullptr);
52
53    this->node = this->node->next;
54
55    return *this;
56  }
57
58  const int& our_list::const_iterator::operator*() const {
59    return this->node->value;
60  }
61
62  bool our_list::const_iterator::operator!=(const our_list::const_iterator& other) const {
63    return this->node != other.node;
64  }
65
66  bool our_list::const_iterator::operator==(const our_list::const_iterator& other) const {
67    return this->node == other.node;
68  }
```

Swap

```
5  void our_list::swap(unsigned int index) {
6    if (index == 0) {
7      assert(this->head != nullptr);
8      assert(this->head->next != nullptr);
9      lnode* tmp = this->head->next;
10     this->head->next = this->head->next->next;
11     tmp->next = this->head;
12     this->head = tmp;
13   } else {
14     lnode* prev = nullptr;
15     lnode* curr = this->head;
16
17     // Find the element.
18     while (index > 0) {
19       prev = curr;
20       curr = curr->next;
21       --index;
22     }
23
24     assert(curr != nullptr);
25     assert(curr->next != nullptr);
26
27     // Swap with the next one.
28     lnode* tmp = curr->next;
29     curr->next = curr->next->next;
30     tmp->next = curr;
31     prev->next = tmp;
32   }
33 }
```