17. Classes

Classes, Member Functions, Constructors, Stack, Linked List, Dynamic Memory, Copy-Constructor, Assignment Operator, Concept Dynamic Datatype

Encapsulation: public / private

```
class rational {
  int n;
  int d; // INV: d != 0
};
```

Good news: `r.d = 0` cannot happen any more by accident.

Bad news: the customer cannot do anything any more ...

Application Code

... and we can't, either. (no `operator+`, ...)

```
rational r;
r.n = 1;      // error: n is private
r.d = 2;      // error: d is private
int i = r.n; // error: n is private
```


Member Functions: Declaration

```
class rational {
public:
  // POST: return value is the numerator of *this
  int numerator () const {
    return n;
  }
  // POST: return value is the denominator of *this
  int denominator () const {
    return d;
  }
private:
  int n;
  int d; // INV: d!= 0
};
```

public area

member function

member functions have access to private data

the scope of members in a class is the whole class, independent of the declaration order

Member Functions: Call

```
// Definition des Typs
class rational {
    ...
};
...
// Variable des Typs
rational r;

int n = r.numerator();    // Zaehler
int d = r.denominator(); // Nenner
```

member access

Member Functions: Definition

```
// POST: returns  numerator of *this
int numerator ()  const
{
  return n;
}
```

- A member function is called for an expression of the class. in the function, `*this` is the name of this implicit argument. `this` itself is a pointer to it.
- const refers to `*this`, i.e., it promises that the value associated with the implicit argument cannot be changed
- `n` is the shortcut in the member function for `(*this).n`

Comparison

It would look like this...

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return (*this).n;
    }
};

rational r;
...
std::cout << r.numerator();
```

... without member functions

```
struct bruch {
    int n;
    ...
};

int numerator (const bruch∗ dieser)
{
    return (∗dieser).n;
}

bruch r;
..
std::cout << numerator(&r);
```


Member-Definition: In-Class vs. Out-of-Class

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return n;
    }
    ....
};
```

- No separation between declaration and definition (bad for libraries)

```
class rational {
    int n;
    ...

    int numerator () const;
    ...
};

int rational::numerator () const
{
  return n;
}
```

- This also works.

Constructors

- are special member functions of a class that are named like the class
- can be overloaded like functions, i.e. can occur multiple times with varying *signature*
- are called like a function when a variable is declared. The compiler chooses the “closest” matching function.
- if there is no matching constructor, the compiler emits an *error message.*

Initialisation? Constructors!

```
class rational
{
public:
    rational (int num, int den)
        : n (num), d (den)   ⟵ Initialization of the member variables
    {
        assert (den != 0);   ⟵ function body.
    }
...
};
...
rational r (2,3); // r = 2/3
```


Constructors: Call

- directly

```
rational r (1,2); // initialisiert r mit 1/2
```

- indirectly (copy)

```
rational r = rational (1,2);
```


Initialisation “rational = int”?

```
class rational
{
public:
    rational (int num)
        : n (num), d (1)
    {}   ⟵ empty function body
...
};
...
rational r (2);    // explicit initialization with 2
rational s = 2;    // implicit conversion
```


User Defined Conversions

are defined via constructors with exactly *one* argument

```
rational (int num)   ⟵
    : n (num), d (1)
  {}

rational r = 2; // implizite Konversion
```

User defined conversion from `int` to `rational`. values of type `int` can now be converted to `rational`.

The Default Constructor

```
class rational
{
public:
    ...
    rational ()      ← empty list of arguments
      : n (0), d (1)
    {}
 ...
};
 ...
rational r;     // r = 0
```

⇒ There are no uninitiatlized variables of type rational any more!

The Default Constructor

- is automatically called for declarations of the form
  `rational r;`
- is the unique constructor with empty argmument list (if existing)
- must exist, if `rational r;` is meant to compile
- if in a struct there are no constructors at all, the default constructor is automatically generated

RAT PACK® Reloaded ...

Customer's program now looks like this:

```
// POST: double approximation of r
double to_double (const rational r)
{
  double result = r.numerator();
  return result / r.denominator();
}
```

- We can adapt the member functions together with the representation ✓

RAT PACK® Reloaded ...

before

```
class rational {
...
private:
  int n;
  int d;
};
```

```
int numerator () const
{
  return n;
}
```

after

```
class rational {
...
private:
  unsigned int n;
  unsigned int d;
  bool is_positive;
};
```

```
int numerator () const{
  if (is_positive)
    return n;
  else {
    int result = n;
    return −result;
  }
}
```

RAT PACK® Reloaded ?

```
class rational {
...
private:
  unsigned int n;
  unsigned int d;
  bool is_positive;
};
```

```
int numerator () const
{
  if (is_positive)
    return n;
  else {
    int result = n;
    return −result;
  }
}
```

- value range of nominator and denominator like before
- possible overflow in addition

Encapsulation still Incompleete

Customer's point of view (`rational.h`):

```
class rational {
public:
   // POST: returns numerator of *this
   int numerator () const;
   ...
private:
  // none of my business
};
```

- We determined denominator and nominator type to be `int`
- Solution: encapsulate not only data but alsoe types.

Fix: "our" type `rational::integer`

Customer's point of view (`rational.h`):

```
public:
   typedef int integer; // might change
   // POST: returns numerator of *this
   integer numerator () const;
```

- We provide an additional type!
- Determine only Functionality, e.g:
  - implicit conversion `int` → `rational::integer`
  - function `double to_double (rational::integer)`

RAT PACK® Revolutions

Finally, a customer program that remains stable

```
// POST: double approximation of r
double to_double (const rational r)
{
  rational::integer n = r.numerator();
  rational::integer d = r.denominator();
  return to_double (n) / to_double (d);
}
```

Separate Declaration and Definition

```
class rational {
public:
   rational (int num, int denum);
   typedef int integer;
   integer numerator () const;
   ...
private:
 ...
};
```

rational.h

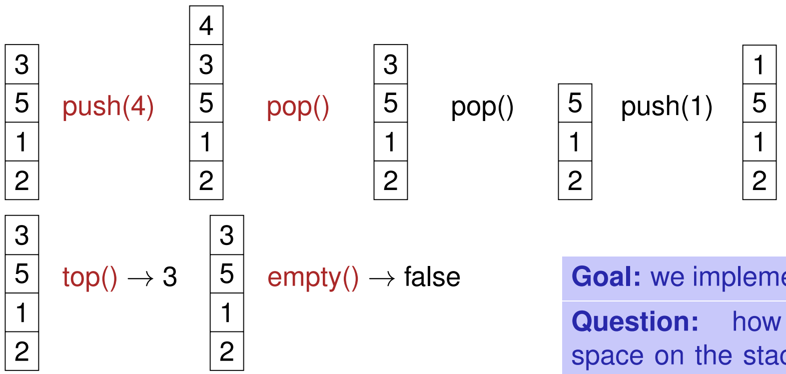
```
rational::rational (int num, int den):
    n (num), d (den) {}
rational::integer rational::numerator () const
{
    return n;
}
```

rational.cpp

class name :: member name

Motivation: Stack

Motivation: Stack (`push`, `pop`, `top`, `empty`)



Goal: we implement a stack class

Question: how do we create space on the stack when push is called?

We Need a new Kind of Container

Our main container: Array (`T[]`)

- Contiguous area of memory, random access (to ith element)
- Simulation of a stack with an array?
- No, at some point the array will become "full".

not possible to execute `push(3)` here!

Arrays are no all-rounders…

- It is expensive to insert or delete elements “in the middle ”.

1	5	6	3	8	9	3	3	8	9

↑
8

If we want to insert, we have to move everything to the right (if there is space at all!)

Arrays are no all-rounders…

- It is expensive to insert or delete elements “in the middle ”.

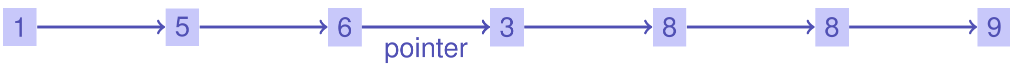
1	5	6	3	8	8	9	3	3	8	9

↑

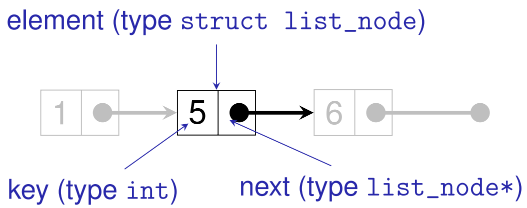
If we want to remove this element, we have to move everything to the right of it.

The new Container: Linked List

- *No* contiguous area of memory and *no* random access
- Each element “knows” its successor
- Insertion and deletion of arbitrary elements is simple, *even at the beginning of the list*
- ⇒ A stack can be implemented as linked list

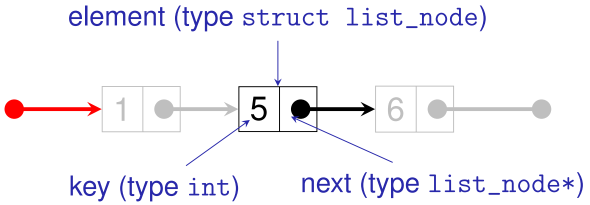


Linked List: Zoom




```
struct list_node {
  int           key;
  list_node*    next;
  // constructor
  list_node (int k, list_node* n)
    : key (k), next (n) {}
};
```

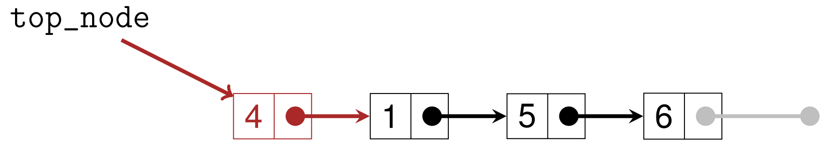
Stack = Pointer to the Top Element




```
class stack {
public:
   void push (int value) {...}
   ...
private:
   list_node* top_node;
};
```


Sneak Preview: `push(4)`

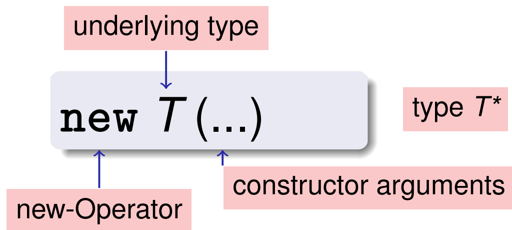
```
void push (int value)
{
  top_node = new list_node (value, top_node);
}
```



Dynamic Memory

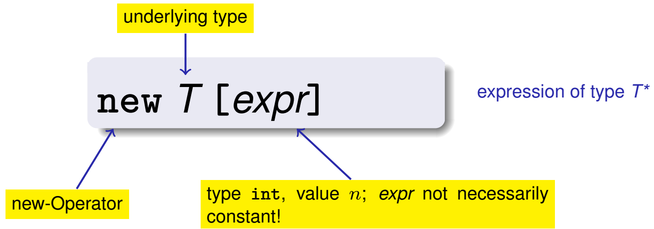
- For dynamic data structures like lists we need *dynamic memory*
- Up to now we had to fix the memory sizes of variable at *compile time*
- Pointers allow to request memory at *runtime*
- Dynamic memory management in C++ with operators `new` and `delete`

The `new` Expression



- **Effect:** new object of type *T* is allocated in memory ...
- ... and initialized by means of the matching constructor.
- **Value:** address of the new object

`new` for Arrays

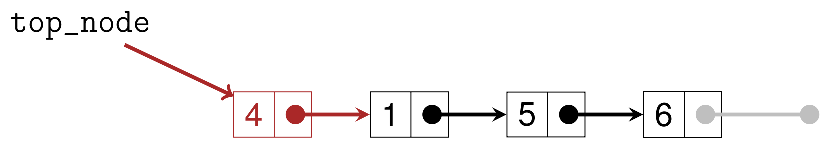


- memory for an array with length n and underlying type *T* is allocated
- Value of the expression is the address of the first element of the array

The `new` Expression push(4)

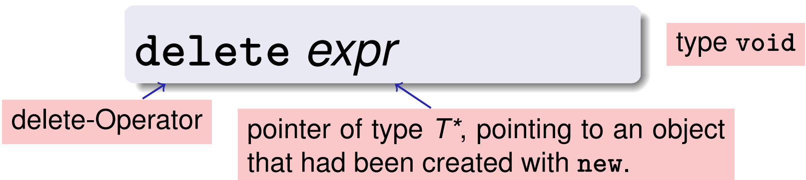
- **Effect:** new object of type *T* is allocated in memory …
- … and intialized by means of the matching constructor
- **Value:** address of the new object

```
top_node = new list_node (value, top_node);
```



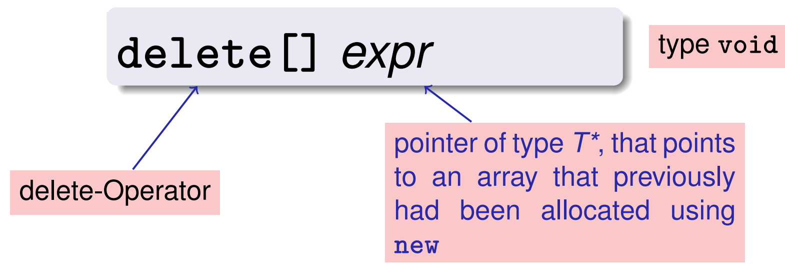
The `delete` Expression

Objects generated with `new` have *dynamic storage duration:* they "live" until they are explicitly *deleted*



- **Effect:** object is deleted and memory is released

`delete` for Arrays



- **Effect:** array is deleted and memory is released

Carefult with `new` and `delete`!

```
rational* t = new rational;       ← memory for t is allocated
rational* s = t;                  ← other pointers may also point to the same object
delete s;                         ← ... and used for releaseing the object
int nominator = (*t).denominator(); ← error: memory already released!
                 ↑
      Dereferencing of „dangling pointers”
```

- Pointer to released objects: *dangling pointers*
- Releasing an object more than once using `delete` is a similar severe error
- `delete` can be easily forgotten: consequence are *memory leaks*. Can lead to memory overflow in the long run.

Who is born must die. . .

Guideline “Dynamic Memory”
For each `new` there is a matching `delete`!

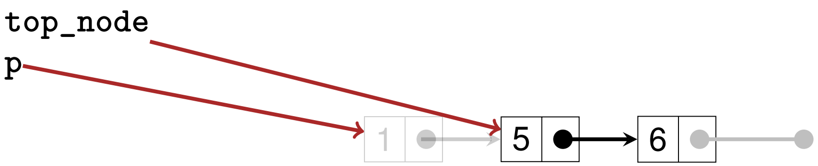
Non-compliance leads to memory leaks

- old objects that occupy memory. . .
- . . . until it is full (heap overflow)

Stack Continued: `pop()`

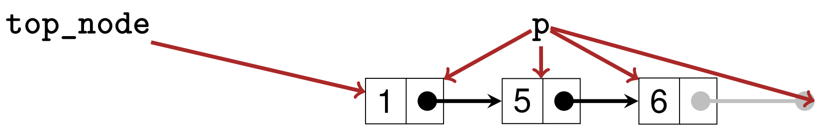
```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

↑ shortcut for `(*top_node).next`



Traverse the Stack `print()`

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " "; // 1 5 6
    p = p->next;
  }
}
```



Output Stack: `operator<<`

```
class stack {
public:
   void push (int value) {...}
   ...
   void print (std::ostream& o) const {...}
private:
   list_node* top_node;
};

// POST: s is written to o
std::ostream& operator<< (std::ostream& o, const stack& s)
{
   s.print (o);
   return o;
}
```


Empty Stack , `empty()`, `top()`

```
stack()      // default constructor
    : top_node (0)
{}

bool empty () const
{
  return top_node == 0;
}

int top () const
{
  assert (!empty());
  return top_node->key;
}
```


Stack Done? Obviously not. . .

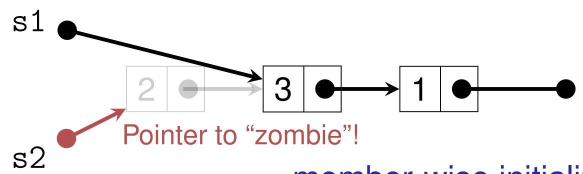
```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1

stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop (); // Oops, crash!
```


What has gone wrong?



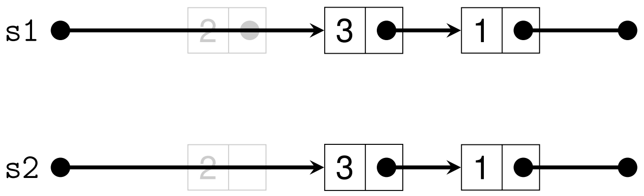
member-wise initialization: copies the `top_node` pointer only.

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();  // Oops, crash!
```

We need a real copy




```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();  // ok
```


The Copy Constructor

- The copy constructor of a class *T* is the unique constructor with declaration

  T (`const` *T*& x);

- is automatically called when values of type *T* are initialized with values of type `T`

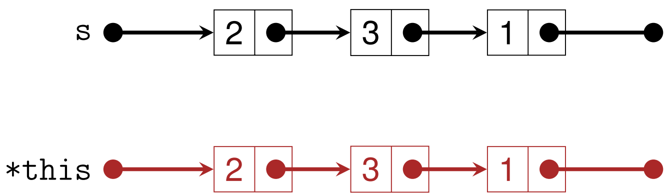
  T `x = t;` (t of type T)
  T `x (t);`

- If there is no copy-constructor declared then it is generated automatically (and initializes member-wise – reason for the problem above

It works with a Copy Constructor

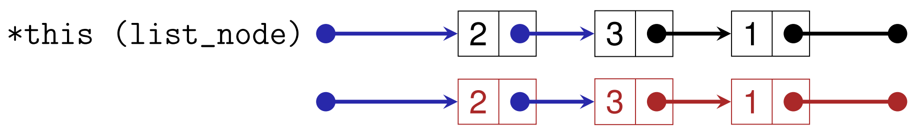
We use a copy function of the `list_node`:

```
// POST: *this is initialized with a copy of s
stack (const stack& s)
  : top_node (0)
{
  if (s.top_node != 0)
    top_node = s.top_node->copy();
}
```



The (Recursive) Copy Function of `list_node`

```
// POST: pointer to a copy of the list starting
//       at *this is returned
list_node* copy () const
{
  if (next != 0)
    return new list_node (key, next->copy());
  else
    return new list_node (key, 0);
}
```



Initialization ≠ Assignment!

```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1

stack s2;
s2 = s1; // Zuweisung

s1.pop ();
std::cout << s1 << "\n"; // 3 1
s2.pop (); // Oops, Crash!
```


The Assignment Operator

- Overloading `operator=` as a member function
- Like the copy-constructor without initializer, but additionally
  - Releasing memory for the “old” value
  - Check for self-assignment (`s1=s1`) that should not have an effect
- If there is no assignment operator declared it is automatically generated (and assigns member-wise – reason for the problem above

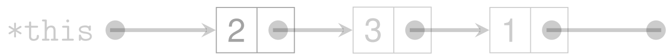
It works with an Assignment Operator!

Here a release function of the `list_node` is used:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
  if (top_node != s.top_node) { // keine Selbtszuweisung!
    if (top_node != 0) {
      top_node->clear(); // loesche Knoten in *this
      top_node = 0;
  }
    if (s.top_node != 0)
      top_node = s.top_node->copy(); // kopiere s nach *this
    }
  return *this; // Rueckgabe als L−Wert (Konvention)
}
```


The (recursive) release function of `list_node`

```
// POST: the list starting at *this is deleted
void clear ()
{
  if (next != 0)
    next->clear();
  delete this;
}
```



Zombie Elements

```
{
  stack s1; // local variable
  s1.push (1);
  s1.push (3);
  s1.push (2);
  std::cout << s1 << "\n"; // 2 3 1
}
// s1 has died (become invalid)...
```

- ...but the three elements of the stack `s1` continue to live (memory leak)!
- They should be released together with `s1`.

The Destructor

- The Destructor of class *T* is the unique member function with declaration

  $\sim T\,(\,);$

- is automatically called when the memory duration of a class object ends
- If no destructor is declared, it is automatically generated and calls the destructors for the member variables (pointers `top_node`, no effect – reason for zombie elements

Using a Destructor, it Works

```
// POST: the dynamic memory of *this is deleted
~stack()
{
  if (top_node != 0)
    top_node->clear();
}
```

- automatically deletes all stack elements when the stack is being released
- Now our stack class follows the guideline “dynamic memory”

Dynamic Datatype

- Type that manages dynamic memory (e.g. our class for a stack)
- Other Applications:
  - Lists (with insertion and deletion “in the middle”)
  - Trees (next week)
  - waiting queues
  - graphs
- Minimal Functionality:
  - Constructors
  - Destructor
  - Copy Constructor
  - Assignment Operator

  (Destructor, Copy Constructor, Assignment Operator:) Rule of Three: if a class defines at least one of them, it must define all three