TDDE18 & 726G77
Classes & Pointers

Christoffer Holm

Department of Computer and information science

LINKÖPING
UNIVERSITY

LINKÖPING UNIVERSITY

Classes

What is a class?

```
struct Date
{
  int day;
  int month;
  int year;
};


bool operator<(Date d1, Date d2)
{
  // ...
}

Date next_date(Date d)
{
  // ...
}
```

Classes

What is a class?

```cpp
struct Date
{
  int day;
  int month;
  int year;
};


bool operator<(Date d1, Date d2)
{
  // ...
}

Date next_date(Date d)
{
  // ...
}
```

```cpp
class Date
{
public:
  int day;
  int month;
  int year;

  bool operator<(Date d)
  {
    // ...
  }

  Date next_date()
  {
    // ...
  }
};
```

Classes

What is a class?

- A `class` is like a `struct`.

- With classes we *couple* data and functionality.

- That is; we store the functions and the data in a neatly packaged structure called a class.

Classes

How does it work?

```
Date today {27, 9, 2019};
Date tomorrow {next_date(today)};

if (today < tomorrow)
{
  // ...
}
```

Classes

How does it work?

```
Date today {27, 9, 2019};
Date tomorrow {next_date(today)};

if (today < tomorrow)
{
  // ...
}
```

```
Date today {27, 9, 2019};
Date tomorrow {today.next_date()};

if (today < tomorrow)
{
  // ...
}
```

Classes

How does it work?

```
Date today {27, 9, 2019};
Date tomorrow {next_date(today)};

if (operator<(today, tomorrow))
{
  // ...
}
```

```
Date today {27, 9, 2019};
Date tomorrow {today.next_date()};

if (today.operator<(tomorrow))
{
  // ...
}
```

Classes

How does it work?

- The functions that are declared inside a class are called *member functions*.

- They are not like ordinary functions, because they have a hidden argument.

- The hidden argument is the object we are calling this function from.

- A member function is called like this: `obj.function()` where `obj` is the current object we are calling this function on (the hidden parameter).

Classes

this

```
struct Date
{
  // ...
};

void increase_year(Date& date)
{
  ++(date.year);
}
// ...
```

```
class Date
{
  // ...

  void increase_year()
  {
    ++(this->year);
  }
  // ...
};
```

Classes

this

```
struct Date
{
  // ...
};

void increase_year(Date& date)
{
  ++(date.year);
}
// ...
```

```
class Date
{
  // ...

  void increase_year()
  {
    ++year;
  }
  // ...
};
```

Classes

`this`

- You can access this hidden parameter through the `this` keyword.

- Instead of accessing data members (fields) through `.` we use `->` for `this`.

- However, most of the time we don't need to use `this` at all.

- As long as there is no ambiguity what is meant we can just skip `this->` completely.

Classes

When is `this` mandatory?

```cpp
class Date
{
  // ...
  void set_year(int year)
  {
    year = year;
  }
};
```

Classes

When is `this` mandatory?

```cpp
class Date
{
  // ...
  void set_year(int year)
  {
    this->year = year;
  }
};
```

Classes

When is `this` mandatory?

```cpp
class Date
{
  // ...
  void set_year(int y)
  {
    year = y;
  }
};
```

Classes

When is `this` mandatory?

- When there is a more *local* variable with the same name as the member you are trying to access.

- In this example we have two things called `year`, the data member in the class and the parameter.

- The parameter is closer (more local), so the compiler will automatically choose that whenever we just write `year`.

- In this case we have to further specify that it is the member we want, which we do by using `this`.

Classes

Why though?

```
Date today {27, 9, 2019};

today.day = 48; // should not be allowed
```

Classes

Why though?

- When coupling data and functionality we want the user of our class to only use the specified functions.

- The user should not have the power to modify data fields however they want.

- In this case this would allow the user to set a nonsensical value to data members.

- Therefore it would be nice to hide things inside our class.

- This is done with *access levels*.

IL.U LINKÖPING
UNIVERSITY

Classes

private & public

```cpp
class Date
{
public:
  bool operator<(Date const& d) const
  {
    // ...
  }

  Date next_date()
  {
    // ...
  }

private:
  int day;
  int month;
  int year;
};
```

Classes

private & public

```cpp
class Date
{
public:
  bool operator<(Date const& d) const
  {
    // ...
  }

  Date next_date()
  {
    // ...
  }

private:
  int day;
  int month;
  int year;
};
```

```cpp
int main()
{

  Date today {27, 9, 2019};
  // not allowed
  today.day = 48;
}
```

Classes

private & public

```cpp
class Date
{
public:
  bool operator<(Date const& d) const
  {
    // ...
  }

  Date next_date()
  {
    // ...
  }

private:
  int day;
  int month;
  int year;
};
```

```cpp
int main()
{
  // will not work
  Date today {27, 9, 2019};
  // not allowed
  today.day = 48;
}
```

Classes

`private` & `public`

- We can start blocks of access levels.

- All functions and data members placed after a `public` declaration will be freely available from outside of the class through an object.

- All members placed after a `private` declaration will only be accessible from inside the member functions.

- I.e. `private` allows us to hide away things that only the class is allowed to see.

Classes

private & public

- However there is a problem.

- Once the data members are private the compiler cannot initialize those data members as they are now private, meaning only a member function has access to them.

- Fortunately we can create special member functions called *constructors* that handles the initialization of data members.

Classes

Constructors

```cpp
class Date
{
public:
  Date(int d, int m, int y)
    // member initialization list
    : day{d}, month{m}, year{y}
  {
  }

  // ...

private:
  int day;
  int month;
  int year;
};
```

Classes

Constructors

```cpp
class Date
{
public:
  Date(int d, int m, int y)
    // member initialization list
    : day{d}, month{m}, year{y}
  {
  }

  // ...

private:
  int day;
  int month;
  int year;
};
```

```cpp
int main()
{
  // works!
  Date today {27, 9, 2019};
  // not allowed
  today.day = 48;
}
```

Classes

Constructors

- Constructors work like normal functions with 2 exceptions:

- A constructor is only called when an object is created,

- they initializes variables inside the *member initialization list*.

Classes

Constructors

- You can declare multiple constructors as long as they have a unique set of parameters.

- When initializing the object the user passes in parameters and the appropriate constructor will be called based on the parameters passed in.

Classes

Member initialization list

- The member initialization list is a comma-separated list of all the data members that this constructor is supposed to initialize.

- Each member in the list must be initialized with either { . . . } or (. . .).

Classes

Declaration & Definition

```cpp
class Date;                      // class declaration
class Date                       // class definition
{
  // ...
  Date(int d, int m, int y); // declare a construtor
  void increase_year();        // declare a member function
  // ...
private:                        // data members
  int day;
  int month;
  int year;
};

Date::Date(int d, int m, int y) // define a construtor
  : day{d}, month{m}, year{y}   // member initialization list
{ }

void Date::increase_year()      // define a member function
{
  ++year;
}
```

Classes

Declaration & Definition

- It is possible to declare a class before defining it, just as we can do with functions.

- You can also declare member functions before defining them.

- To define a member function after the class has been defined you add the `ClassName::` before the member function name to communicate which class this member function resides in.

Classes

`const` member functions

```
class Date
{
  // ...
  int get_day()
  {
    day = 7; // allowed
    return day;
  }
};
```

Classes

const member functions

```
class Date
{
  // ...
  int get_day() const
  {
    day = 7; // NOT allowed
    return day;
  }
};
```

Classes

const member functions

- Inside a member function you have access to private data members.

- This means that a function can modify these members however they want.

- If another programmer decides to use your function they might not expect it to modify anything.

- You can declare your member functions as const which disallows they modification of data members inside this specific function.

LiU LINKÖPING
UNIVERSITY

Classes

`const` member functions

You might want to do this for two major reasons:

1. This makes sure that you, the programmer doesn't accidentally change a variable when you are not supposed to.

2. It also communicates to other programmers that this member function won't change the state of the object, meaning it is always safe to call in all contexts.

Classes

const member functions

```cpp
class Date
{
  // ...
  int get_day()
  {
    return day;
  }
  // ...
};
```

```cpp
Date d1{27, 9, 2019};
cout << d1.get_day() << endl;

Date const d2{28, 9, 2019};

// doesn't work
cout << d2.get_day() << endl;
```

Classes

const member functions

```cpp
class Date
{
  // ...
  int get_day() const
  {
    return day;
  }
  // ...
};
```

```cpp
Date d1{27, 9, 2019};
cout << d1.get_day() << endl;

Date const d2{28, 9, 2019};

// works!
cout << d2.get_day() << endl;
```

Classes

const member functions

- If an object is declared as const then no data members can be modified.

- This means that only member functions that are declared as const are allowed to be called from these objects, with the exception of constructors and the destructor.

Classes

Inner class

```cpp
class Outer
{
public:

  void fun();


  class Inner
  {
  public:

    void fun();

  };

};
```

```cpp
void Outer::fun()
{
  // ...
}

void Outer::Inner::fun()
{
  // ...
}
```

```cpp
Outer o{};
o.fun();

Outer::Inner i{}; // works!
i.fun();
```

Classes

Inner class

```cpp
class Outer
{
public:

  void fun();

private:

  class Inner
  {
  public:

    void fun();

  };

};
```

```cpp
void Outer::fun()
{
  // ...
}

void Outer::Inner::fun()
{
  // ...
}
```

```cpp
Outer o{};
o.fun();

Outer::Inner i{}; // doesn't work
i.fun();
```

Classes

Inner class

- It is possible to declare classes inside other classes.

- These classes adhere to the access level they are placed in.

- Meaning an inner class declared as `private` is not accessible from the outside.

- To define member functions of an inner class you first have to access the outer class, and then the inner class.

- So you write `Outer::Inner::` before the function name.

IIU LINKÖPING
UNIVERSITY

Classes

friend

```
class Date
{
  // ...

private:

  int day;
  int month;
  int year;

  friend bool same_month(Date d1, Date d2);
};

bool same_month(Date d1, Date d2)
{
  return d1.year == d2.year && d1.month == d2.month;
}
```

Classes

friend

- It is possible to declare a function as a friend to your class.

- This allows that friend to access the private members.

- Should only be used if absolutely necessary, since it couples the class with a function that is not a member.

LINKÖPING
UNIVERSITY

Pointers

What is a pointer?

```
int x{5};
int y{7};
```

Pointers

What is a pointer?

```
int x{5};
int y{7};
```

x $\boxed{5}$

y $\boxed{7}$

Pointers

What is a pointer?

```
int x{5};
int y{7};
int* ptr{};
```

x | 5 |

y | 7 |

Pointers

What is a pointer?

```
int x{5};
int y{7};
int* ptr{};
```

ptr □   x 5

y 7

Pointers

What is a pointer?

```
int x{5};
int y{7};
int* ptr{&x};
```

ptr ◻   x 5

              y 7

Pointers

What is a pointer?

```
int x{5};
int y{7};
int* ptr{&x};
```

ptr → x 5

y 7

Pointers

What is a pointer?

```
int x{5};
int y{7};
int* ptr{&x};
ptr = &y;
```

ptr ☐   x `5`

y `7`

Pointers

What is a pointer?

```
int x{5};
int y{7};
int* ptr{&x};
ptr = &y;
```

ptr   x ⬚ 5

y ⬚ 7

LINKÖPING UNIVERSITY
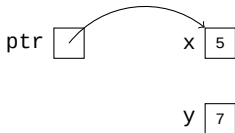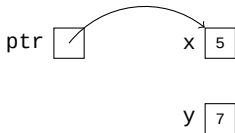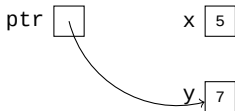
Pointers

What is a pointer?

- A pointer is a variable that stores an address to some location in memory.

- A pointer can only point to a specific data type (specified by the programmer).

- This means that when the pointer retrieves the value stored at the address, it will always be the specified type.

- Read pointer declarations backwards: `int* x` reads: `x` is a pointer(`*`) to an `int`.

Pointers

What is a pointer?

- Each variable has a location in memory (they have an address) which can be retrieved with `operator&`, as such: `&x`.

- A pointer can therefore work like a reference, by storing a specific variables address in the pointer.

- The difference between a reference and a pointer is that pointers can change what they point to while references cannot.

Pointers

How to use a pointer

```
int x{5};
int *ptr{&x};

cout << *ptr << endl;
```

ptr [ ] ⟶ x [5]

Pointers

How to use a pointer

```
int x{5};
int *ptr{&x};

cout << *ptr << endl;
```

ptr [ ]   x [5]

Pointers

How to use a pointer

```
int x{5};
int *ptr{&x};

cout << *ptr << endl;
```

ptr [ ]   x [5]
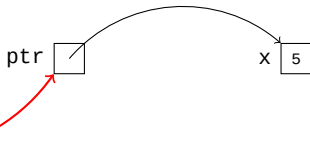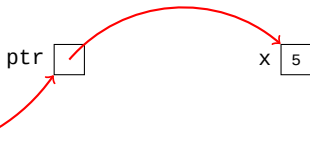
Pointers

How to use a pointer

- To access the data on the other end of the pointer (i.e. the thing it points to) we have to *dereference* the pointer.

- You dereference a pointer with operator*.

- Like this: *ptr.

Pointers

Pointing to nothing

```
int* ptr{};
ptr == nullptr;
```

Pointers

Pointing to nothing

- The default value of pointers is a special value called `nullptr`.

- A pointer that is `nullptr` points at nothing.

- This is also a big difference between pointers and references; a reference **must** refer to something.

Pointers

Manual memory

```
int* ptr{};
```

Pointers

Manual memory

```
int* ptr{};
```

ptr ◻

Pointers

Manual memory

```
int* ptr{new int{5}};
```

ptr ◻

Pointers

Manual memory

```
int* ptr{new int{5}};
```

ptr ◻   5

Pointers

Manual memory

```
int* ptr{new int{5}};
```

ptr → 5

Pointers

Manual memory

```
int* ptr{new int{5}};
*ptr = 7;
```

ptr [ ] ⟶ 5

Pointers

Manual memory

```
int* ptr{new int{5}};
*ptr = 7;
```

ptr

Pointers

Manual memory

```
int* ptr{new int{5}};
*ptr = 7;
delete ptr;
```

ptr [→] 7

Pointers

Manual memory

```
int* ptr{new int{5}};
*ptr = 7;
delete ptr;
```

ptr

Pointers

Manual memory

```
int* ptr{new int{5}};
*ptr = 7;
delete ptr;
ptr = nullptr;
```

ptr

Pointers

Manual memory

```cpp
int* ptr{new int{5}};
*ptr = 7;
delete ptr;
ptr = nullptr;
```

ptr

Pointers

Manual memory

- A pointer doesn't have to point to a variable.

- You can place data inside the memory directly with `new`.

- This is called *allocating* data.

- When allocating memory it is *your* responsibility to destroy the object that lies in that memory.

- You can *deallocate* the memory by calling `delete` on the pointer.

Pointers

Manual memory

- You can only deallocate memory that has been allocated (i.e. for every `delete` there must be a `new`).

- You can only `delete` memory once.

- Calling `delete` does *not* remove the pointer!

- Meaning the pointer will still point to the memory address, but the object has been removed, so it points to nonsense.

LINKÖPING UNIVERSITY

List lab

- In lab 3 you are going to create a *linked list*.

- This is a list that consists of values that the user inserts.

- Each value is stored in a *node* which then points to the next value in the list.

- On the next slide is a simple implementation of a *node*.

List lab

Node

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

Node n2 {2, nullptr};
Node n1 {8, &n2};

Node* first {&n1};
```

List lab

Node

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

Node n2 {2, nullptr};
Node n1 {8, &n2};

Node* first {&n1};
```
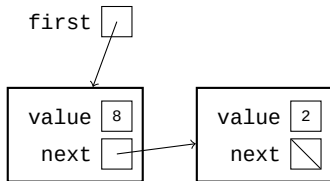
List lab

Node

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

Node n2 {2, nullptr};
Node n1 {8, &n2};

Node* first {&n1};
```
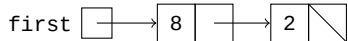
List lab

Node

- `first` points to whichever element is first in the list.

- Each `Node` then points to the next element in the list.

- Once the `next` pointer is `nullptr` we have reached the end of the list.

List lab

Accessing data in Node

```
Node n2 {2, nullptr};
Node n1 {8, &n2};

Node* first {&n1};

cout << (*first).value << endl;
```

List lab

Accessing data in Node

```
Node n2 {2, nullptr};
Node n1 {8, &n2};

Node* first {&n1};

cout << first->value << endl;
```

List lab

Accessing data in Node

- You access data members in an object that a pointer points to by either:

- Dereferencing the pointer to get normal access ((*first).value), or

- Using the -> operator (first->value).

- These two ways are exactly the same.

- It is recommended to use ->, since it is easier on the eyes.

IIₒU LINKÖPING
UNIVERSITY

List lab

List

```
class List
{
public:

  // ...

private:
  Node* first{};
};
```

List lab

`insert`

```cpp
class List
{
public:

  void remove();
  void insert(int value)
  {
    Node* tmp{new Node{value}};
    tmp->next = first;
    first = tmp;
  }

private:
  Node* first{};
};
```

first

List lab

insert

```cpp
class List
{
public:

  void remove();
  void insert(int value)
  {
    Node* tmp{new Node{value}};
    tmp->next = first;
    first = tmp;
  }

private:
  Node* first{};
};
```

first   9

List lab

`insert`

```
class List
{
public:

  void remove();
  void insert(int value)
  {
    Node* tmp{new Node{value}};
    tmp->next = first;
    first = tmp;
  }

private:
  Node* first{};
};
```

first ▧   ┌─9─┬──┐

tmp ┌──┬──┐

List lab

insert

```
class List
{
public:

  void remove();
  void insert(int value)
  {
    Node* tmp{new Node{value}};
    tmp->next = first;
    first = tmp;
  }

private:
  Node* first{};
};
```

List lab

insert

```cpp
class List
{
public:

  void remove();
  void insert(int value)
  {
    Node* tmp{new Node{value}};
    tmp->next = first;
    first = tmp;
  }

private:
  Node* first{};
};
```

List lab

insert

```cpp
class List
{
public:

  void remove();
  void insert(int value)
  {
    Node* tmp{new Node{value}};
    tmp->next = first;
    first = tmp;
  }

private:
  Node* first{};
};
```

List lab

insert

```
class List
{
public:

  void remove();
  void insert(int value)
  {
    Node* tmp{new Node{value}};
    tmp->next = first;
    first = tmp;
  }

private:
  Node* first{};
};
```

List lab

insert

```cpp
class List
{
public:

  void remove();
  void insert(int value)
  {
    Node* tmp{new Node{value}};
    tmp->next = first;
    first = tmp;
  }

private:
  Node* first{};
};
```

List lab

`insert`

```
class List
{
public:

  void remove();
  void insert(int value)
  {
    Node* tmp{new Node{value}};
    tmp->next = first;
    first = tmp;
  }

private:
  Node* first{};
};
```

List lab

insert

```cpp
class List
{
public:

  void remove();
  void insert(int value)
  {
    Node* tmp{new Node{value}};
    tmp->next = first;
    first = tmp;
  }

private:
  Node* first{};
};
```

List lab

`insert`

- We have to allocate a new `Node` at insertion, why?

- If `tmp` is a normal variable, then it will disappear once the `insert` function has finished.

- But we want it to persist until we want to remove it.

- Therefore we have to allocate nodes with `new`.

List lab

A problem!

```cpp
class List
{
public:

  void remove()
  {
    first = first->next;
  }
  void insert(int value);

private:
  Node* first{};
};
```
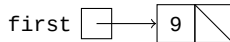
List lab

A problem!

```
class List
{
public:

  void remove()
  {
    first = first->next;
  }
  void insert(int value);

private:
  Node* first{};
};
```


first [ ] [9 |] [4 |]

Remove

List lab

A problem!

```cpp
class List
{
public:

  void remove()
  {
    first = first->next;
  }
  void insert(int value);

private:
  Node* first{};
};
```

List lab

A problem!

```cpp
class List
{
public:

  void remove()
  {
    first = first->next;
  }
  void insert(int value);

private:
  Node* first{};
};
```

List lab

A problem!

```
class List
{
public:

  void remove()
  {
    first = first->next;
  }
  void insert(int value);

private:
  Node* first{};
};
```


Memory leak
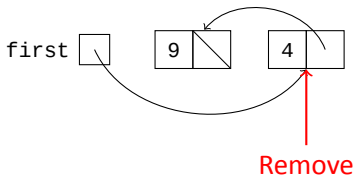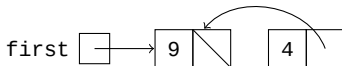
List lab

A problem!

- When removing a node it does not disappear by itself.

- Since we called new to create it, we have to call delete on the node for it to actually disappear.

- The pointer that kept track of our memory got lost, meaning there is no way for us to access it again.

- If this is done repeatedly by our program our memory will slowly be filled up by these inaccessible objects.

- This type of problem is called *memory leak*.

IIJ LINKÖPING
UNIVERSITY

List lab

Let's try again!

```cpp
class List
{
public:

  void remove()
  {
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
  void insert(int value);

private:
  Node* first{};
};
```


first □ → | 9 | | → | 4 | |

List lab

Let's try again!

```cpp
class List
{
public:

  void remove()
  {
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
  void insert(int value);

private:
  Node* first{};
};
```


first ⬚   9 ⬚   4 ⬚

Remove

List lab

Let's try again!

```cpp
class List
{
public:

  void remove()
  {
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
  void insert(int value);

private:
  Node* first{};
};
```

List lab

Let's try again!

```cpp
class List
{
public:

  void remove()
  {
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
  void insert(int value);

private:
  Node* first{};
};
```

List lab

Let's try again!

```cpp
class List
{
public:

  void remove()
  {
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
  void insert(int value);

private:
  Node* first{};
};
```

List lab

Let's try again!

```cpp
class List
{
public:

  void remove()
  {
    Node* tmp = first;
    first = first->next;
    delete tmp;
  }
  void insert(int value);

private:
  Node* first{};
};
```

List lab

Let's try again!

- Now we can insert and remove values correctly in our list without any memory leaks.

- This was done because we `delete` the node that gets removed, thus giving that memory back to the operating system.

- But there is still a problem... What happens when the list is destroyed?

- Then everything will leak since we haven't called `delete` on the nodes that are left.

IILU LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Special Member Functions

Destructor

```cpp
class List
{
public:

  List() // constructor
    : first{nullptr}
  {
  }

  ~List() // destructor
  {
    // go through each node in our list and call delete on them
  }

  void remove();
  void insert(int value);

private:
  Node* first{};
};
```

Special Member Functions

Destructor

- The constructor gets called when an object is created,

- There is a related function we can create called the *destructor*.

- The destructor is instead called when the object is destroyed.

- In the destructor we should remove anything that doesn't get removed by itself.

- The destructor has the same name as the class, but with a ~ added to the start.

III.U LINKÖPING
UNIVERSITY

Special Member Functions

Constructors & destructors

```
{
  List my_list{}; // the constructor is called

  // do things with the list

} // the destructor is called
```

Special Member Functions

Constructors & destructors

- When a `List` is created the constructor is called.

- Once the `List` falls out of scope, the destructor is called (this happens when the variable is destroyed).

Special Member Functions

Copies


```cpp
int main()
{
  List my_list{};
  my_list.insert(5);
  my_list.insert(2);
  {
    // copy my_list into copy
    List copy {my_list};
  }
}
```

Special Member Functions

Copies


```cpp
int main()
{
  List my_list{};
  my_list.insert(5);
  my_list.insert(2);
  {
    // copy my_list into copy
    List copy {my_list};
  }
}
```

Special Member Functions

Copies

```
int main()
{
  List my_list{};
  my_list.insert(5);
  my_list.insert(2);
  {
    // copy my_list into copy
    List copy {my_list};
  }
}
```


first

my_list

Special Member Functions

Copies

```cpp
int main()
{
  List my_list{};
  my_list.insert(5);
  my_list.insert(2);
  {
    // copy my_list into copy
    List copy {my_list};
  }
}
```


first

my_list

Special Member Functions

Copies

```cpp
int main()
{
  List my_list{};
  my_list.insert(5);
  my_list.insert(2);
  {
    // copy my_list into copy
    List copy {my_list};
  }
}
```


first

my_list

Already removed
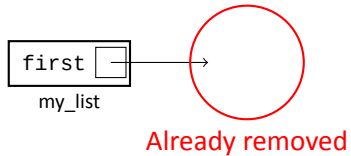
Special Member Functions

Copies

- We can copy objects in C++ and our list is no exception.

- The compiler doesn't understand how to *properly* copy our list.

- It will simply say that they point to exactly the same lists.

- This happens because the compiler will just copy the first pointer, not its content.

- This is called a *shallow copy*.

Special Member Functions
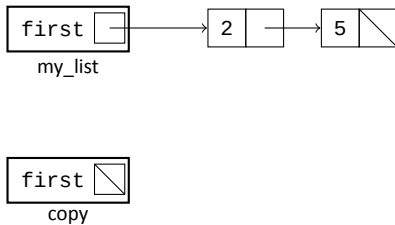
Deep copies

Special Member Functions
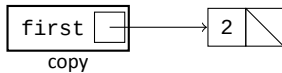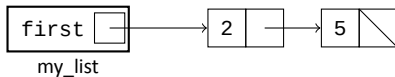
Deep copies

Special Member Functions
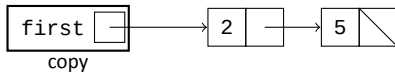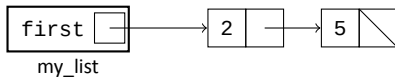
Deep copies

Special Member Functions

Deep copies

- A *deep copy* is when we copy everything, not just the pointers.

- In this particular case it is when we copy each node in my_list into copy (in the same order).

Special Member Functions

Copy constructor

```cpp
class List
{
public:

  List(); // default constructor

  List(List const& other)
  {
    // perform a deep copy of the lists
  }

  ~List(); // destructor
  void remove();
  void insert(int value);

private:
  Node* first{};
};
```

Special Member Functions

Copy constructor

- A copy constructor is a special constructor that tells the compiler how it is supposed to copy our class.

- It is the constructor that takes an object of the same class as a `const&`.

- The compile calls this constructor whenever it wants to copy an object of your class.

Special Member Functions

Copy assignment

```
List my_list{};
my_list.insert(5);
my_list.insert(2);

List copy{};
copy.insert(1);
copy.insert(4);

// copy assignment
copy = my_list;
```

Special Member Functions

Copy assignment

- Copy assignment is how we copy objects into each other *after* they have been created.

- They suffer from the same problem as the copy constructor;

- namely that they have to perform a deep copy, which the compiler doesn't know how to do.

- But they suffer from one other problem...

Special Member Functions
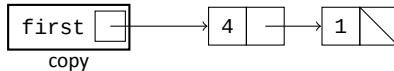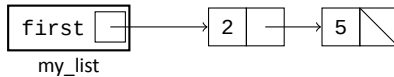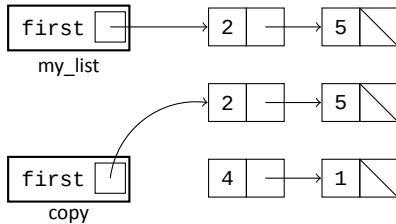
Copy assignment

Special Member Functions

Copy assignment

Special Member Functions

Copy assignment

Special Member Functions

Copy assignment
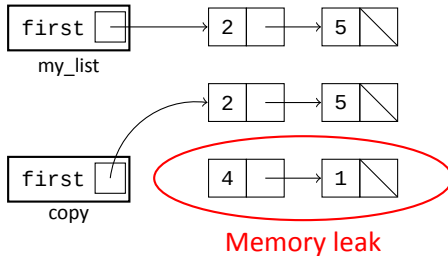
Memory leak

Special Member Functions
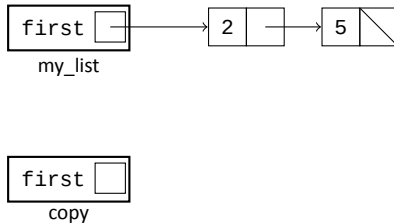
Copy assignment

Special Member Functions

Copy assignment

- The list we are copying to might already contain elements.

- If just perform a deep copy than the previous elements will be lost (i.e. we get a memory leak).

- This is a big problem. So when doing copy assignment, we have to make sure that we remove the previous elements.

Special Member Functions
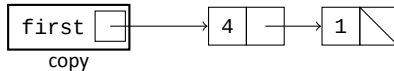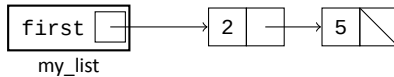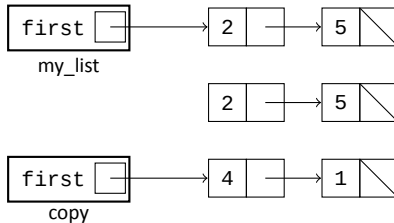
Copy assignment

Special Member Functions

Copy assignment

Special Member Functions

Copy assignment
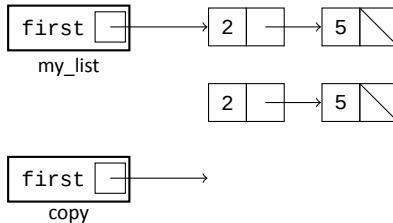
Special Member Functions

Copy assignment
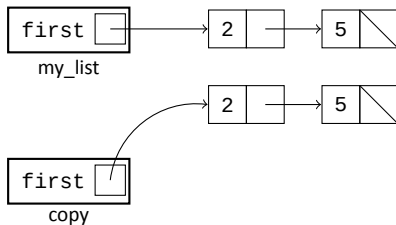
Special Member Functions

Copy assignment operator

```cpp
class List
{
public:

  List(); // default constructor
  List(List const& other); // copy constructor
  ~List(); // destructor

  List& operator=(List const& other)
  {
    // remove previous list and deep copy other
    return *this;
  }

  // ...
};
```

Special Member Functions

Copy assignment operator

- The copy assignment operator is a special operator overload that allows us to specify what happens during copy assignment.

- This operator *must* be declared inside the class.

- It must also return a reference to the list that was just assigned to.

- This is done by returning *this.

- Here we dereference this which is a pointer to the object we are inside right now.

72/82

Special Member Functions

Temporary objects


```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```
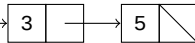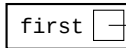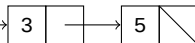
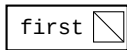first ◺

my_list

first ◺

list

Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```
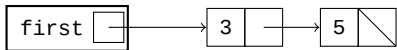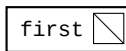
Special Member Functions

Temporary objects

```cpp
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

Special Member Functions

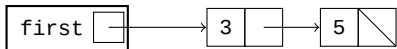Temporary objects


```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```
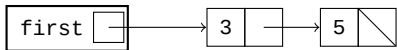
Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```


first ▨

my_list

first ▨

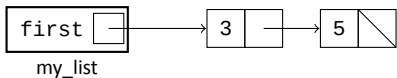list

first ▭ ──→ 3 │ ── ──→ 5 ◹

Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

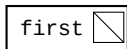first

my_list

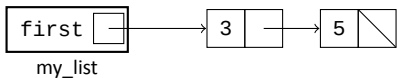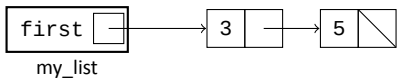first → 3 → 5

Special Member Functions

Temporary objects


```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

Special Member Functions

Temporary objects


```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

Special Member Functions

Temporary objects

my_list

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

Special Member Functions

Temporary objects

- Inside `get_list` we create a new `List` object.

- Then we return it *as a copy*.

- The object we return is temporary.

- We then copy that temporary object into `my_list`.

- There are a lot of copies being made here...

IEU LINKÖPING
UNIVERSITY

Special Member Functions

Temporary objects

first

my_list

```cpp
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

first ◹

my_list

first ◹

list

Special Member Functions

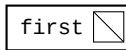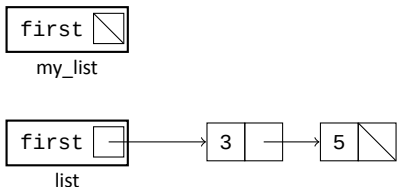Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

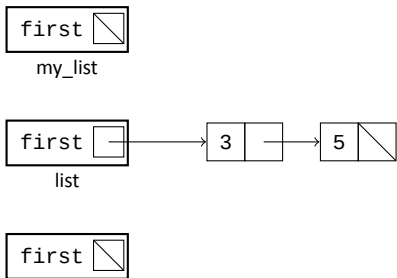first

my_list

first → 3 → 5

list

first

Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```
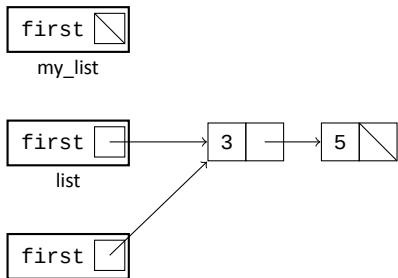
Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

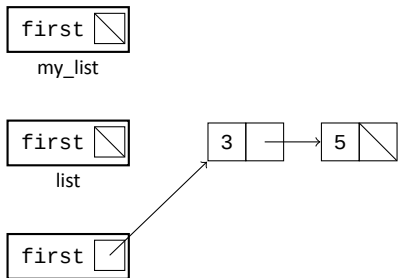# Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```
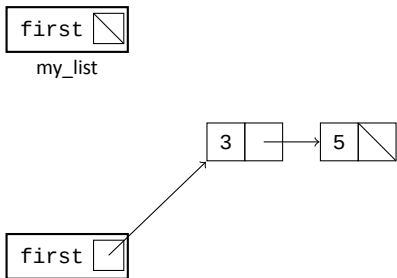
Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```
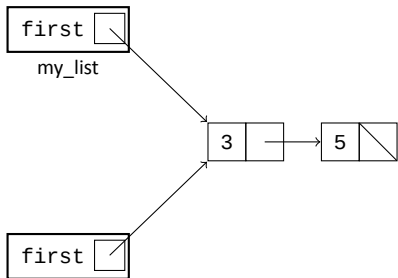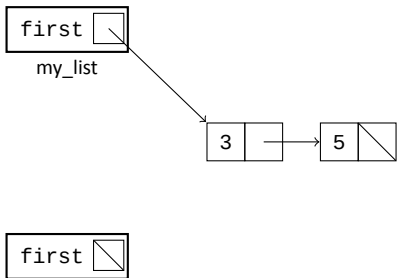
Special Member Functions

Temporary objects

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list {get_list()};
```

Special Member Functions

Temporary objects

- In C++ there is a way to reduce these copies.

- We could just steal the content from the temporary list by changing the pointers.

- Make sure to set the temporary `first` pointer to `nullptr` since othewise the destructor of the temporary list will remove the content.

- This is called *move*; we move memory from one object to another.

Special Member Functions

Temporary objects

- The compiler knows when you are trying to copy temporary objects.

- So it would be nice if we could tell the compiler how to perform these move operations.

- We can introduce something called a *Move constructor* to tell the compiler how we perform a move.

Special Member Functions

Move constructor

```cpp
class List
{
public:

  List(); // default constructor
  List(List const& other); // copy constructor

  List(List&& other)
  {
    // perform the move by shuffling the first pointers
  }

  ~List(); // destructor

  List& operator=(List const& other); // copy assignment operator

  // ...
};
```

Special Member Functions

Move assignment

```
List get_list()
{
  List list{};
  list.insert(5);
  list.insert(3);
  return list;
}

List my_list{};
my_list.insert(4);
my_list.insert(2);

my_list = get_list();
```

Special Member Functions

Move assignment operator

```cpp
class List
{
public:

  List(); // default constructor
  List(List const& other); // copy constructor
  List(List&& other); // move constructor

  ~List(); // destructor

  List& operator=(List const& other); // copy assignment operator

  List& operator=(List&& other)
  {
    // remove old content of the list
    // move content from other to this object
  }

  // ...
};
```

Special Member Functions

Special Member Functions

```cpp
class List
{
public:

  List(); // default constructor
  List(List const& other); // copy constructor
  List(List&& other); // move constructor
  ~List(); // destructor
  List& operator=(List const& other); // copy assignment operator
  List& operator=(List&& other); // move assignment operator

  // ...
};
```

Special Member Functions

Nice initialization

```
List list1 {1, 2, 3};
List list2 {4, 5};
List list3 {6, 7, 8, 9};
```

```cpp
#include <initializer_list>

class List
{
  public:
  List(std::initializer_list<int> list)
  {
    for (int i : list)
    {
      insert(i);
    }
  }
};
```

LINKÖPING UNIVERSITY

Special Member Functions

Nice initialization

- It is possible to create a special constructor that takes an arbitrary amount of values.

- These values must all share the same type.

- This is done by creating a constructor that takes `std::initializer_list` as argument.

- To go through the initializer list you must use a special type of `for`-loop, we will talk more about it later. For now, just know that it iterates through the list and places the current value inside the variable `i`.

www.liu.se