

21. Dynamic Datatypes and Memory Management

Problem

Last week: dynamic data type

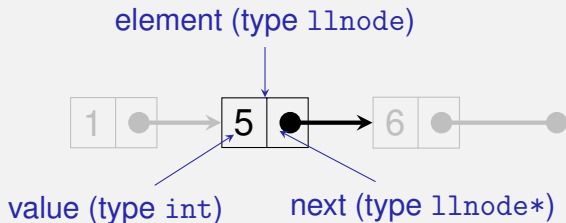
Have allocated dynamic memory, but not released it again. In particular: no functions to remove elements from `llvec`.

Today: correct memory management!

Goal: class stack with memory management

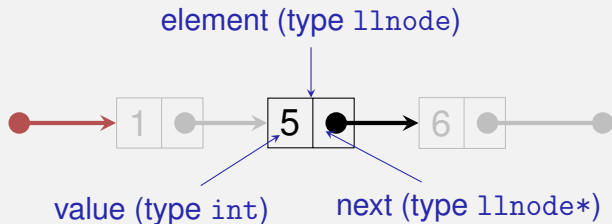
```
class stack{
public:
    // post: Push an element onto the stack
    void push(int value);
    // pre: non-empty stack
    // post: Delete top most element from the stack
    void pop();
    // pre: non-empty stack
    // post: return value of top most element
    int top() const;
    // post: return if stack is empty
    bool empty() const;
    // post: print out the stack
    void print(std::ostream& out) const;
    ...
};
```

Recall the Linked List



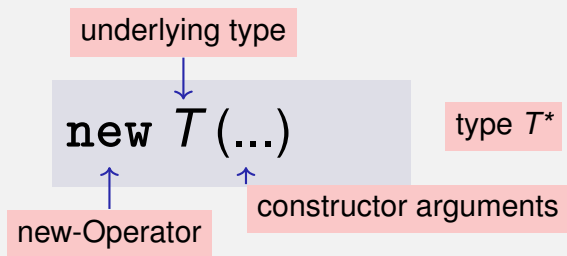
```
struct llnode {  
    int value;  
    llnode* next;  
    // constructor  
    llnode (int v, llnode* n) : value (v), next (n) {}  
};
```

Stack = Pointer to the Top Element



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

Recall 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

The new Expression

push(4)

```
void stack::push(int value){  
    topn = new llnode (value, topn);  
}
```

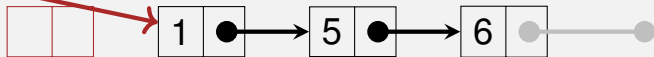
topn



- **Effect:** new object of type T is allocated in memory ...

```
void stack::push(int value){  
    topn = new llnode (value, topn);  
}
```

topn



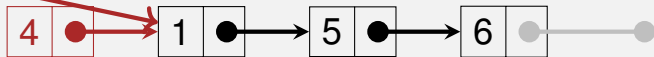
The new Expression

push(4)

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

```
void stack::push(int value){  
    topn = new llnode (value, topn);  
}
```

topn



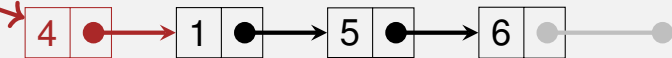
The new Expression

push(4)

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

```
void stack::push(int value){  
    topn = new llnode (value, topn);  
}
```

topn

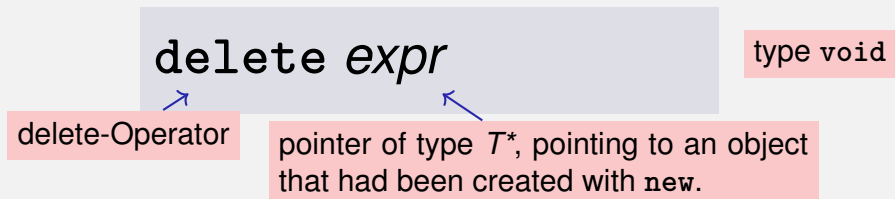


The delete Expression

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

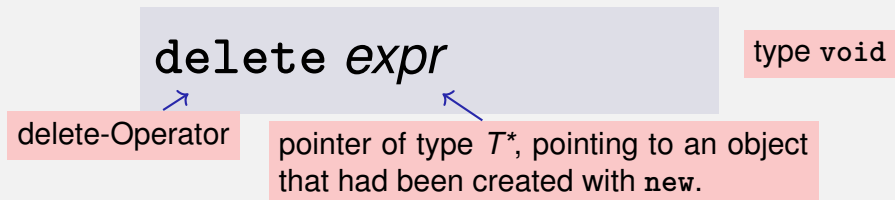
The delete Expression

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



The delete Expression

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



- **Effect:** object is *deconstructed* (explanation below)
... and *memory is released*.

Who is born must die...

Guideline “Dynamic Memory”

For each `new` there is a matching `delete`!

Who is born must die...

Guideline “Dynamic Memory”

For each `new` there is a matching `delete`!

Non-compliance leads to memory leaks

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...

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**)

Careful with new and delete!

```
rational* t = new rational;  
rational* s = t;  
delete s;  
int nominator = (*t).denominator();
```

Careful with new and delete!

```
rational* t = new rational; ←———— memory for t is allocated  
rational* s = t;  
delete s;  
int nominator = (*t).denominator();
```

Careful with new and delete!

```
rational* t = new rational; ←———— memory for t is allocated  
rational* s = t; ←———— other pointers may point to the same object  
delete s;  
int nominator = (*t).denominator();
```

Careful with `new` and `delete`!

```
rational* t = new rational; ←———— memory for t is allocated  
rational* s = t; ←———— other pointers may point to the same object  
delete s; ←———— ... and used for releasing the object  
int nominator = (*t).denominator();
```

Careful with `new` and `delete`!

```
rational* t = new rational; ← memory for t is allocated
rational* s = t; ← other pointers may point to the same object
delete s; ← ... and used for releasing the object
int nominator = (*t).denominator(); ← error: memory released!
```

↑
Dereferencing of „dangling pointers”

- Pointer to released objects: *dangling pointers*

Careful with `new` and `delete`!

```
rational* t = new rational; ← memory for t is allocated
rational* s = t; ← other pointers may point to the same object
delete s; ← ... and used for releasing the object
int nominator = (*t).denominator(); ← error: memory released!
```

↑
Dereferencing of „dangling pointers”

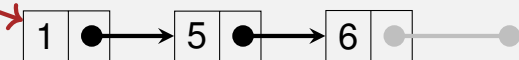
- Pointer to released objects: *dangling pointers*
- Releasing an object more than once using `delete` is a similar severe error

Stack Continued:

pop()

```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```

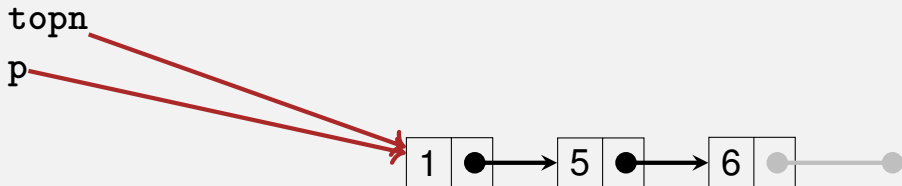
topn



Stack Continued:

pop()

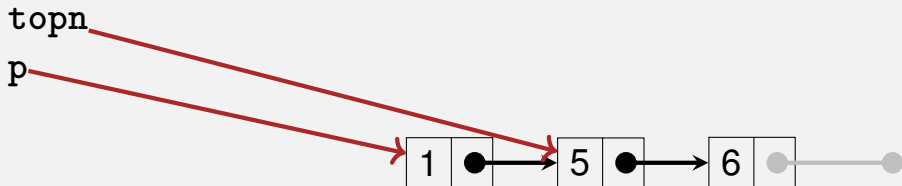
```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```



Stack Continued:

pop()

```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```

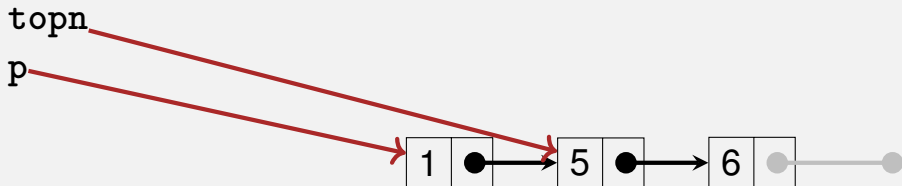


Stack Continued:

pop()

```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```

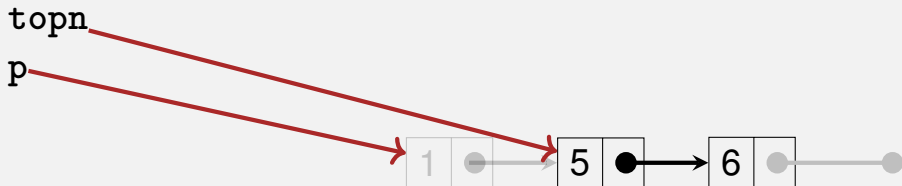
reminder: shortcut for (*topn).next



Stack Continued:

pop()

```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```

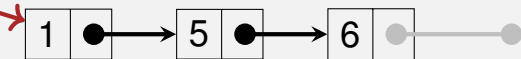


Print the Stack

print()

```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " ";  
}
```

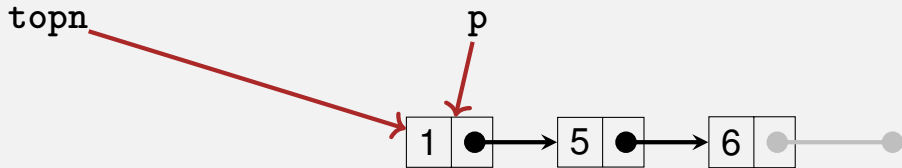
topn



Print the Stack

print()

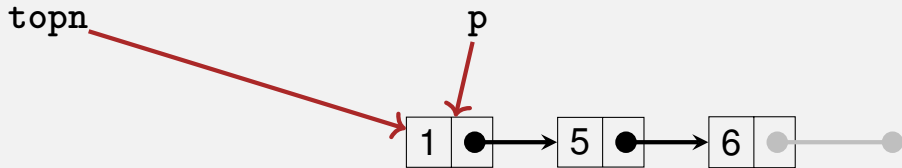
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " ";  
}
```



Print the Stack

print()

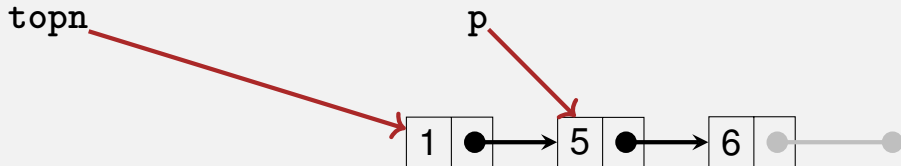
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1  
}
```



Print the Stack

print()

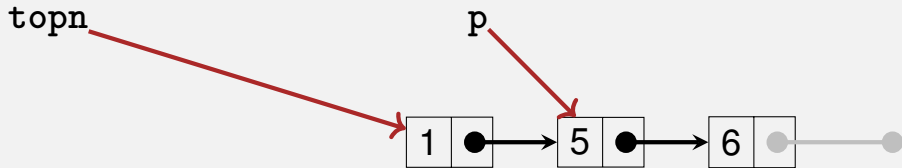
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1  
}
```



Print the Stack

print()

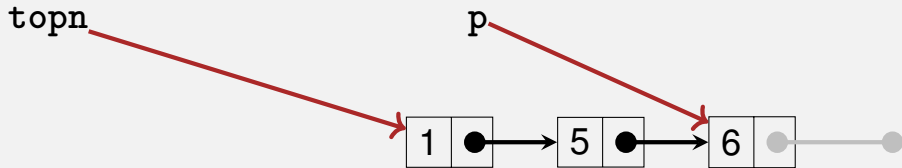
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1 5  
}
```



Print the Stack

print()

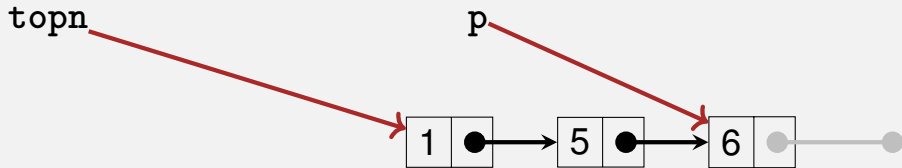
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1 5  
}
```



Print the Stack

print()

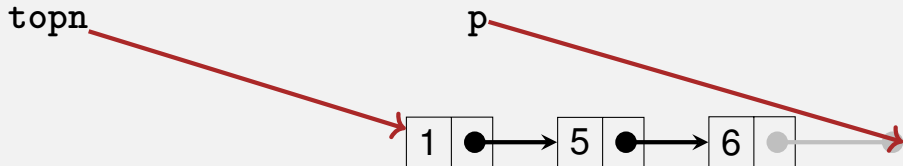
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1 5 6  
}
```



Print the Stack

print()

```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1 5 6  
}
```



Output Stack:

operator<<

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

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

empty(), top()

```
bool stack::empty() const {  
    return top == nullptr;  
}
```

```
int stack::top() const {  
    assert(!empty());  
    return topn->value;  
}
```

Empty Stack

```
class stack{
public:
    stack() : topn (nullptr) {} // default constructor

    void push(int value);
    void pop();
    void print(std::ostream& out) const;
    int top() const;
    bool empty() const;
private:
    llnode* topn;
}
```

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)...
```


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)!

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 – i.e. when `delete` is called on an object of type T^* or when the enclosing scope of an object of type T ends.
- If no destructor is declared, it is automatically generated and calls the destructors for the member variables (pointers `topn`, no effect – reason for zombie elements)

Using a Destructor, it Works

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
    while (topn != nullptr){
        llnode* t = topn;
        topn = t->next;
        delete t;
    }
}
```

Using a Destructor, it Works

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
    while (topn != nullptr){
        llnode* t = topn;
        topn = t->next;
        delete t;
    }
}
```

- automatically deletes all stack elements when the stack is being released

Using a Destructor, it Works

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
    while (topn != nullptr){
        llnode* t = topn;
        topn = t->next;
        delete t;
    }
}
```

- automatically deletes all stack elements when the stack is being released
- Now our stack class seems to follow the guideline “dynamic memory” (?)

Stack Done?

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

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stack Done?

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

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```


Stack Done?

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

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stack Done?

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

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stack Done?

```
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";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stack Done?

```
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";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stack Done?

```
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";
```

```
s2.pop ();
```

Stack Done?

```
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";
```

```
s2.pop ();
```

Stack Done?

```
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 ();
```

Stack Done?

```
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 ();
```


Stack Done?

```
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!
```

Stack Done?

Obviously not...

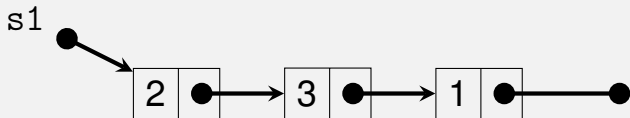
```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
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?



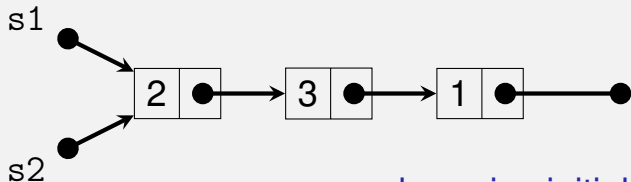
...

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

What has gone wrong?



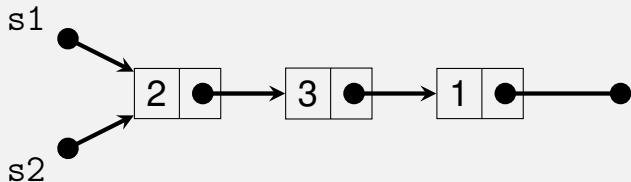
member-wise initialization: copies the
topn pointer only.

```
...  
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

What has gone wrong?



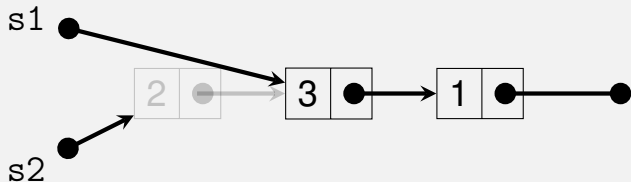
...

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

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

What has gone wrong?



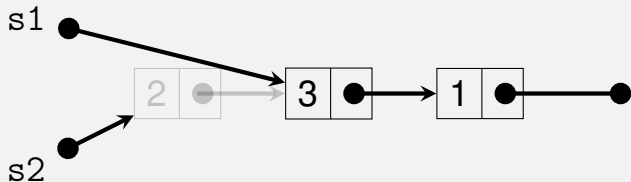
...

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

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

What has gone wrong?



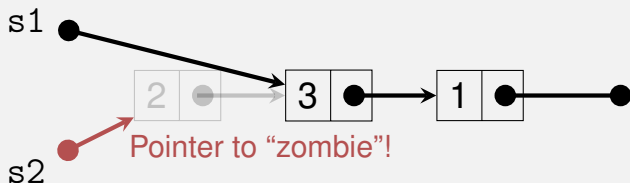
...

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

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

```
s2.pop ();
```

What has gone wrong?



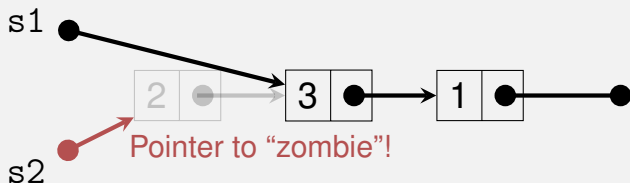
...

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

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

```
s2.pop ();
```


What has gone wrong?



...

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

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

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

The actual problem

Already this goes wrong:

```
{  
    stack s1;  
    s1.push(1);  
    stack s2 = s1;  
}
```

When leaving the scope, both stacks are deconstructed. But both stacks try to delete the same data, because both stacks have *access to the same pointer*.

Possible solutions

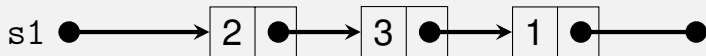
Smart-Pointers (we will not go into details here):

- Count the number of pointers referring to the same objects and delete only when that number goes down to 0
`std::shared_pointer`
- Make sure that not more than one pointer can point to an object:
`std::unique_pointer.`

or:

- We make a real copy of all data – as discussed below.

We make a real copy



...

```
stack s2 = s1;
```

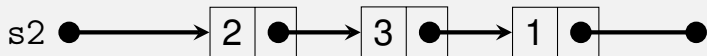
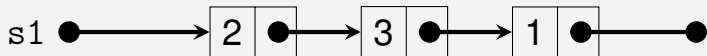
```
std::cout << s2 << "\n";
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

We make a real copy



...

```
stack s2 = s1;
```

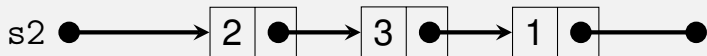
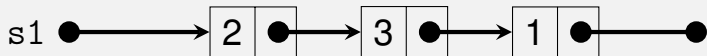
```
std::cout << s2 << "\n";
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

We make a real copy



...

```
stack s2 = s1;
```

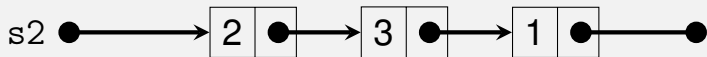
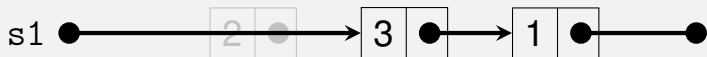
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

We make a real copy



...

```
stack s2 = s1;
```

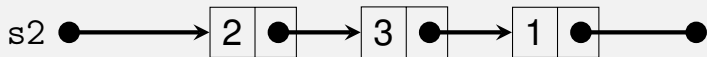
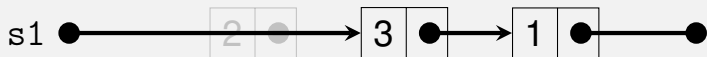
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

We make a real copy



...

```
stack s2 = s1;
```

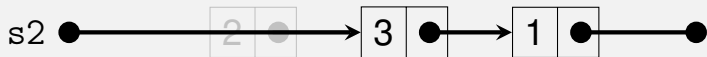
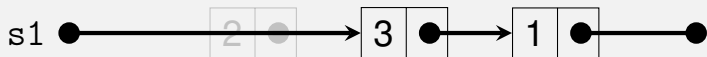
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```


We make a real copy



...

```
stack s2 = s1;
```

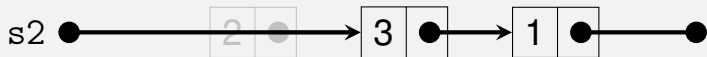
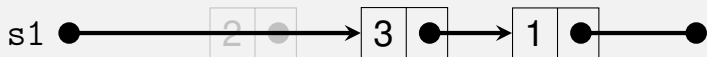
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

We make 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

$$T(\text{const } T\& x);$$

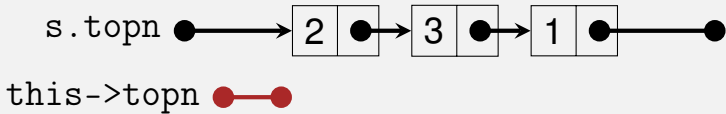
- is automatically called when values of type T are initialized with values of type T

$$T\ x = t; \quad (\text{t of type } T)$$
$$T\ x(t);$$

- 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

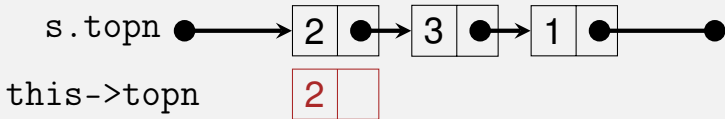
```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



prev

It works with a Copy Constructor

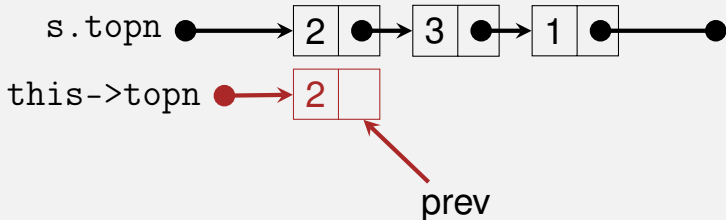
```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



prev

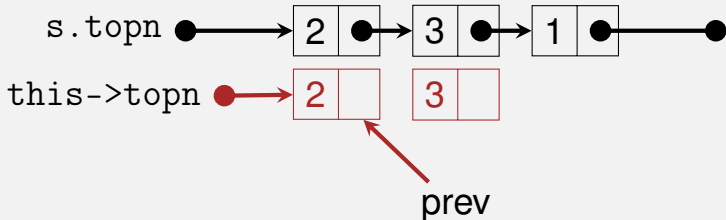
It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



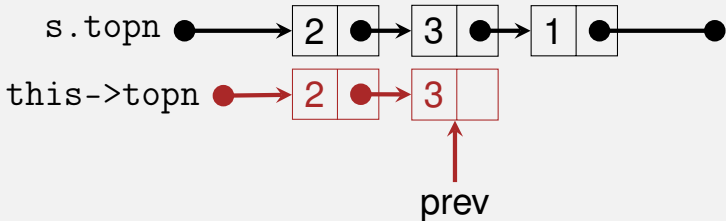
It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



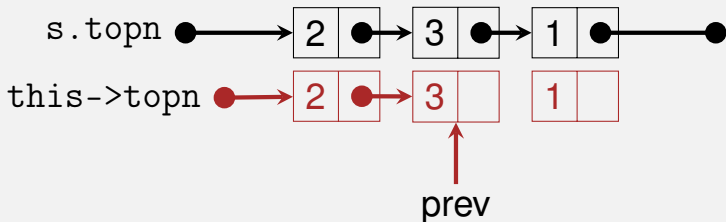
It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



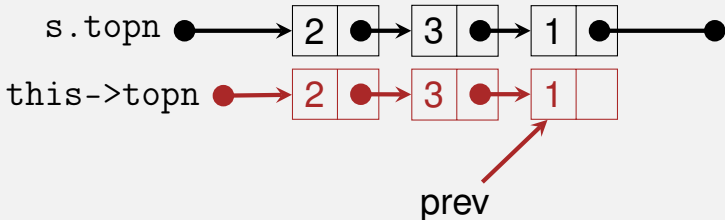
It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



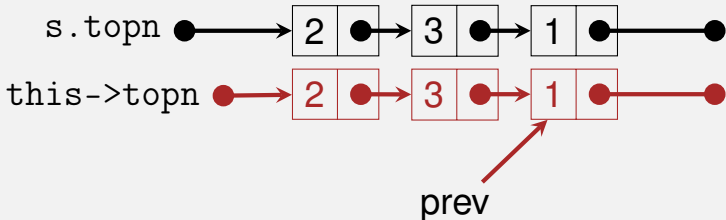
It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



Aside: copy recursively

```
llnode* copy (node* that){  
    if (that == nullptr) return nullptr;  
    return new llnode(that->value, copy(that->next));  
}
```

Elegant, isn't it?

⁷not an overflow of the stack that we are implementing but the call stack

Aside: copy recursively

```
llnode* copy (node* that){  
    if (that == nullptr) return nullptr;  
    return new llnode(that->value, copy(that->next));  
}
```

Elegant, isn't it? Why did we not do it like this?

⁷not an overflow of the stack that we are implementing but the call stack

Aside: copy recursively

```
llnode* copy (node* that){  
    if (that == nullptr) return nullptr;  
    return new llnode(that->value, copy(that->next));  
}
```

Elegant, isn't it? Why did we not do it like this?

Reason: linked lists can become very long. `copy` could then lead to stack overflow⁷. Stack memory is usually smaller than heap memory.

⁷not an overflow of the stack that we are implementing but the call stack

Initialization \neq Assignment!

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

```
stack s2 = s1; // Initialisierung
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
s2.pop (); // ok: Copy Constructor!
```

Initialization \neq 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!

```
// POST: *this (left operand) becomes a  
//           copy of s (right operand)  
stack& stack::operator= (const stack& s)
```

It works with an Assignment Operator!

```
// POST: *this (left operand) becomes a  
//           copy of s (right operand)  
stack& stack::operator= (const stack& s){  
    if (topn != s.topn){ // no self-assignment
```

It works with an Assignment Operator!

```
// POST: *this (left operand) becomes a  
//           copy of s (right operand)  
stack& stack::operator= (const stack& s){  
    if (topn != s.topn){ // no self-assignment  
        stack copy = s; // Copy Construction
```

It works with an Assignment Operator!

```
// POST: *this (left operand) becomes a
//           copy of s (right operand)
stack& stack::operator= (const stack& s){
    if (topn != s.topn){ // no self-assignment
        stack copy = s; // Copy Construction
        std::swap(topn, copy.topn); // now copy has the garbage
```

It works with an Assignment Operator!

```
// POST: *this (left operand) becomes a
//           copy of s (right operand)
stack& stack::operator= (const stack& s){
    if (topn != s.topn){ // no self-assignment
        stack copy = s; // Copy Construction
        std::swap(topn, copy.topn); // now copy has the garbage
    } // copy is cleaned up -> deconstruction
    return *this; // return as L-Value (convention)
}
```

It works with an Assignment Operator!

```
// POST: *this (left operand) becomes a
//           copy of s (right operand)
stack& stack::operator= (const stack& s){
    if (topn != s.topn){ // no self-assignment
        stack copy = s; // Copy Construction
        std::swap(topn, copy.topn); // now copy has the garbage
    } // copy is cleaned up -> deconstruction
    return *this; // return as L-Value (convention)
}
```

Cool trick! 😊

Done

```
class stack{
public:
    stack(); // constructor
    ~stack(); // destructor
    stack(const stack& s); // copy constructor
    stack& operator=(const stack& s); // assignment operator

    void push(int value);
    void pop();
    int top() const;
    bool empty() const;
    void print(std::ostream& out) const;
private:
    llnode* topn;
}
```


Dynamic Datatype

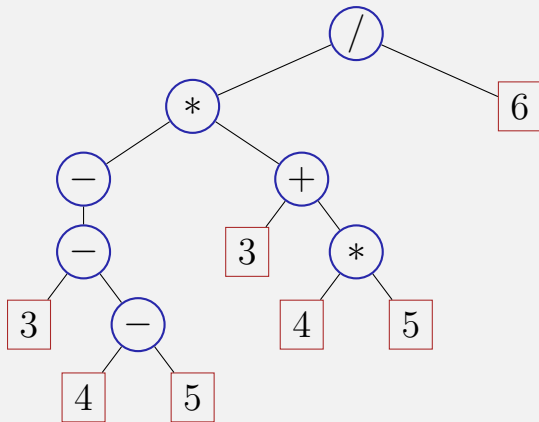
- Type that manages dynamic memory (e.g. our class for a stack)
- Minimal Functionality:
 - Constructors
 - Destructor
 - Copy Constructor
 - Assignment Operator

Dynamic Datatype

- Type that manages dynamic memory (e.g. our class for a stack)
 - Minimal Functionality:
 - Constructors
 - Destructor
 - Copy Constructor
 - Assignment Operator
- } *Rule of Three:* if a class defines at least one of them, it must define all three

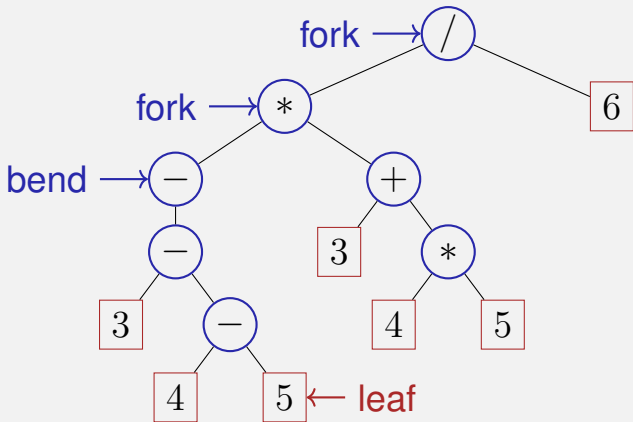
(Expression) Trees

$$-(3-(4-5))*(3+4*5)/6$$



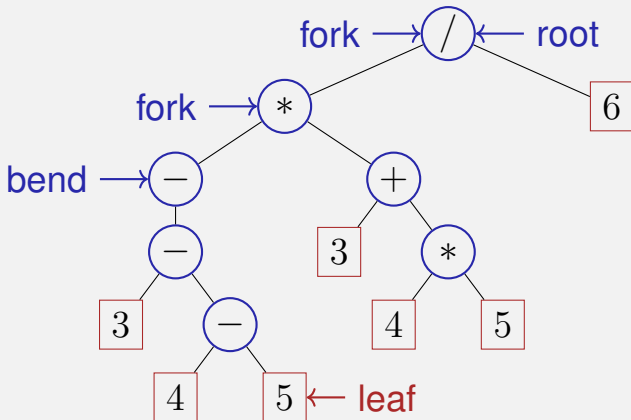
(Expression) Trees

$$-(3-(4-5))*(3+4*5)/6$$



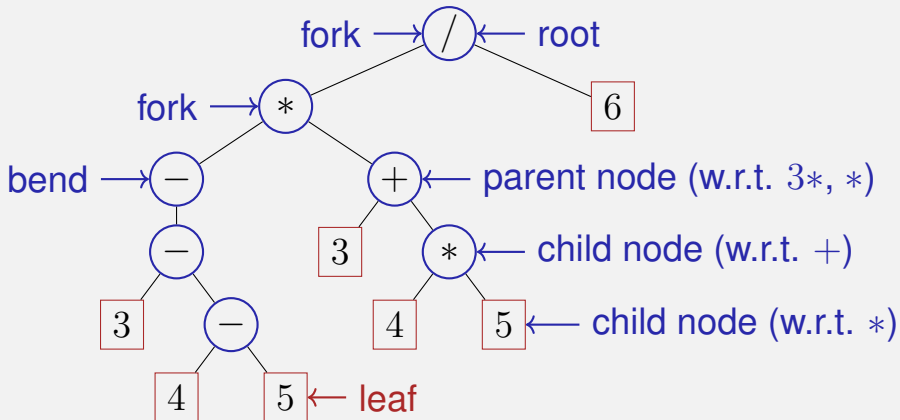
(Expression) Trees

$$-(3-(4-5))*(3+4*5)/6$$

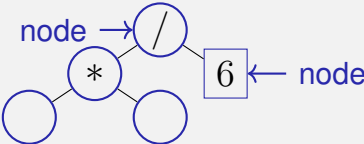


(Expression) Trees

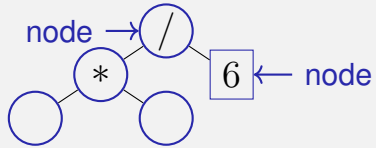
$$-(3-(4-5))*(3+4*5)/6$$



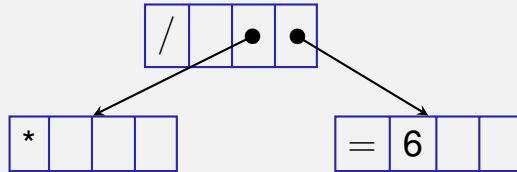
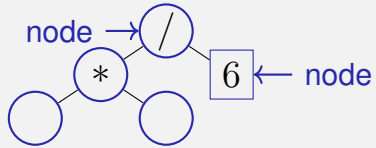
Nodes: Forks, Bends or Leaves



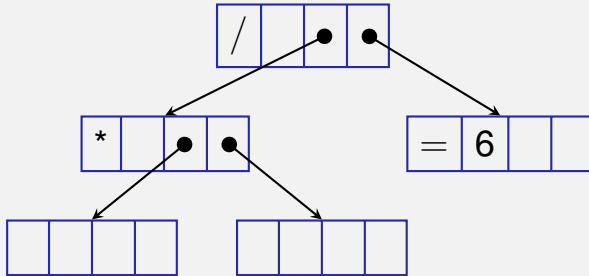
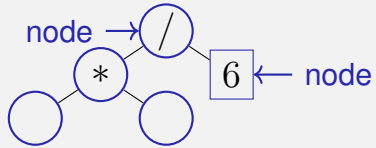
Nodes: Forks, Bends or Leaves



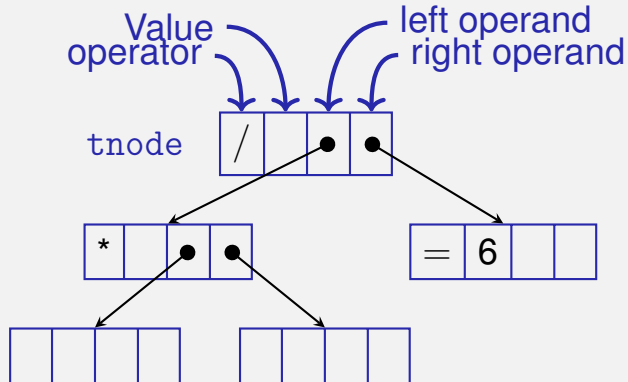
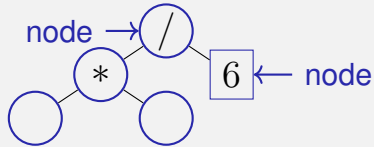
Nodes: Forks, Bends or Leaves



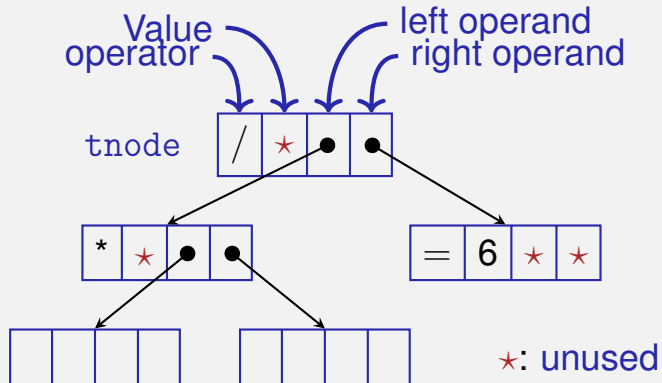
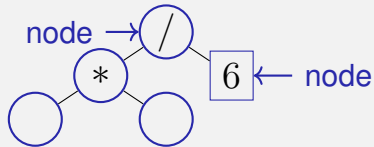
Nodes: Forks, Bends or Leaves



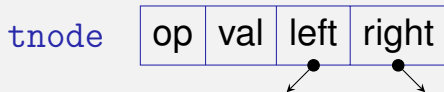
Nodes: Forks, Bends or Leaves



Nodes: Forks, Bends or Leaves



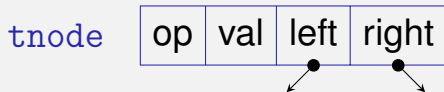
Nodes (struct tnode)



```
struct tnode {
    char op; // leaf node: op is '='
             // internal node: op is '+', '-', '*', or '/'
    double val;
    tnode* left;
    tnode* right;

    tnode(char o, double v, tnode* l, tnode* r)
        : op(o), val(v), left(l), right(r) {}
};
```

Nodes (struct tnode)



```
struct tnode {
    char op; // leaf node: op is '='
             // internal node: op is '+', '-', '*', or '/'
    double val;
    tnode* left; // == nullptr for unary minus
    tnode* right;

    tnode(char o, double v, tnode* l, tnode* r)
        : op(o), val(v), left(l), right(r) {}
};
```

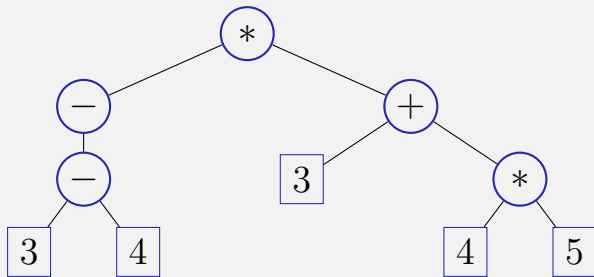
Nodes (struct tnode)



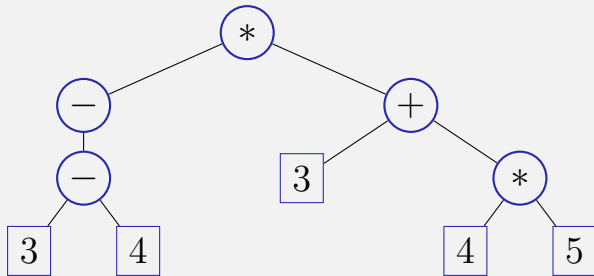
```
struct tnode {
    char op; // leaf node: op is '='
             // internal node: op is '+', '-', '*', or '/'
    double val;
    tnode* left; // == nullptr for unary minus
    tnode* right;

    tnode(char o, double v, tnode* l, tnode* r)
        : op(o), val(v), left(l), right(r) {}
};
```

Size = Count Nodes in Subtrees

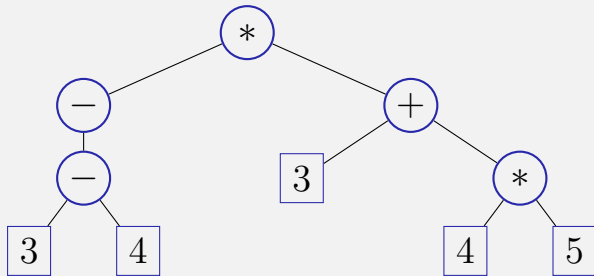


Size = Count Nodes in Subtrees



- Size of a leaf: 1
- Size of other nodes: 1 + sum of child nodes' size

Size = Count Nodes in Subtrees



- Size of a leaf: 1
- Size of other nodes: 1 + sum of child nodes' size
- E.g. size of the "+"-node is 5

Count Nodes in Subtrees

```
// POST: returns the size (number of nodes) of
//       the subtree with root n
int size (const tnode* n) {
    if (n){ // shortcut for n != nullptr
        return size(n->left) + size(n->right) + 1;
    }
    return 0;
}
```



Evaluate Subtrees

```
// POST: evaluates the subtree with root n
```

```
double eval(const tnode* n){
    assert(n);
    if (n->op == '=') return n->val; ← leaf...
    double l = 0;                               ... or fork:
    if (n->left) l = eval(n->left); ← op unary, or left branch
    double r = eval(n->right); ← right branch
    switch(n->op){
        case '+': return l+r;
        case '-': return l-r;
        case '*': return l*r;
        case '/': return l/r;
        default: return 0;
    }
}
```



Cloning Subtrees

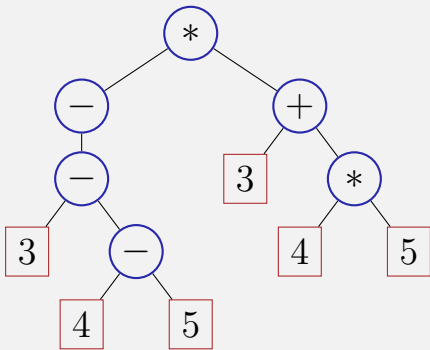
```
// POST: a copy of the subtree with root n is made
//       and a pointer to its root node is returned
tnode* copy (const tnode* n) {
    if (n == nullptr)
        return nullptr;
    return new tnode (n->op, n->val, copy(n->left), copy(n->right));
}
```



Felling Subtrees

```
// POST: all nodes in the subtree with root n are deleted
```

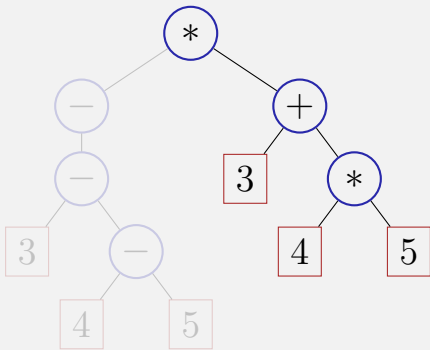
```
void clear(tnode* n) {  
    if(n){  
        clear(n->left);  
        clear(n->right);  
        delete n;  
    }  
}
```



Felling Subtrees

```
// POST: all nodes in the subtree with root n are deleted
```

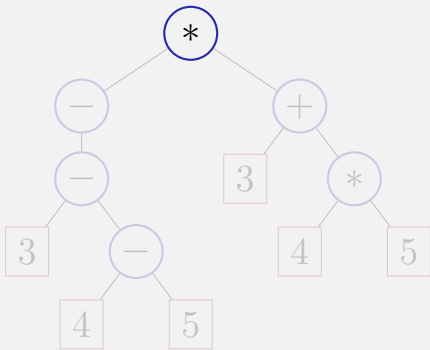
```
void clear(tnode* n) {  
    if(n){  
        clear(n->left);  
        clear(n->right);  
        delete n;  
    }  
}
```



Felling Subtrees

```
// POST: all nodes in the subtree with root n are deleted
```

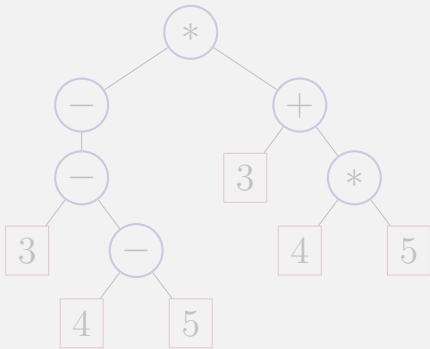
```
void clear(tnode* n) {  
    if(n){  
        clear(n->left);  
        clear(n->right);  
        delete n;  
    }  
}
```



Felling Subtrees

```
// POST: all nodes in the subtree with root n are deleted
```

```
void clear(tnode* n) {  
    if(n){  
        clear(n->left);  
        clear(n->right);  
        delete n;  
    }  
}
```



Using Expression Subtrees

```
// Construct a tree for  $1 - (-(3 + 7))$ 
tnode* n1 = new tnode('=', 3, nullptr, nullptr);
tnode* n2 = new tnode('=', 7, nullptr, nullptr);
tnode* n3 = new tnode('+', 0, n1, n2);
tnode* n4 = new tnode('-', 0, nullptr, n3);
tnode* n5 = new tnode('=', 1, nullptr, nullptr);
tnode* root = new tnode('-', 0, n5, n4);

// Evaluate the overall tree
std::cout << "1 - (-(3 + 7)) = " << eval(root) << '\n';


// Evaluate a subtree
std::cout << "3 + 7 = " << eval(n3) << '\n';

clear(root); // free memory
```

Planting Trees

```
class texpression {
public:
    texpression (double d)
        : root (new tnode ('=', d, 0, 0)) {}
    ...
private:
    tnode* root;
};
```

creates a tree with
one leaf

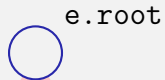


Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



*this



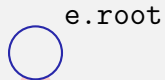
e

Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



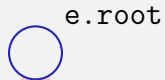
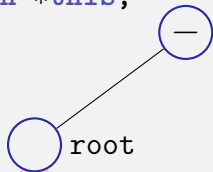
*this



e

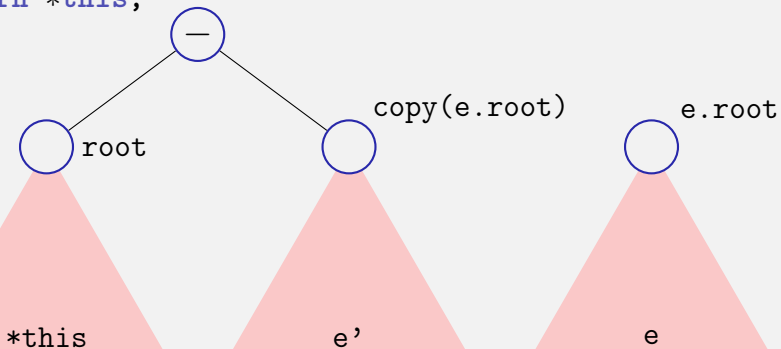
Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



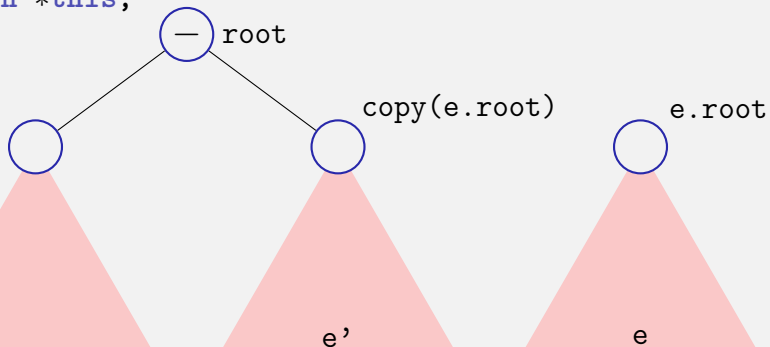
Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



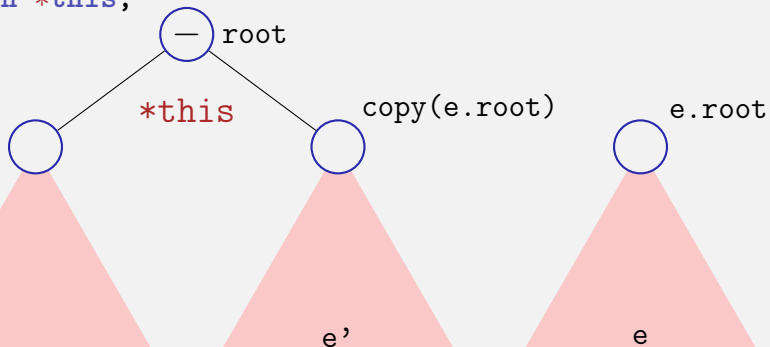
Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



Raising Trees

```
texpression operator- (const texpression& l,  
                      const texpression& r){  
    texpression result = l;  
    return result -= r;  
}
```

```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```

Raising Trees

```
texpression operator- (const texpression& l,  
                      const texpression& r){  
    texpression result = l;  
    return result -= r;  
}
```

```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```

3

Raising Trees

```
texpression operator- (const texpression& l,  
                      const texpression& r){  
    texpression result = l;  
    return result -= r;  
}
```

```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```

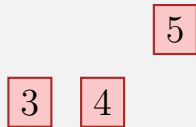
3

4

Raising Trees

```
texpression operator- (const texpression& l,  
                      const texpression& r){  
    texpression result = l;  
    return result -= r;  
}
```

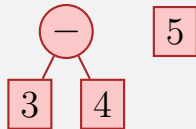
```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```



Raising Trees

```
expression operator- (const expression& l,  
                    const expression& r){  
    expression result = l;  
    return result -= r;  
}
```

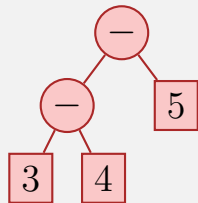
```
expression a = 3;  
expression b = 4;  
expression c = 5;  
expression d = a-b-c;
```



Raising Trees

```
expression operator- (const expression& l,  
                      const expression& r){  
    expression result = l;  
    return result -= r;  
}
```

```
expression a = 3;  
expression b = 4;  
expression c = 5;  
expression d = a-b-c;
```



Rule of three: Clone, reproduce and cut trees

```
expression::~~expression(){  
    clear(root);  
}
```

```
expression::expression (const expression& e)  
    : root(copy(e.root)) { }
```

```
expression::expression& operator=(const expression& e){  
    if (root != e.root){  
        expression cp = e;  
        std::swap(cp.root, root);  
    }  
    return *this;  
}
```


Concluded

```
class texpression{
public:
    texpression (double d); // constructor
    ~texpression(); // destructor
    texpression (const texpression& e); // copy constructor
    texpression& operator=(const texpression& e); // assignment op
    texpression operator-();
    texpression& operator-=(const texpression& e);
    texpression& operator+=(const texpression& e);
    texpression& operator*=(const texpression& e);
    texpression& operator/=(const texpression& e);
    double evaluate();
private:
    tnode* root;
};
```

From values to trees!

```
// term = factor { "*" factor | "/" factor }
double term (std::istream& is){
    double value = factor (is);
    while (true) {
        if (consume (is, '*'))
            value *= factor (is);
        else if (consume (is, '/'))
            value /= factor (is);
    }
    return value;
}
```

calculator.cpp
(expression value)

From values to trees!

```
using number_type = double;
```

```
// term = factor { "*" factor | "/" factor }  
number_type term (std::istream& is){  
    number_type value = factor (is);  
    while (true) {  
        if (consume (is, '*'))  
            value *= factor (is);  
        else if (consume (is, '/'))  
            value /= factor (is);  
        else  
            return value;  
    }  
}
```

double_calculator.cpp
(expression value)

From values to trees!

```
using number_type = texpression ;
```

```
// term = factor { "*" factor | "/" factor }  
number_type term (std::istream& is){  
    number_type value = factor (is);  
    while (true) {  
        if (consume (is, '*'))  
            value *= factor (is);  
        else if (consume (is, '/'))  
            value /= factor (is);  
    }  
    return value;  
}
```

double_calculator.cpp
(expression value)

→

texpression_calculator.cpp
(expression tree)

Concluding Remark

- In this lecture, we have intentionally refrained from implementing member functions in the node classes of the list or tree.⁸
- When there is inheritance and polymorphism used, the implementation of the functionality such as evaluate, print, clear (etc..) is better implemented in member functions.
- In any case it is not a good idea to implement the memory management of the composite data structure list or tree within the nodes.

⁸Parts of the implementations are even simpler (because the case `n==nullptr` can be caught more easily