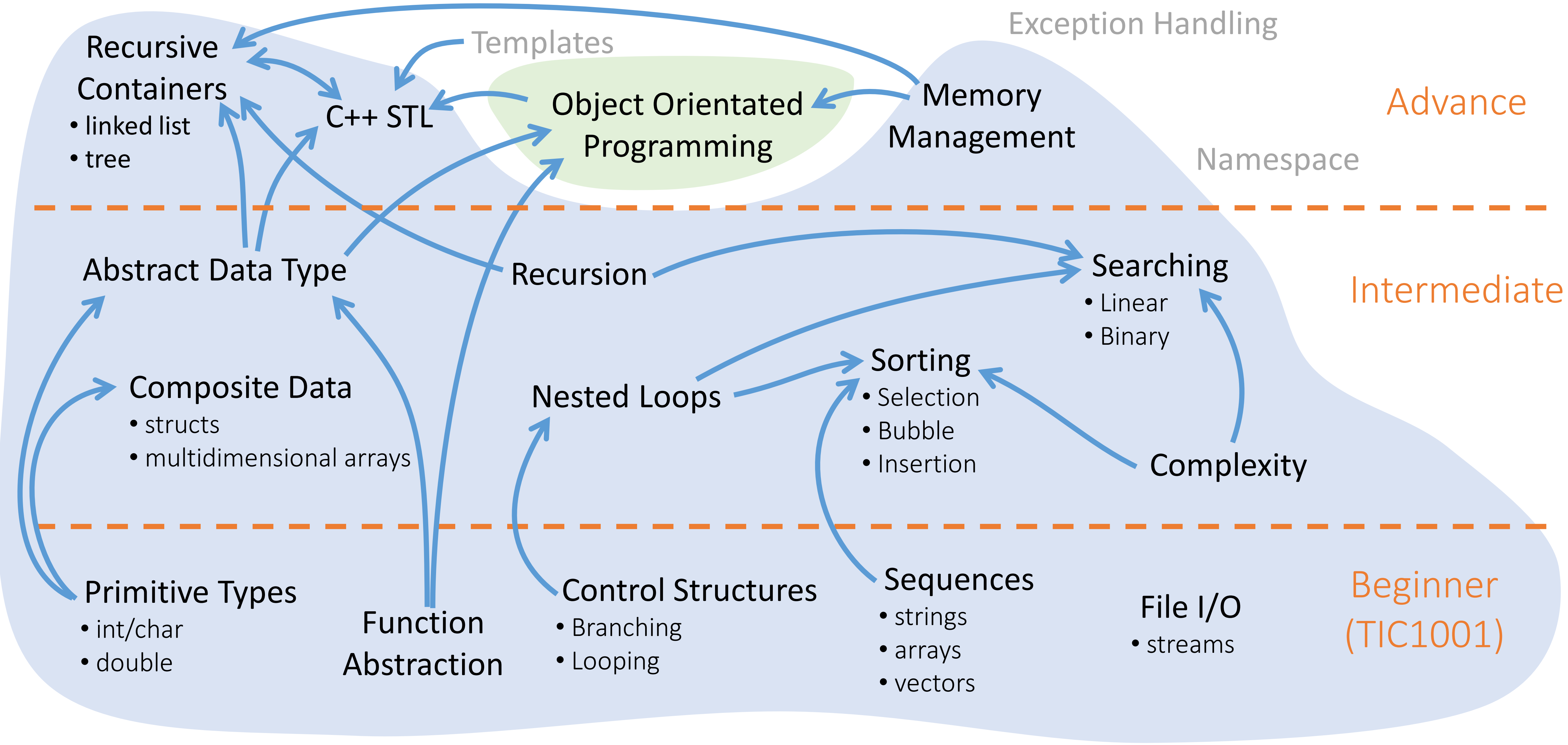
# Lecture 8
# Object Oriented Programming

TIC1002 Introduction to Computing and Programming II

# TIC1001/2 Roadmap



Recursive Containers
- linked list
- tree

Templates

C++ STL

Object Orientated Programming

Memory Management

Exception Handling

Namespace

Advance

Abstract Data Type

Recursion

Searching
- Linear
- Binary

Intermediate

Composite Data
- structs
- multidimensional arrays

Nested Loops

Sorting
- Selection
- Bubble
- Insertion

Complexity

Primitive Types
- int/char
- double

Function Abstraction

Control Structures
- Branching
- Looping

Sequences
- strings
- arrays
- vectors

File I/O
- streams

Beginner (TIC1001)

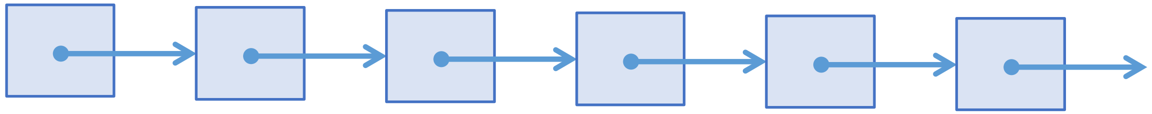
# Course Schedule

| Week | Topic(s) | | |
|---|---|---|---|
| 7 | Midterm Test | | |
| 8 | Abstract Data Type & C++ STL | | |
| 9 | Working with Collections | Problem Set 3 | |
| 10 | Object Oriented Programming | | Problem Set 4 |
| 11 | OOP: Inheritance | | |
| 12 | OOP: Polymorphism | | |
| 13 | Revision | Practical Exam 2 | |
| Reading | | | |
| Exam | Final Exam (Tue 27 Apr) | | |

# Recursive Structures

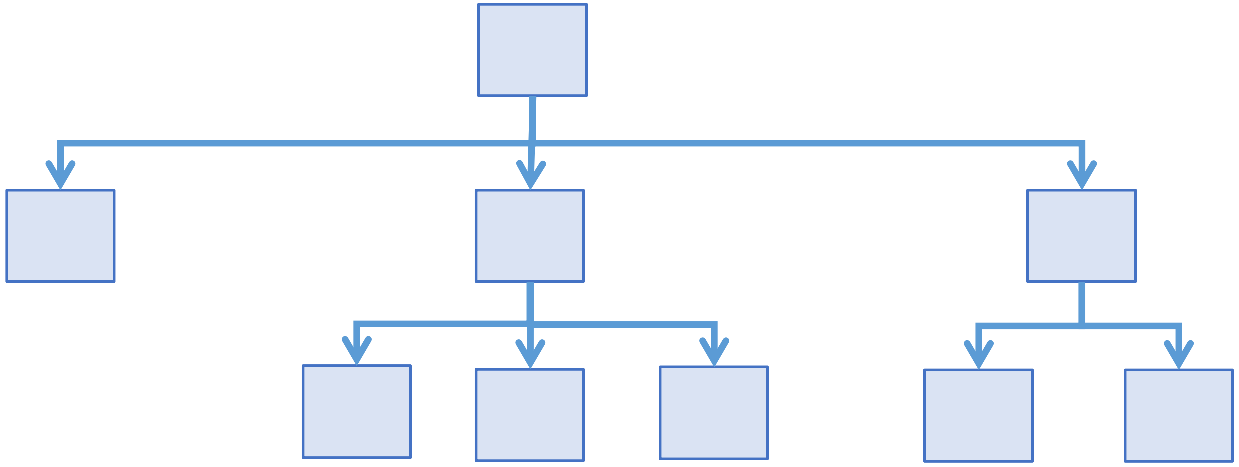
## Linked list

- Each node can only link to at most one other node



## Tree

- Each node can link to multiple children nodes



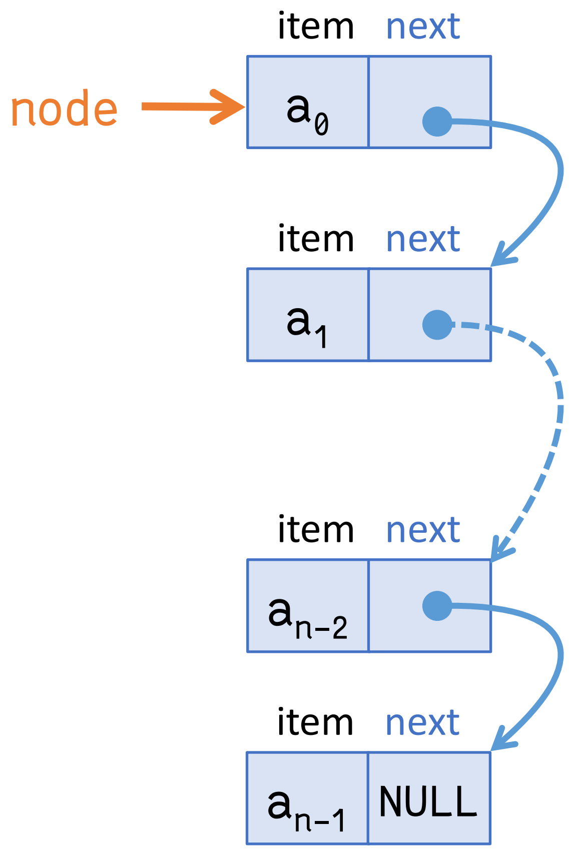
Isn’t a Linked List also a kind of tree?

# Traversing Recursive Structures

## Count the number of nodes in a Linked List

– can be done with iteration

```
int num_nodes(LinkNode<T> &node) {
  int count = 1;
  while (node.next != NULL) {
    node = *node.next;
    count += 1;
  }
  return count;
}
```



http://tiny.cc/ahlulz

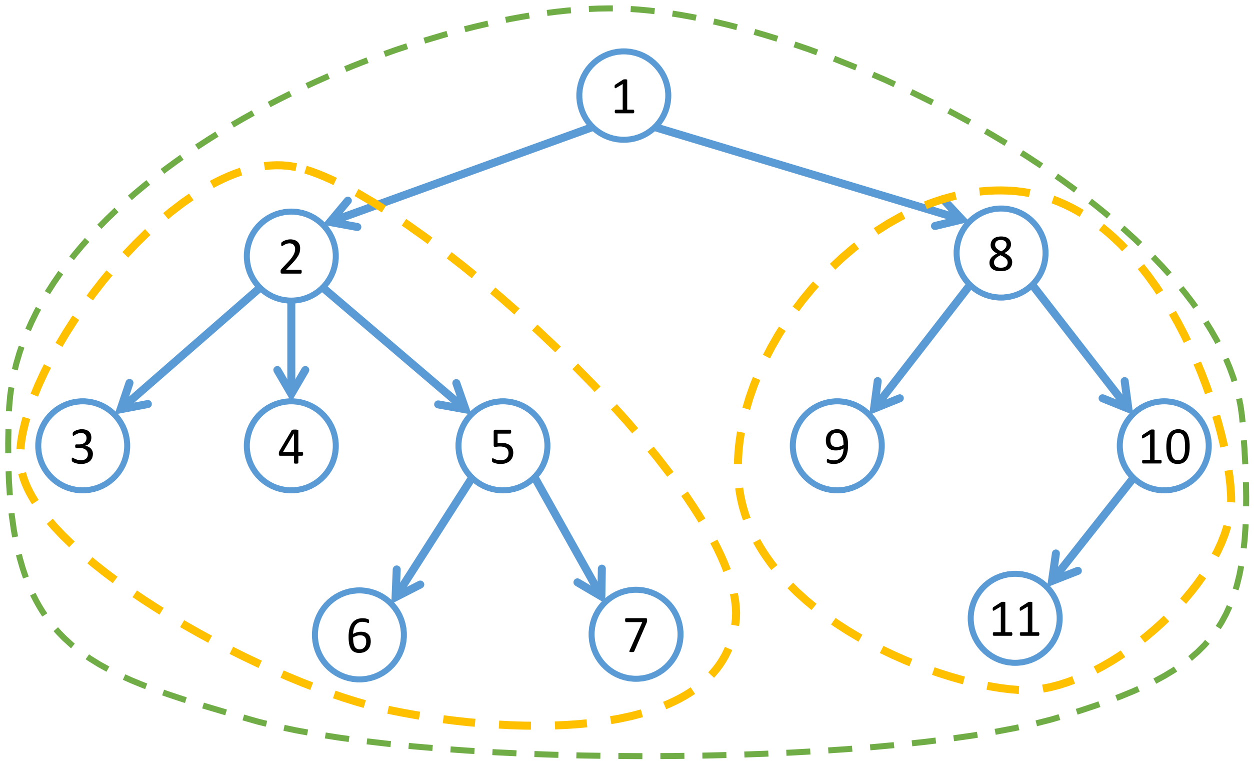
# Traversing Recursive Structures

## Count number of nodes in a tree

- Easier to use recursion

## Key Insight

- #nodes of tree = 1 + #nodes of each children



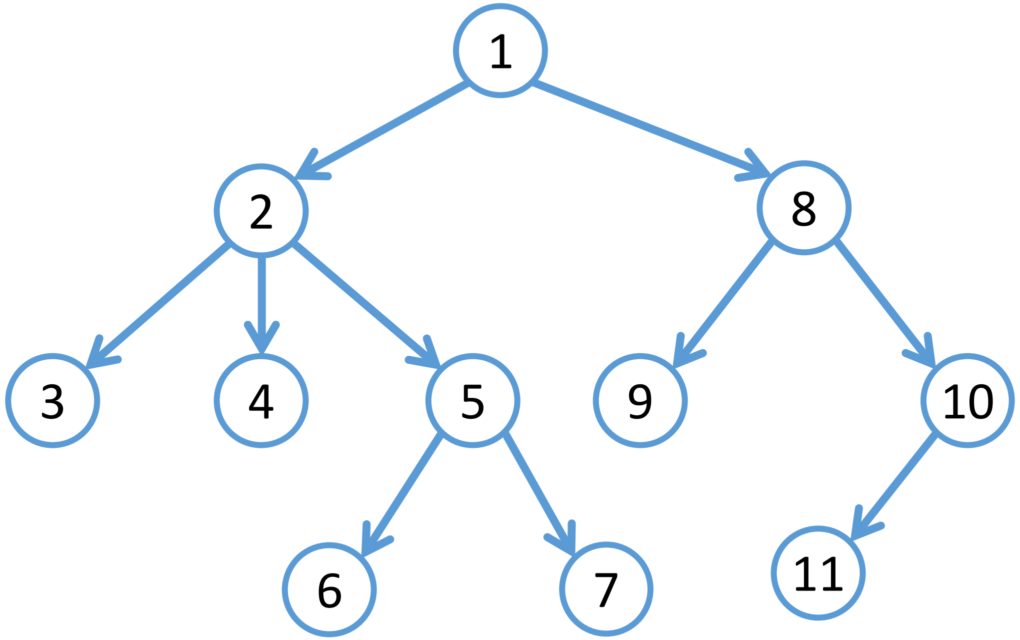
# Traversing Recursive Structures

## Count number of nodes in a tree

- Easier to use recursion

## Key Insight

- #nodes of tree = 1 + #nodes of each children

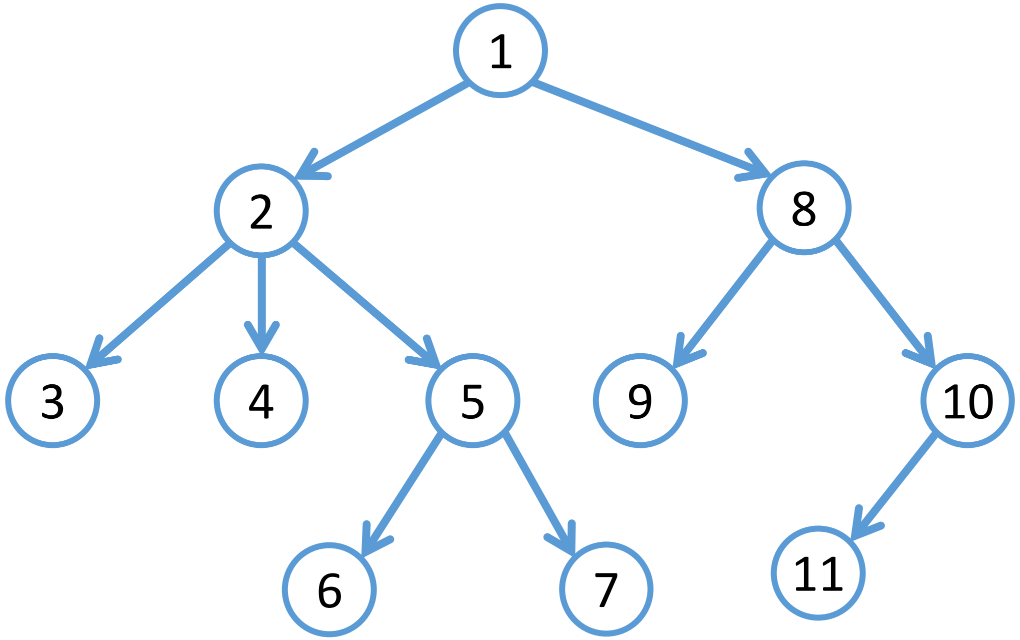


```
int num_nodes(TreeNode<T> *node) {
  count = 0;
  for (TreeNode<T> *child : node->children)
    count += num_nodes(child);
  return 1 + count;
}
```

# Number of leaves in a Tree

## Same insight

- #leaves of tree is sum of #leaves of each child
- if no child, then itself is leaf → 1



```
int num_leaves(TreeNode<T> *node) {
  if (node->children.size() == 0)
    return 1;
  int n = 0;
  for (TreeNode<T> *child : node->children) {
    n += num_leaves(child);
  return n;
}
```

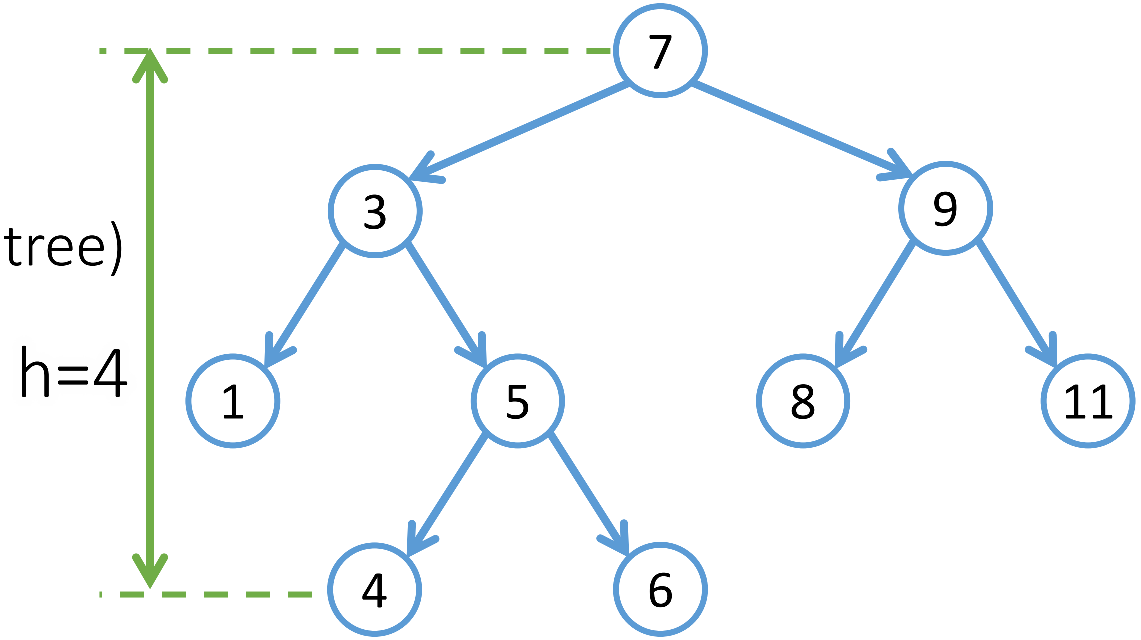
# Height of a Tree

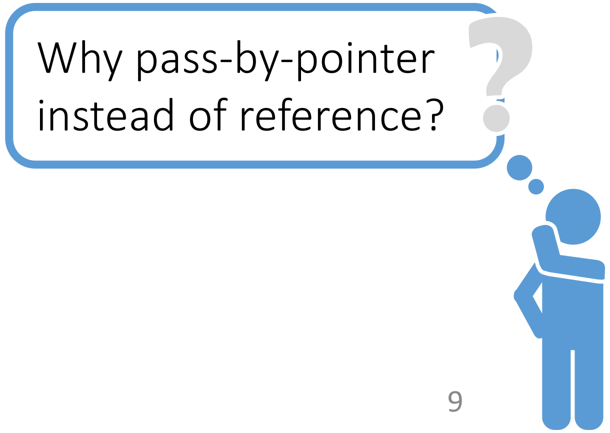
## Maximum number of levels

– Again, by recursion

height of tree = 1 + max(height of each sub-tree)

```
int height(BSTNode<T> *node) {
  if (node == NULL)
    return 0;
  else
    return 1 + max(height(node->left),
                   height(node->right));
}
```




Why pass-by-pointer instead of reference?

# Function Overloading

Functions are identified by

- Name
- Type of  the parameters

In C++, multiple versions of a function is allowed

- Same function name
- Different signature – number of parameters and type

# Function Overloading

```
int max(int a, int b) {
  if (a > b) return a;
  else return b;
}

int max(int a, int b, int c) {
  return max(max(a, b), c);
}

int max(double a, double b) {
  if (a - b > 0.00001) return a;
  else return b;
}
```

## max function is overloaded

- Which version is called depends on what arguments are supplied

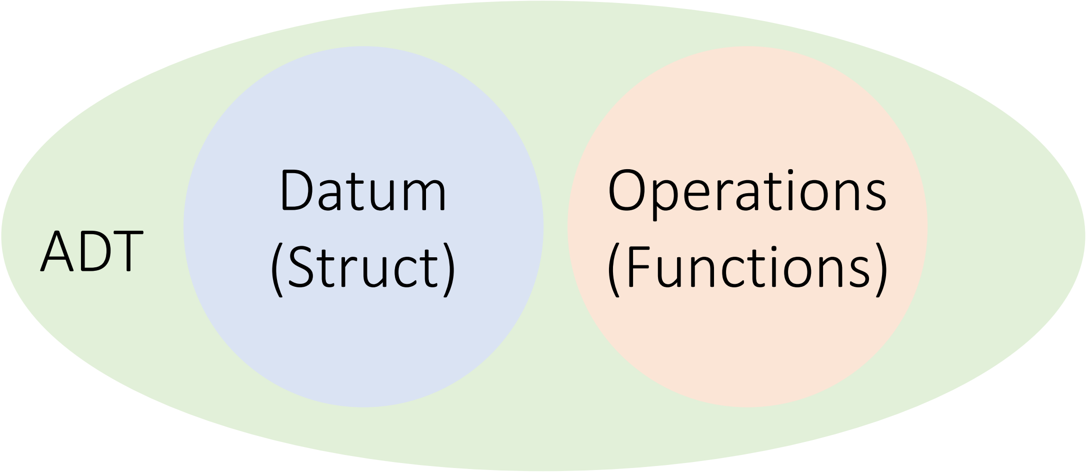
```
int x=0, y=1, z=3;
max(x, y);

max(x, y, z);

max(5.1, 5.2);
```

# Recall: Abstract Data Type

ADT = Datum + Operations



So far…

- data (struct) is loosely coupled with its operations
- not a contract, more of a gentlemen agreement

# What’s the problem?

## Scope of functions not coupled with struct

```
translate(Point &p, double x, double y) {
    p.x += x; p.y += y;
}
```

- translate function is defined in global scope.
- No control over access

## Naming conflicts if using different libraries

- What if Point is defined in polar form?
- The two are not compatible

# Illustration: Bank Account

## Basic information

- Account number: an integer
- Balance: a double

## Operations

- Withdrawal
- Deposit

# Example: Bank Account ADT

```
struct BankAcct {
    int acct_num;
    double balance;
};

BankAcct make_account(int acct_num, double bal) {
    return {acct_num, balance};  // C++11 extension syntax
}

void deposit(BankAcct &acct, double amt) {
    acct.balance += amt;
}
```

# Example: Bank Account ADT

Withdrawing cannot leave negative balance

```
bool withdraw(BankAcct &acct, double amt) {
    if (acct.balance >= amt) {
        acct.balance -= amt;
        return true;
    } else
        return false;
}
```

# Using Bank Account

```
BankAccount my_acct = make_account(1234, 1000);
deposit(my_acct, 1000);
withdraw(my_acct, 500);
withdraw(my_acct, 2000);  // this should fail

cout << my_acct.balance << endl;
```

## Malicious access

- Access to data is public

```
my_acct.balance = 10000000;
```

# Object Oriented Programming

A paradigm shift

# Programming Model

## Every language has a model

- aka programming paradigm
- how information and processes are organized
- dictates certain way of thinking or approach
- a "world view" of the language

## Popular programming paradigms

- Imperative: C, Pascal, Fortran ,etc.
- Object-Oriented: Java, C++, C#, etc.
- Functional: Scala, Haskell, Scheme, Javascript, etc.
- Declarative: Prolog

# Procedural vs Object-Oriented Language

## Program is viewed as

- A collection of functions
- Data and functions are separated
- Everything is public

- A collection of objects
  - capabilities (functions) → generally public
  - information (data) → generally private

## Pros and Cons

✓ Closely resembles execution model of hardware

✓ Less overhead

✘ Hard to understand, maintain and extend

✘ Closely resembles models and relations in real life

✘ More overhead

✓ Easy to extend and customize through inheritance and polymorphism

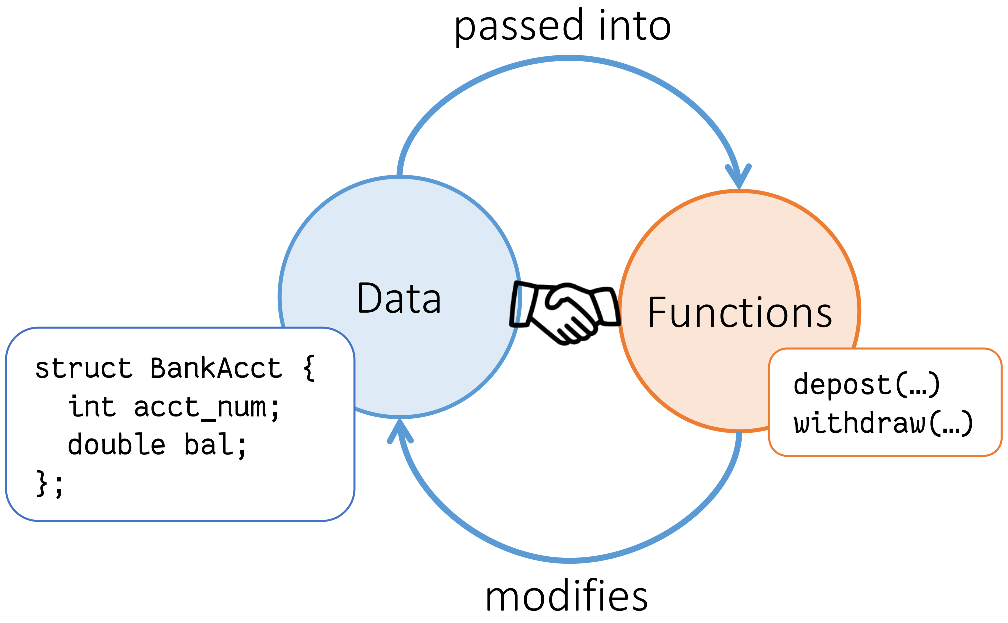
# Concepts Object Oriented Languages

1. Encapsulation
   - Group data and function together
   - Internal details hidden/abstracted
2. Inheritance
   - Extend current implementation
   - Logical relationship between entities
3. Polymorphism
   - Behaviour changes according to actual data type

# Comparing Programming Paradigms

## Procedural Model

- Data (struct) and process (functions) are separate entities

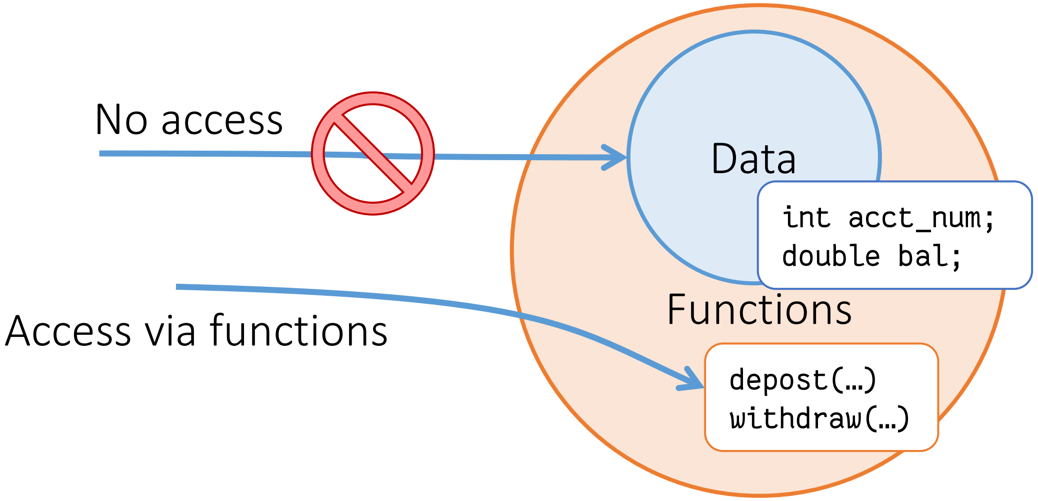


```
struct BankAcct {
  int acct_num;
  double bal;
};
```

```
depost(…)
withdraw(…)
```

## Object Oriented Model

- Data is encapsulated in functions
- No direct access to data
- Only access using exposed functions



```
int acct_num;
double bal;
```

```
depost(…)
withdraw(…)
```

# Bank Account using Classes

```
class BankAcct {

private:
    int _acct_num;
    double _balance = 0;

public:
    virtual bool withdraw(double amt) {
        if (_balance < amt) return false;
        _balance -= amt;
        return true;
    }

   virtual void deposit(double amt) {
        _balance += amt;
    }
};
```

class similar to struct

private indicates no visibility from outside

attributes or properties

default value for property

publicly accessible definitions

functions are called methods

methods can access all attributes

# Comparing

```
struct BankAcct {
    int acct_num;
    double balance;
};

BankAcct make_account(int acct_num,
                      double bal) {
    return {acct_num, balance};
}

bool withdraw(BankAcct &acct, double amt) {
    if (acct.balance < amt) return false;
    acct.balance -= amt;
    return true;
}

void deposit(BankAcct &acct, double amt) {
    acct.balance += amt;
}
```

properties/fields

accessors/getters

```
class BankAcct {

private:
    int _acct_num;
    double _balance;

public:
    virtual bool withdraw(double amt) {
        if (_balance < amt) return false;
        _balance -= amt;
        return true;
    }

    virtual void deposit(double amt) {
        _balance += amt;
    }
};
```

# Accessibility

public

- Anyone can access
- Typically for methods only

private

- Only instances of the same class can access
- Recommended for all attributes

protected

- Only instances of the same class or subclass can access
- For attributes/methods common in a family
- More on this later

# Using BankAcct Class

```
// Assume BankAcct class declared previously
int main() {
    BankAcct ba;

    ba.deposit(500);
    ba.withdrawl(1000);

    ba._balance += 1000000000;
    cout << ba._acct_num << end;
}
```

Declare an instance of BankAcct

Update state using mutators/setters

Error: property access is private

# Selectors/Accessors

- Cannot access details of BankAcct
  - Instance attributes are private
- Need to add selector/accessor methods
  - `get_acct_num` method
  - `get_balance` method

# Implement Accessors/Getters

```
class BankAcct {

private:
  int _acct_num;
  double _balance = 0;

public:
  // Mutators
  virtual bool withdraw(double amt) {
    if (_balance < amt) return false;
      _balance -= amt;
      return true;
  }

  virtual void deposit(double amt) {
    _balance += amt;
  }

  // Accessors
  virtual int get_acct_num() {
    return _acct_num;
  }

  virtual double get_balance() {
    return _balance;
  }
};
```

There are no mutators for _acc_num. Thus it is a “read-only” property, i.e. no way to change it

Question: How to initialize the object?

# Comparing

```
struct BankAcct {
    int acct_num;
    double balance;
};

BankAcct make_account(int acct_num,
                      double bal) {
    return {acct_num, balance};
}

bool withdraw(BankAcct &acct, double amt) {
    if (acct.balance < amt) return false;
    acct.balance -= amt;
    return true;
}

void deposit(BankAcct &acct, double amt) {
    acct.balance += amt;
}
```

```
class BankAcct {

private:
    int _acct_num;
    double _balance;

public:
    virtual bool withdraw(double amt) {
        if (_balance < amt) return false;
        _balance -= amt;
        return true;
    }

    virtual void deposit(double amt) {
        _balance += amt;
    }
};
```

properties/fields

constructor

accessors/getters

# Class Constructors

- Each class has one or more constructors
  - specialized method that is called automatically when an instance is created
- Default constructor
  - Takes in no inputs
  - Automatically provided if no constructor is defined
- Non-default constructor
  - Can take in parameters
  - Can have multiple different constructors

# Using BankAcct Class

```
// Assume BankAcct class declared previously
int main() {
    BankAcct ba;
```

Declare an instance of BankAcct, using the default constructor which initializes with default values.

# Implementing Constructors

```
class BankAcct {
private:
  int _acct_num;
  double _balance = 0;

public:
  BankAcct(int acct_num) {
    _acct_num = acct_num;
}

  BankAcct(int acct_num, double amt) {
    _acct_num = acct_num;
    _balance = amt;
  }
};
```

**Constructors** have the same name as the class and **no return type**

_balance is set to default value

**Constructors** are overloaded. So there are two which you can choose to call

overwrite default value with argument

# Usage

```
int main() {
  BankAcct ba1(1234);

  BankAcct ba2(1235, 1000);

  BankAcct ba3;
}
```

Use 1st constructor

Use 2nd constructor

Error: No more default constructor

## Constructors are defined

- No default constructor provided
- Specifically define a default constructor if it is useful

# BankAcct Class

## Constructor(s) ✓

```
BankAcct(int acct_num)
BankAcct(int acct_num, double amt)
```

## Accessors ✓

```
virtual int get_acct_num()
virtual double get_balance()
```

## Mutators ✓

```
virtual void deposit(double amt)
virtual bool withdraw(double amt)
```

# What is an instance

https://goo.gl/aRAJYh

## Instances are separate entities

```
BankAcct ba1(1234, 1000);
BankAcct ba2(4321, 500);

ba1.withdraw(100);
cout << ba1.get_balance() << endl;
cout << ba2.get_balance() << endl;
```

# Passing Objects

## Objects are passed by value

- similar to structs

```
void transfer(BankAcct from, BankAcct to, double amt) {
    from.withdraw(amt);
    to.deposit(amt);
}
```

- What is wrong with this code?

# Passing Objects

## The right way

```
void transfer(BankAcct &from, BankAcct &to, double amt) {
    from.withdraw(amt);
    to.deposit(amt);
}
```

– Recommended to pass all objects by reference

– Word of caution: any modifications to parameter will affect the actual object

– There is still a logic error here

# Passing Objects

## Logically correct way

```
void transfer(BankAcct &from, BankAcct &to, double amt) {
    if (from.withdraw(amt))
        to.deposit(amt);
}
```

– Check if sufficient balance first

# Passing Objects

- `transfer` function still feels procedural
  - Do it in the class

```
class BankAcct {
...
public:
    virtual void transfer(BankAcct &from, double amt) {
        if (from._balance >= amt) {
            from._balance -= amt;
            this._balance += amt;
        }
    }
};
```

private attribute can be accessed

whose balance?

# What is "`this`"?

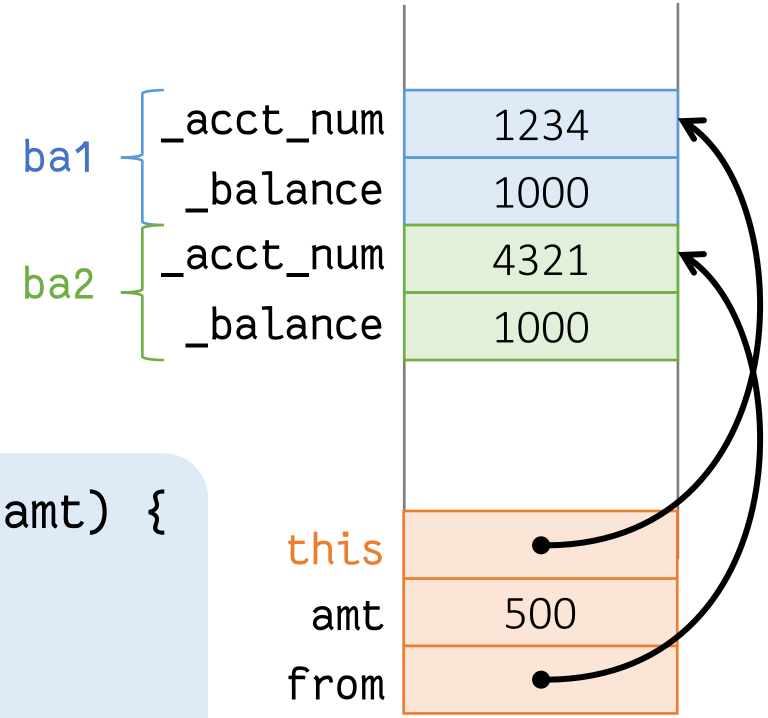
- Whenever a method is called
  - a pointer to the calling object is set automatically
  - in C++, this pointer is called "this"
  - meaning *this particular object*
- Attributes/methods can be accessed implicitly through this pointer
  - If no ambiguity, "`this`" is not needed

# What is "this"?

```
BankAcct ba1(1234, 1000);
BankAcct ba2(4321, 1000);

ba1.transfer(ba2, 500);
```

```
virtual void transfer(BankAcct &from, double amt) {
    if (from._balance >= amt)  {
        from._balance -= amt;
        this._balance += amt;
    }
}
```

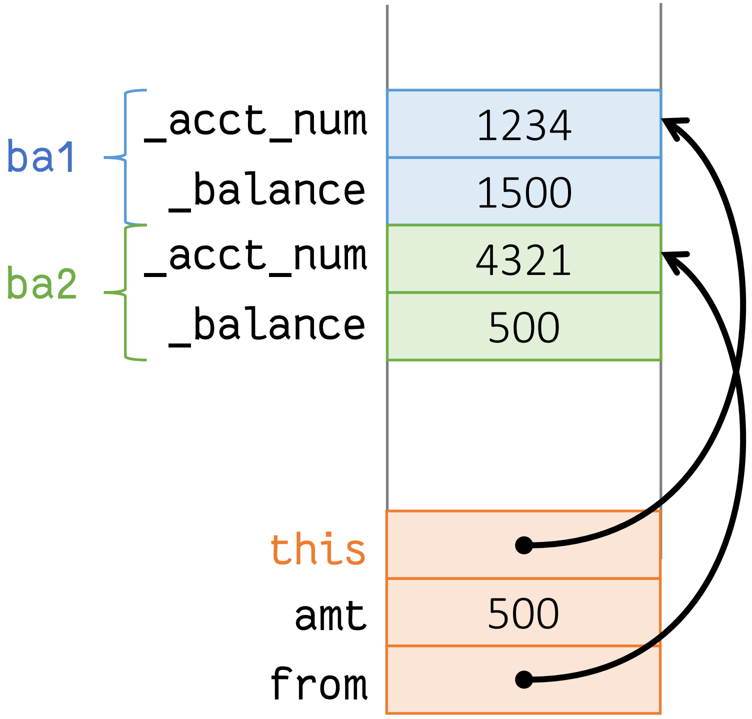


# What is "this"?

```
BankAcct ba1(1234, 1000);
BankAcct ba2(4321, 1000);

ba1.transfer(ba2, 500);
```

```
virtual void transfer(BankAcct &from,
                      double amt) {
    if (from._balance >= amt)  {
        from._balance -= amt;
        this._balance += amt;
    }
}
```



# BankAcct Class

## Constructor(s) ✓

```
BankAcct(int acct_num)
BankAcct(int acct_num, double amt)
```

## Accessors ✓

```
virtual int get_acct_num()
virtual double get_balance()
```

## Mutators ✓

```
virtual void deposit(double amt)
virtual bool withdraw(double amt)
```

## Destructors

# Destructor

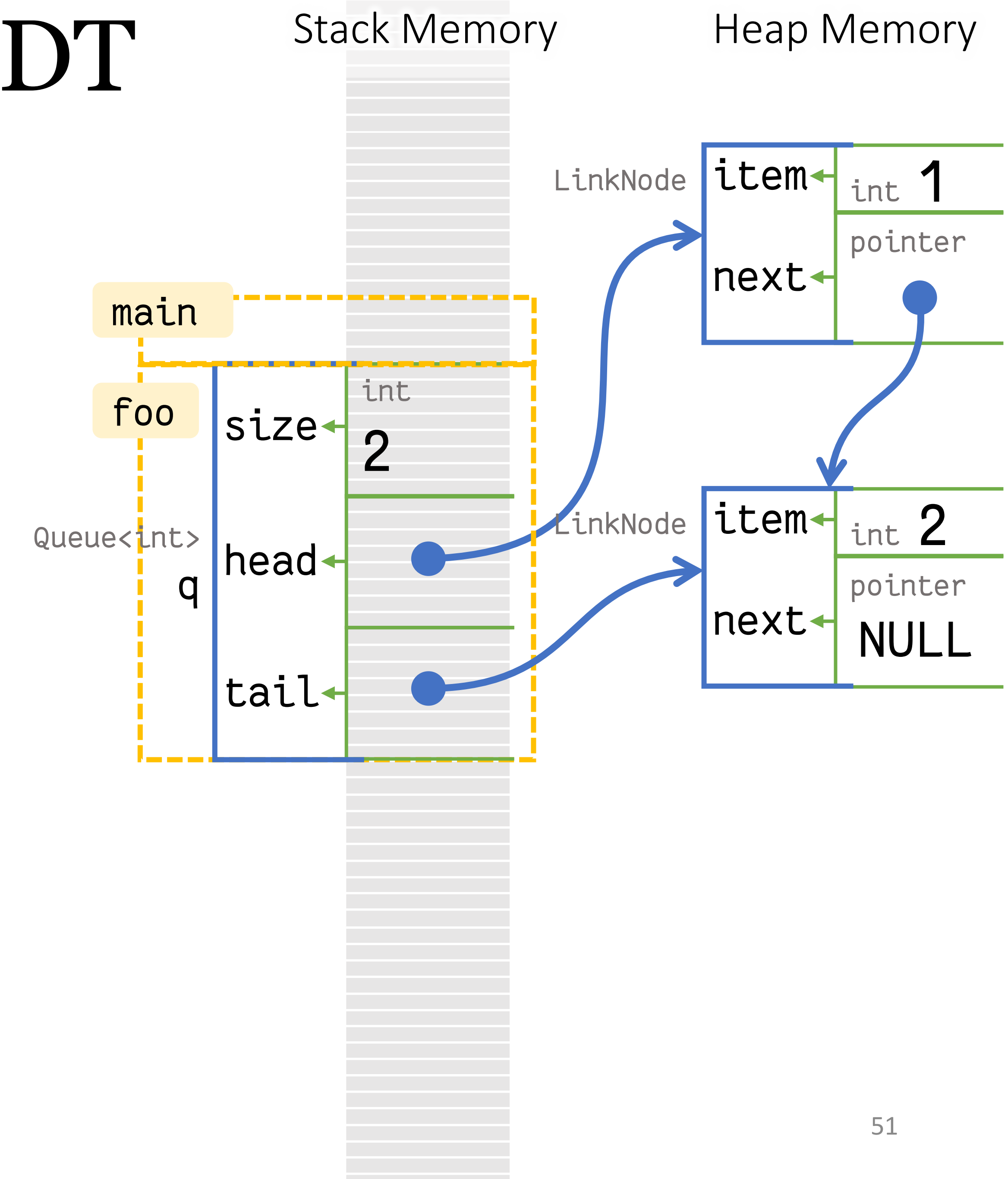
- Called automatically when
  - object of the class goes out of scope
  - object of the class gets deleted explicitly
- Destructor should be defined for classes that
  - allocated memory dynamically
  - requested for system resource ,e.g. file
- Only one destructor per class
  - Similar to constructor, prefixed by ~
  - No inputs, no return type
  - Default destructor used if none implemented

# Flashback to Queue ADT

## There is actually a bug

```
void foo() {
  Queue<int> q;
  enqueue(q, 1);
  enqueue(q, 2);
}

int main() {
  foo();
  return 0;
}
```



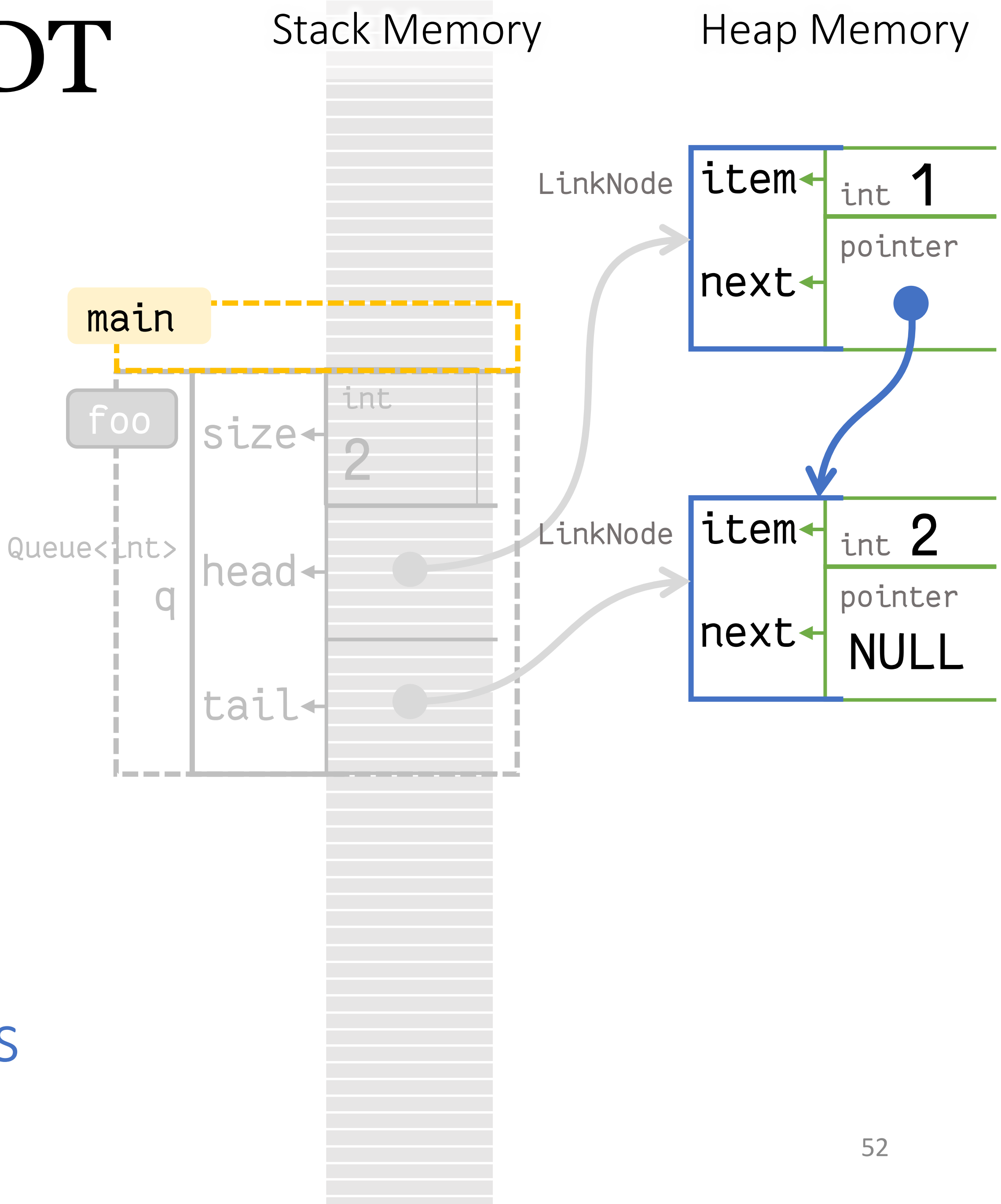
# Flashback to Queue ADT

There is actually a bug

```
void foo() {
  Queue<int> q;
  enqueue(q, 1);
  enqueue(q, 2);
}

int main() {
  foo();
  return 0;
}
```

Memory leaked when foo returns



# Life of an Object

## Allocation (Birth)

- Object declaration or new is used on a class
- Memory is allocated for the object
- Constructor is called (based on the parameters)

## Alive

- After constructor ends
- Object ready to be used

## Deallocation (Death)

- Object is out of scope or delete is used on object pointer
- Destructor of object is called
- Memory occupied is deallocated

# Example

https://goo.gl/K2j2fh

```
void f() {
  Test a(999);                              B
  cout << "End of f()" << endl;
}

int main() {
  Test b(123), *p;

  if (true) {
    Test c(456);                          A
  }

  f();

  p = new Test(789);                        C
  delete p;

  cout << "End of main" << endl;
}
```

```
class Test {
  private:
    int _id;
  public:
    Test(int i) {
      _id = i;
      cout << _id << " alive!\n";
    }
    ~Test() {
      cout << _id << " died!\n";
    }
};
```

```
Output:
123 alive!
456 alive!  ⎫
            ⎬  A
456 died!   ⎭
999 alive!  ⎫
End of f()  ⎬  B
999 died!   ⎭
789 alive!  ⎫
            ⎬  C
789 died!   ⎭
End of main
123 died!
```

# Example: Class Queue

```
template <typename T>
class Queue {
  int size = 0;
  LinkNode<T> *head = NULL, *tail = NULL;

public:
  void enqueue(T item) {
    LinkNode<T> *n =
        new LinkNode<T>{item, NULL};
    if (head == NULL) head = n;
    else tail->next = n;  tail = n;
    size += 1;
  }

  T front() {
    return head->item;
  }
```

```
  void dequeue() {
    LinkNode<T> *n = head;
    if (head == tail) q.tail = NULL;
    head = head->next;
    delete n;
    size--;
  }

  ~Queue() {
    while (head != NULL) {
      LinkNode<T> *n = head;
      head = head->next;
      delete n;
    }
  }
};
```

# OO Paradigm != Language

## Objected-Oriented Paradigm is

- a way of organizing information and process
- a "wordview" of the programming language

## Main ideas found in other OO languages

- Classes and instances
- Attributes and methods
- Visibility

# Java

```
class BankAcct {
  private int _acc_num;
  private double _balance;

  public BankAcct(int acc_num, double bal) {
    _acc_num = acc_num;
    _balance = _bal;
  }

  public void transfer(BankAcct from, int amt) {
    from._balance -= amt;
    _balance += amt;
  }

  ...
}
```

# Python

```
class BankAcct:
  def __init__(self, acc_num, bal):
    self._acct_num = acc_num
    self._balance = bal

  def transfer(self, from, amt):
    from._balance -= amt
    self._balance += amt

  ...
```

# To be continued…

# Summary

What’s new in C++

OO programming paradigm

- Encapsulation
- Accessibility
- Classes and Instances
- Methods and Attributes

C++ support for OOP