Source code organization

- Templates are compiled twice:

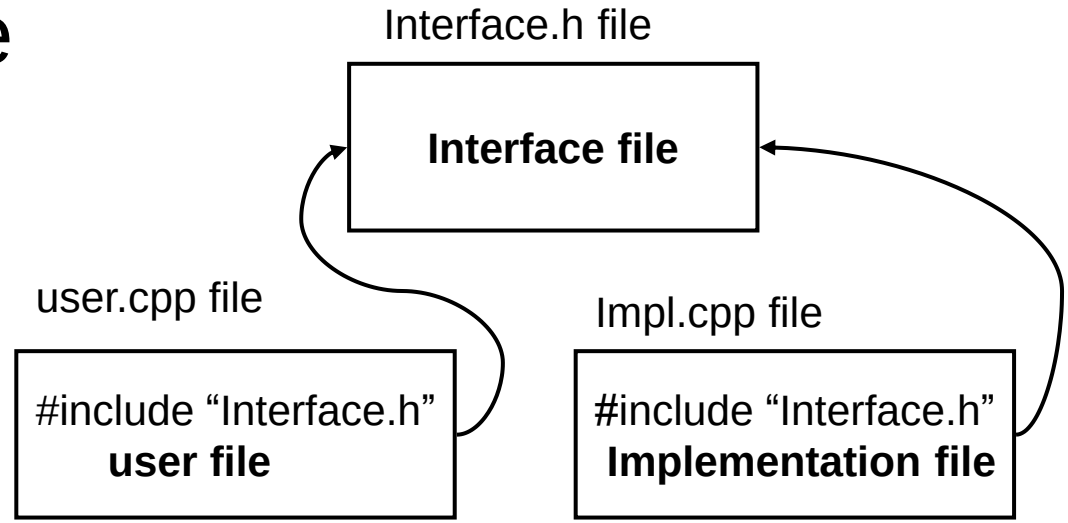
    1. At the point of definition: without instantiation, the template code itself is checked for correct syntax. Syntax errors are discovered, such as missing semicolons.

    2. At the time of instantiation: the template code is checked to ensure that all calls are valid. Invalid calls are discovered, such as unsupported function calls.

- There are several ways to organize template source code:
    - Inclusion model
    - Explicit instantiation
    - Separation model: The keyword export

Template code vs. Non-template code

- Fundamental unit of design
- Separate compilation



- Template code is a little different from ordinary code:

1. A template is not a class or a function. A template is a "pattern" that the compiler uses to generate a family of classes or functions.

2. In order for the compiler to generate the code, it must see both the template definition (not just declaration) and the specific types/whatever used to "fill in" the template.

3. Your compiler probably doesn't remember the details of one .cpp file while it is compiling another .cpp file.

Source code organization- an example

- typeof.h

```
#ifndef TYPEOF_H_INCLUDED
#define TYPEOF_H_INCLUDED

template<class T>
void print_typeof(const T&);

#endif // TYPEOF_H_INCLUDED
```

- typeof.cpp

```
#include "typeof.h"
#include <typeinfo>
#include <iostream>

using std::cout;    using std::endl;
template<typename T>
void print_typeof(const T& t)
{
    cout << typeid(t).name() << endl;
}
```

- main.cpp

```
#include "typeof.h"
#include <iostream>
using namespace std;
int main()
{
    double ice = 3.0;
    print_typeof(ice);
    cout << "Hello world!" << endl;
    return 0;
}
```

- Linker error

```
$ g++ typeof.cpp –c typeof
$ g++ main.cpp –c main
$ ls *.o
main.o     typeof.o
g++ -o typeof.o main.o
main.o: In function ‘main’
 main.cpp: undefined reference to
‘void print_typeof<double>(double const&)’
$
```

- The reason for this error is that the definition of the function template print_typeof() has not been instantiated.

Inclusion model

- Inclusion model: Include the definitions of a template in the header file that declares that template.
- typeof.h

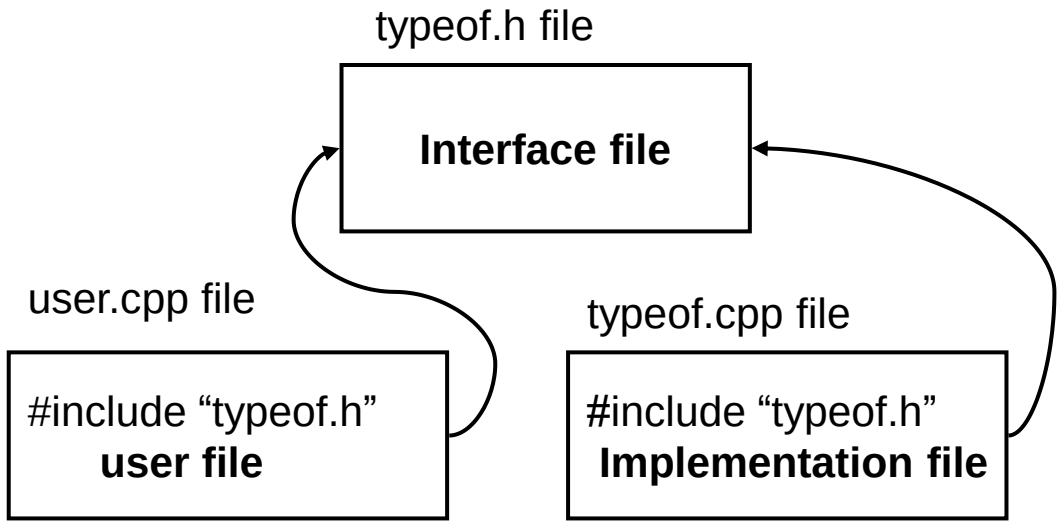
```
template<class T>
void print_typeof(const T& t)
{
    std::cout << typeid(t).name() << std::endl;
}
```

Fibonacci VariableTemplate Prog.

Explicit instantiation model

- Explicit instantiation → Explicit instantiation directive
- The explicit instantiation directive consists of the keyword template followed by the fully substitution declaration of the entity we want to instantiate.

- typeof.cpp




```
#include "typeof.h"
#include <typeinfo>
#include <iostream>

using std::cout;    using std::endl;
template<typename T>
void print_typeof(const T& t)
{
    cout << typeid(t).name() << endl;
}

// explicitly instantiate print_typeof() for type double
template void print_typeof<double>(const double &);
```

Conventional definition

Conventional definition

- Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

  - Bjarne Stroustrup

Conventional definition

• Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

- Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

Conventional definition

• Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

- Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

Conventional definition

• Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

- Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

Conventional definition

• Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

- Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C → Macros, structures & functions
- supports data abstraction → Classes
- supports object-oriented programming → Inheritance & Polymorphism
- supports generic programming → Templates

Conventional definition

• Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

- Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C → Macros, structures & functions
- supports data abstraction → Classes
- supports object-oriented programming → Inheritance & Polymorphism
- supports generic programming → Templates

C++ general rule:

General rule

C++ is a language not a complete system.

Conventional definition

• Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

- Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C → Macros, structures & functions
- supports data abstraction → Classes
- supports object-oriented programming → Inheritance & Polymorphism
- supports generic programming → Templates

C++ general rule:

General rule

C++ is a language not a complete system.

• C++ is a multi-paradigm/multi-style programming language.
• It's old, but still very useful definition.

Everything is an Object!

Everything should be Object-Oriented.

(Single-rooted) class hierarchy

A lot of things don't fit into class hierarchies.

Built-in data types, complex number, date, time, string, …

A clean C++ program tends to be a forest of classes rather than a single large tree.

- C++ high level ideas:

1. express concepts directly in code
   Classes

2. express relations among concepts directly in code
   Class Hierarchy, *Parameterization*

3. express independent concepts in independent code
   Multiple Class Hierarchies, Parameterization

4. compose code representing concepts freely wherever the composition makes sense
   Object-Oriented Programming, *Generic Programming*

Generic programming: A definition

- Generic programming is programming with *concepts*.

  - Alex Stepanov

- Programming using *templates* to express *algorithms* and *data structures* parameterized by *data types*, *operations*, and *polices*.

  - Bjarne Stroustrup

Alex Stepanov

Independent concepts should be independently represented and should be combined only when needed.

Object-Oriented Programming → (Virtual functions) → Run-time polymorphism

Generic Programming → (Templates) → Compile-time polymorphism

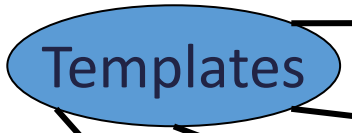
- How templates relate to Generic Programming?

  - The basis for generic programming in C++.

  - Templates provide *direct support* for generic programming.

- Generic programming is more than List<T>.

why Templates?



a template is a mechanism that allows a programmer to use types as parameters for a class, function or a variable.

Templates provide a general mechanism for compile-time programming.

Templates are the basics for generic programming.

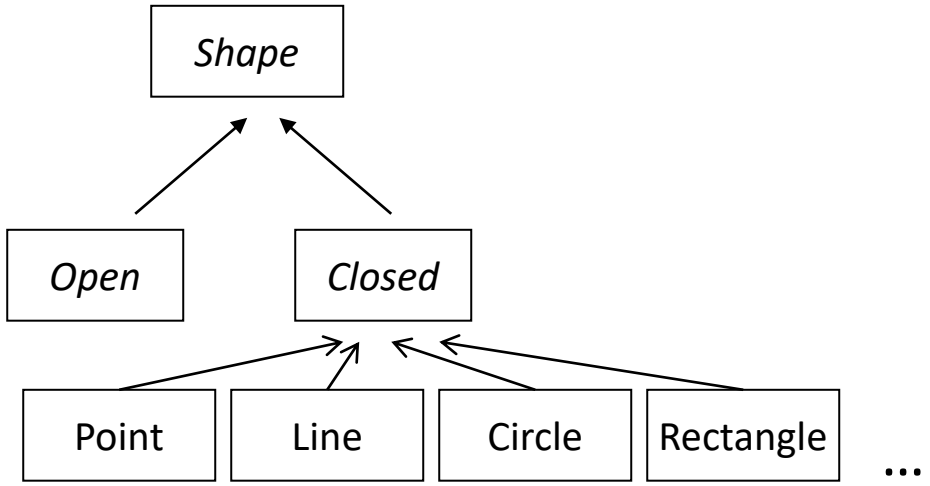
Templates provide a simple way to represent a wide range of general concepts and simple ways to combine them.

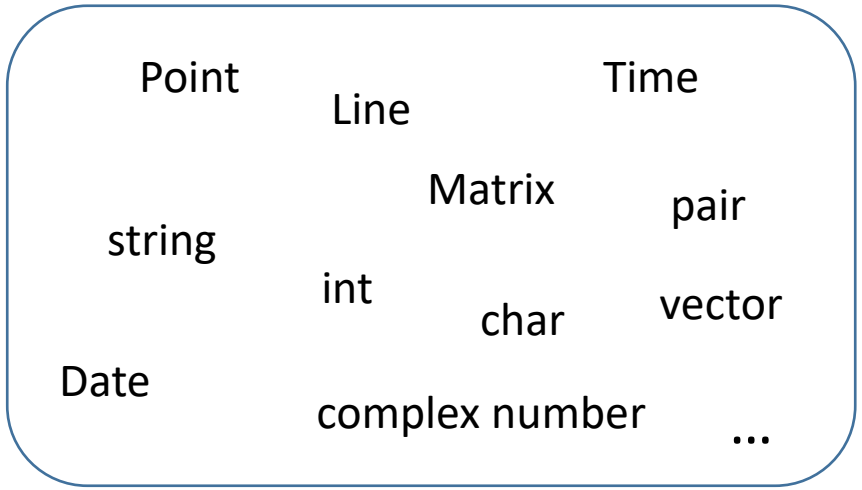
Templates and Duck typing

- C++ offers two kinds of polymorphism:

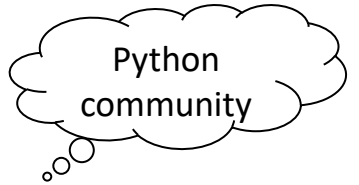
1. Run-time polymorphism ~ Virtual functions ~ Object-Oriented Programming
2. Compile-time polymorphism ~ Templates ~ Generic Programming

type relationship :class Inheritance type relationship: Similar behavior





- C++ compile-time polymorphism is based on *Duck Typing*.



- Template duck typing:

If it looks like a duck, walks like a duck, and quacks like a duck..., so it's a *duck*.

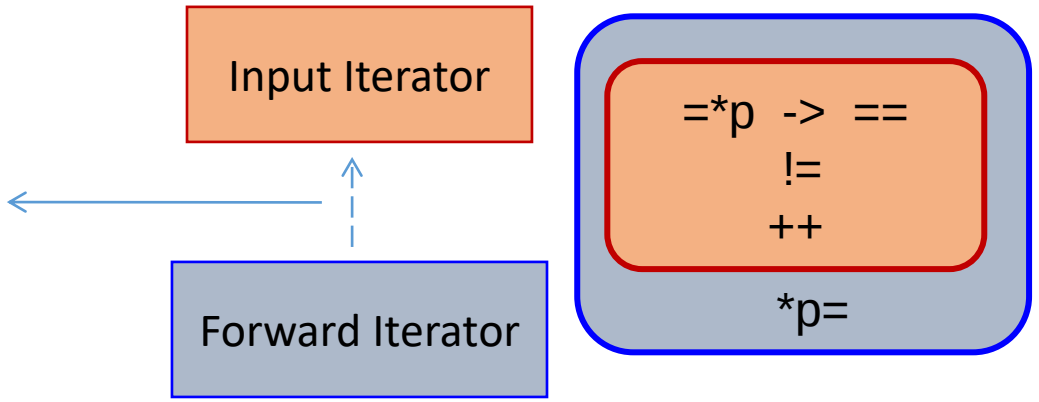
Iota

- iota is a new generic algorithm in C++11.
- More than 30 new generic algorithms were added to C++11.

```
template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value);
```

- *Requires*: T shall be convertible to ForwardIterator's value type. The expression ++val, where val has type T, shall be well formed.
- *Effects*: For each element referred to by the iterator i in the range [first,last), assigns *i = value and increments value as if by ++value.
- *Complexity*: Exactly last - first increments and assignments.

It's not inheritance!



iota and fundamental types

- Here it is a complete program that uses iota with fundamental types:

```
// iota_practice.c++
#include <numeric>
#include <vector>
#include <list>
#include <iostream>
#include <array>
int main()
{
  using namespace std;
  vector<int> vi(1000000);
  list<double> lst(1000000);
  array<char, 26> lower_case; // array is new container
  vector<long long> vll(10); // long long is a new fundamental data type
  iota(vi.begin(), vi.end(), 0); // 0, 1, 2, ..., 999999
  iota(lst.begin(), lst.end(), 0.0); // 0.0, 1.0, 2.0, … 999999.0
  iota(lower_case.begin(), lower_case.end(), 'a'); // 'a', 'b', … 'z'
  iota(vll.begin(), vll.end(), 0LL); // 0LL, 1LL, 2LL, … 9LL
  for (auto c : lower_case) cout << c << ' '; // range-based for loop
  cout << '\n';
  return 0;
}
```

- C++11: array container, long long data type, range-based for loop and auto

iota cont.

- A typical/likely implementation of iota

```
namespace std {
  template<class ForwardIterator, class TYPE_>
  void iota(ForwardIterator first, ForwardIterator last, TYPE_ t)
  {
     for (auto it = first; it != last; ++it, ++t) // prefix ++
       *it = t;
  }
}
```

Iterator requirement

Template type requirement

- Pre increment vs. Post increment

- The value of ++x, is the new (that is, incremented) value of x. The value of x++, is the old value of x.

```
y = ++x; // y = (x += 1) or ++x; y = x;
y = x++; // y = (t = x, x += 1; t) or t = x; x++; y = t;
```

- ++x means to increment x and return the new value, while x++ is to increment x and return the old value. iota uses prefix ++ increment.

iota and user-defined types

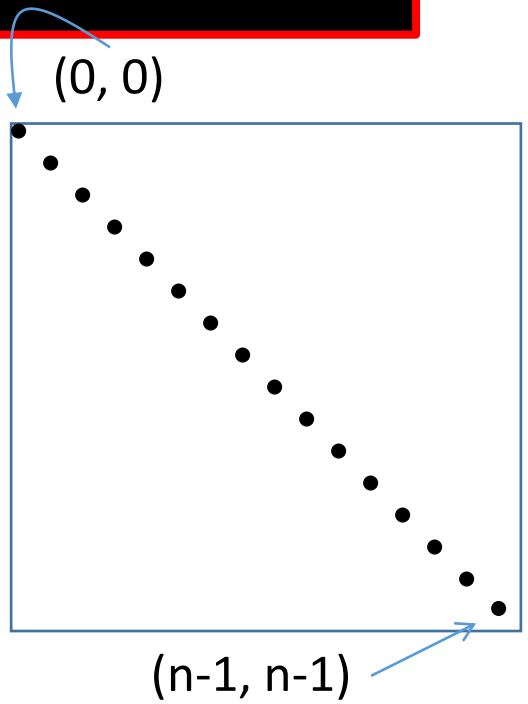
- Rational numbers, 2D points and SE Closing prices, …

1.

```
// rational.h
class Rational { // non-normalized run-time rational number
  int numerator_, denominator_;
public:
  Rational() : numerator_{0}, denominator_{1} {} // new initialization
                                                 // syntax
  // other ctor(s), relop(s), other member functions
  Rational& operator++() { numerator_ += denominator_; return *this; }
};
```

2.

```
// point.h
class Point { // 2D point
  int x_, y_;
public:
  Point() : x_{0}, y_{0} {}
  // ctor(s), graphics-related member functions
  Point& operator++() { ++x_; ++y_; return *this; }
};
```



iota and user-defined types

3.

```
// price.h
class price_stepper { // price stepper for a typical securities exchange
  double price_;
public:
  static const double STEP; // new syntax for uniform initialization
  static const double FACE_VALUE;
  price_stepper() : price_{FACE_VALUE} {} // default ctor
  price_stepper(double price) : price_{price} {}
  operator double() const { return price_; }
  price_stepper& operator++() { price_ += STEP; return *this; }
};
```

4.

```
const double price_stepper::STEP = 0.05;
const double price_stepper::FACE_VALUE = 1000.00;
```

⋮

5.

```
// gadget.h
class FuturisticGadget { // An Unidentified futuristic gadget: year 2050
     int internal_random;
public:
    FuturisticGadget& operator++()
    {
        /* random number generation */
        return *this;
    }
};
```

iota and user-defined data types

- Complete program

```
#include <array>
#include <numeric>
#include <vector>
#include <list>
#include <iostream>
#include "point.h"
#include "rational.h"
#include "price.h"
#include "gadget.h"

int main()
{
  using namespace std;
  array<Point, 10'000> ap;
  list<Rational> lr(100'000);
  vector<price_stepper> p_list(100'000);
  iota(vp.begin(), vp.end(), Point()); // [point(0, 0), point(9999, 9999)]
  iota(lr.begin(), lr.end(), Rational()); // [
  iota(lower_case.begin(), lower_case.end(), 'a'); // 'a', 'b', .. 'z'
  iota(vll.begin(), vll.end(), 0LL); // 0LL, 1LL, 2LL, 9LL

  return 0;
}
```

Contemporary C++: Learning Modern C++ in a Modern Way

آلفا
ALPHA

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 16/24

Session 16. Design and Implementation of Doubly-Linked List (Part I)

- Iterative and Incremental development
- Linked list as node-based data structure
- Singly linked lists vs. Doubly-linked lists
- List representation: Node, Link, Satellite data and Pointers
- Linked list implementation: essential operations
- The list operations: append, insert, and remove operations
- Q&A

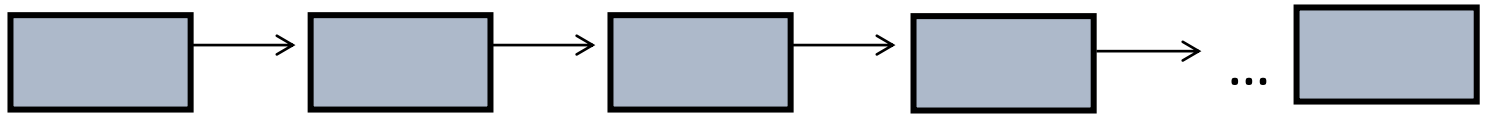
150 min (incl. Q & A)

Round 0

- The concept of Node and Satellite data
- The representation of the Linked list
- The special member functions
- List operations: append
- C++11: auto, nullptr, range-based for loop, decltype
- C++14: Automatic function return type deduction
- The overloaded output operator: <<
- Template code organization: inclusion model
- Coding Style
- Writing simple test cases

Linked list: A brief introduction

- A *linked list* is a data structure in which the objects are arranged in a linear order.
  - Cormen et al. Introduction to Algorithms
- Linear data structure
- constant time insert and erase operations anywhere within the sequence.
- Singly-linked list

elements: 

- Doubly-linked list

elements: 

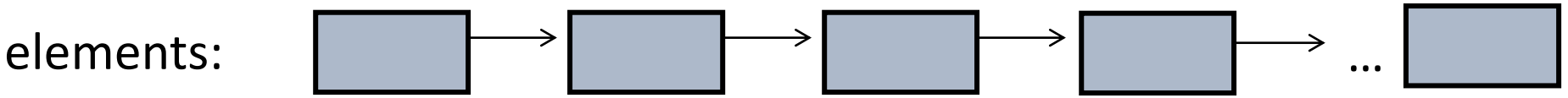
- Arrow means pointer

Linked list vs. Vector

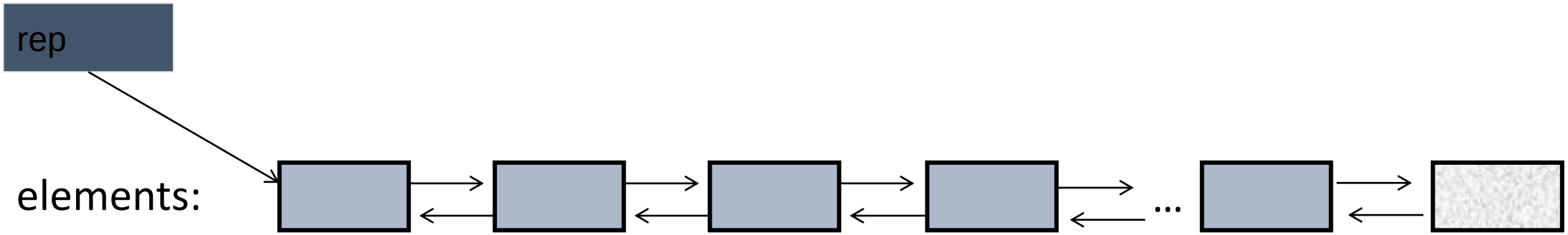
- List is a non-contiguous sequential data structure.
- Vector is a contiguous sequential data structure

Linked list: A brief introduction

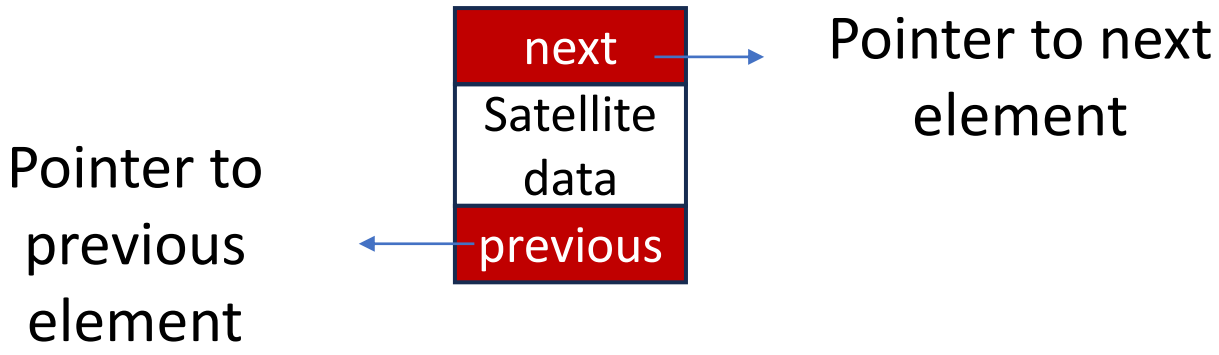
- Linear data structure



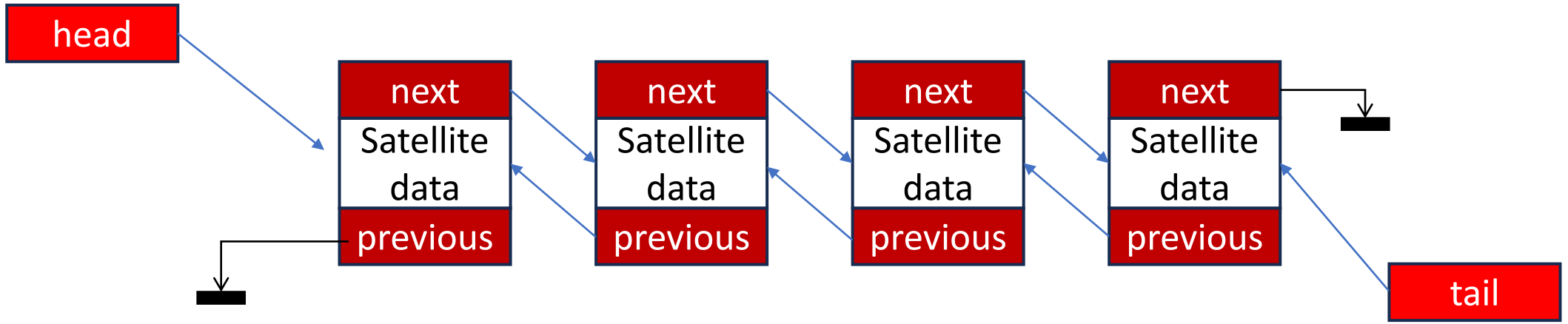
- A *list* is most likely represented by a sequence of links pointing to the elements and the number of elements.



Node


next
Satellite data
previous
Pointer to next element
Pointer to previous element

linked list: Representation



Node-based data structure

head: a pointer to the first node

tail: a pointer to the last node

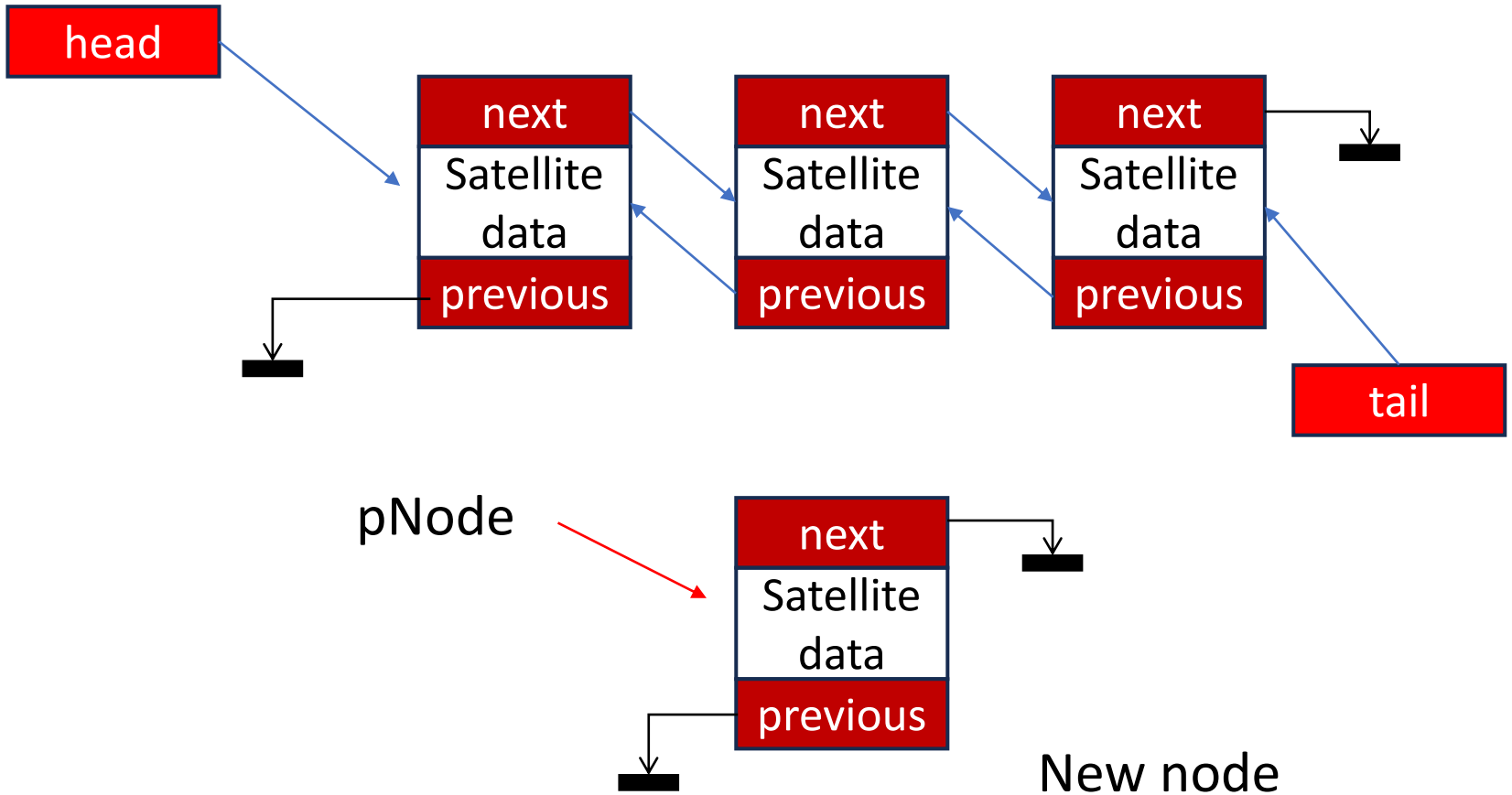
prev vs. previous

```
template<class T>
struct Node {
    T data;
    Node* next;
    Node* prev;
};
template<class T> class LinkedList {
    Node<T>* head_; // pointer to first element
    Node<T>* tail_; // pointer to last element
};
```

Linked list initial operation: append

- Append

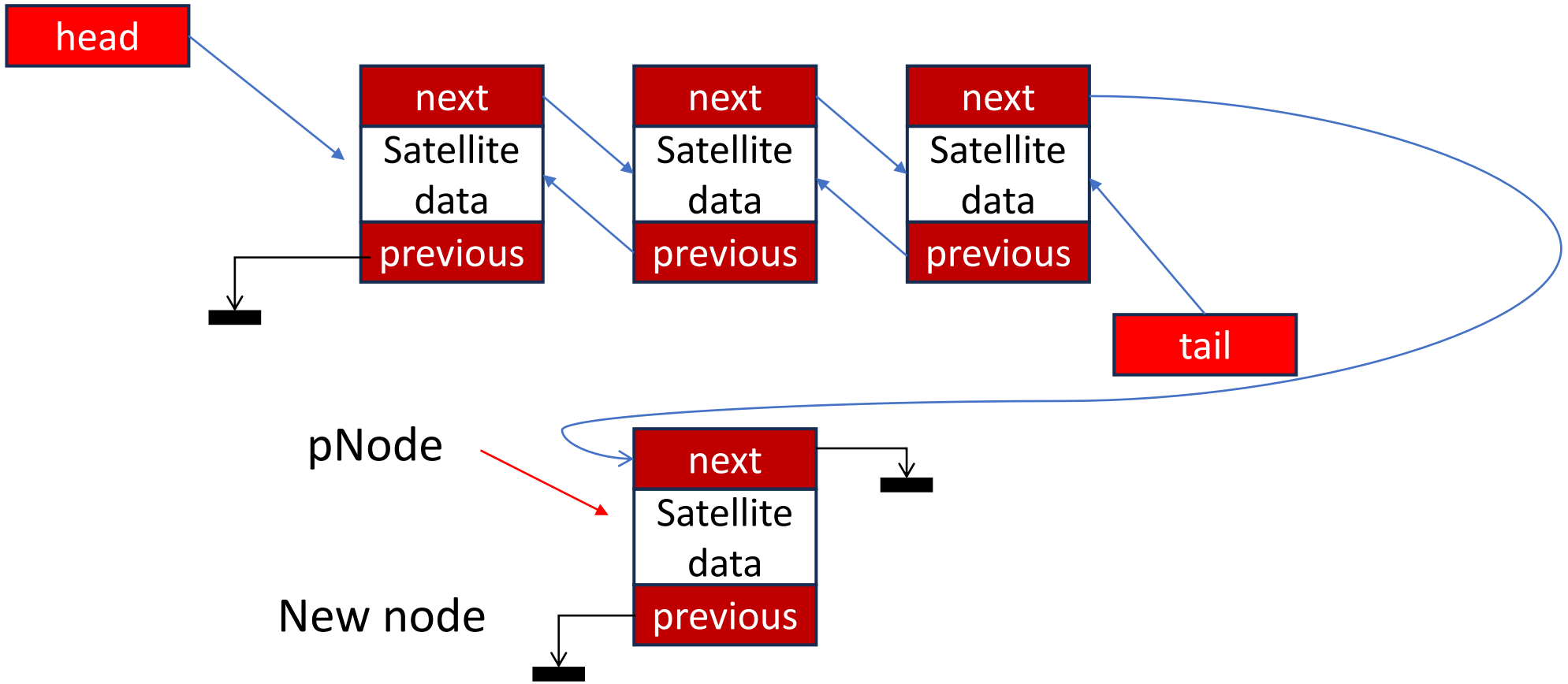
```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



Linked list initial operation: append

- Append

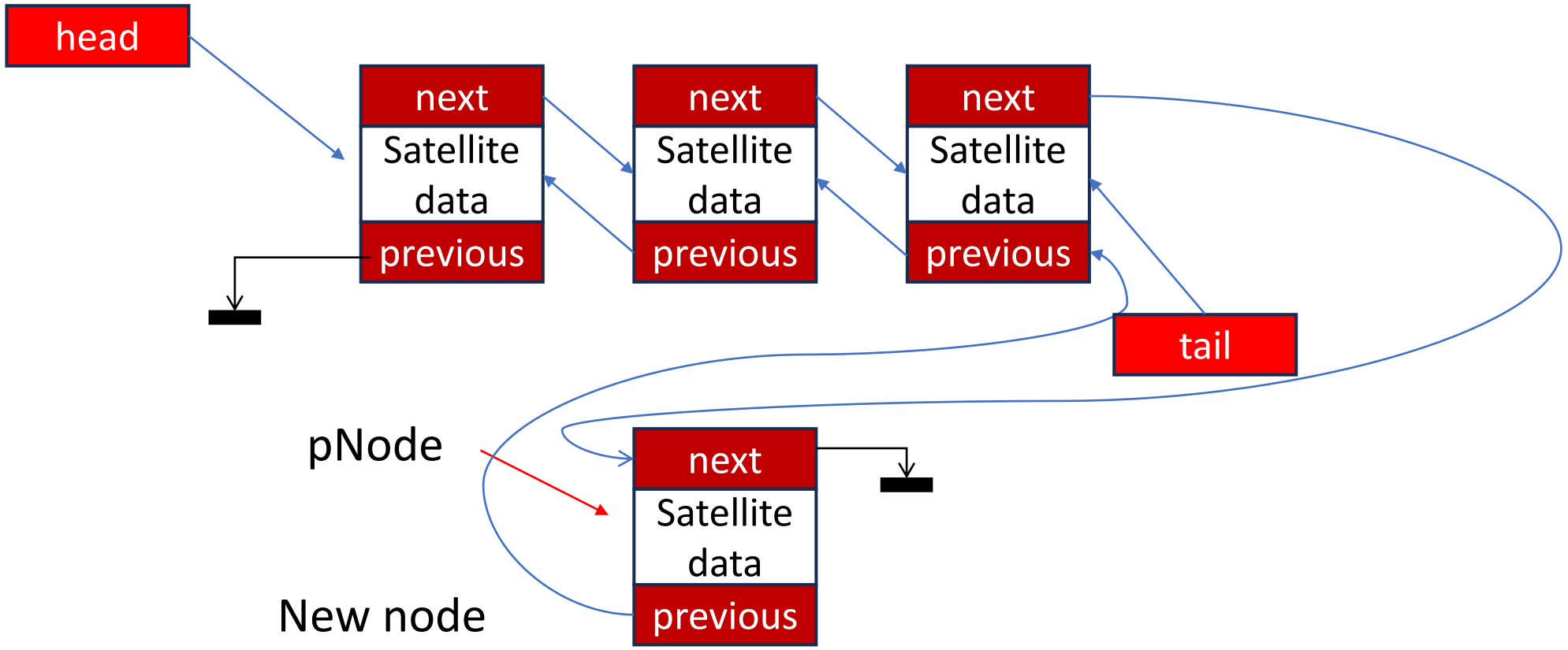
```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



Linked list initial operation: append

- Append

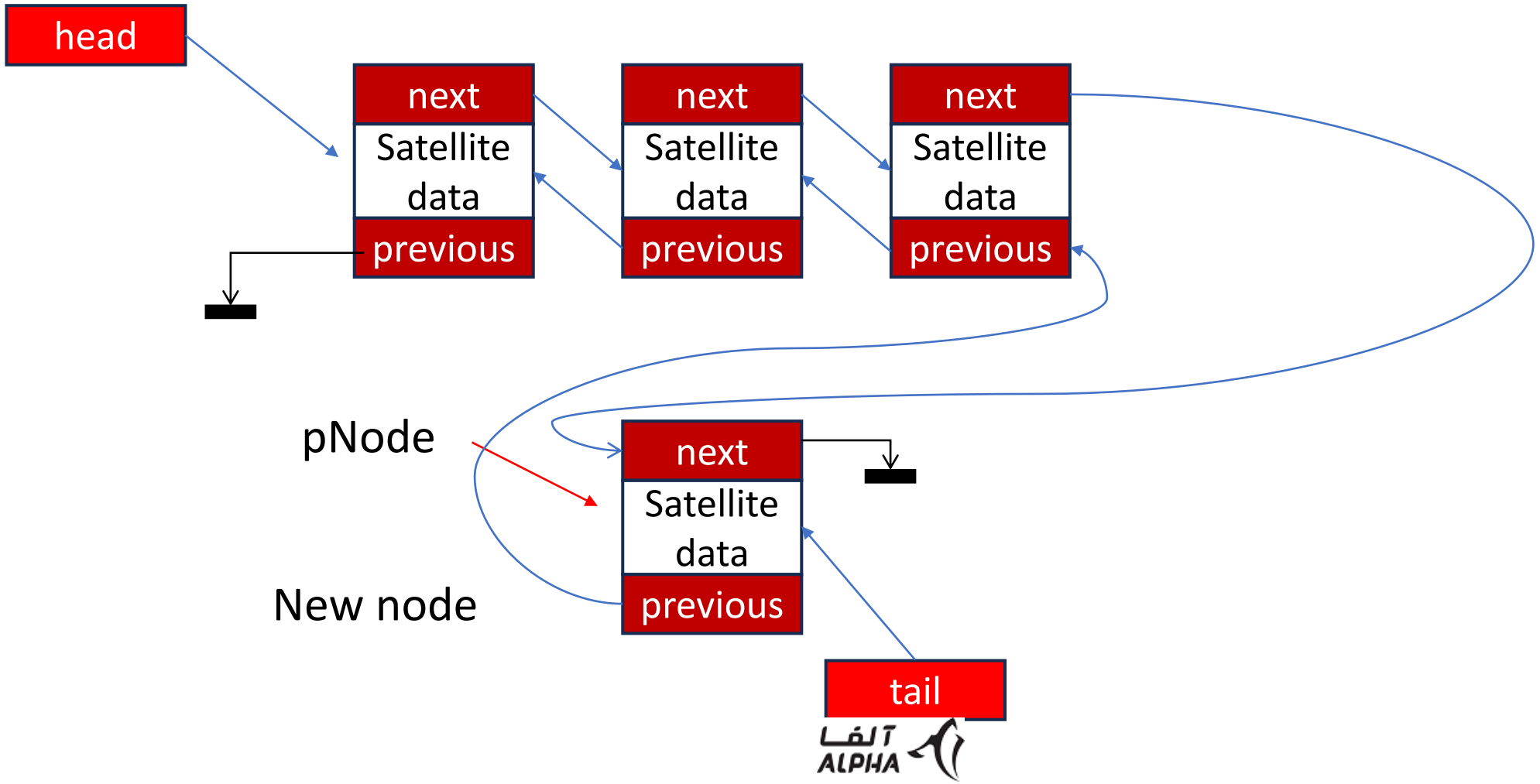
```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



Linked list initial operation: append

- Append

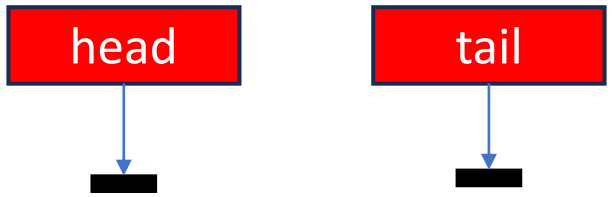
```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```

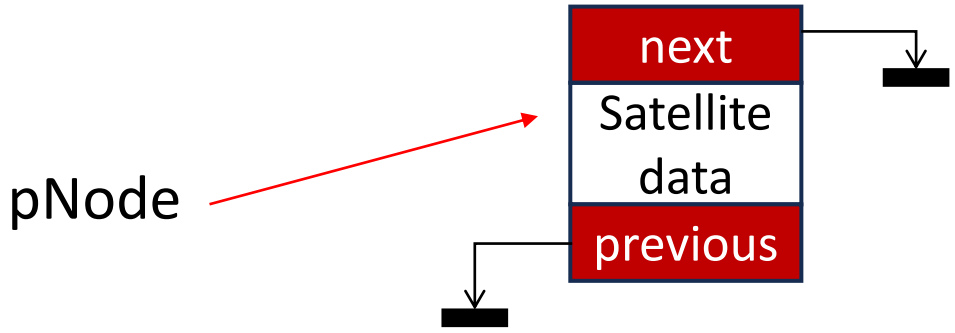


Linked list initial operation: append

- Append to the empty list

```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



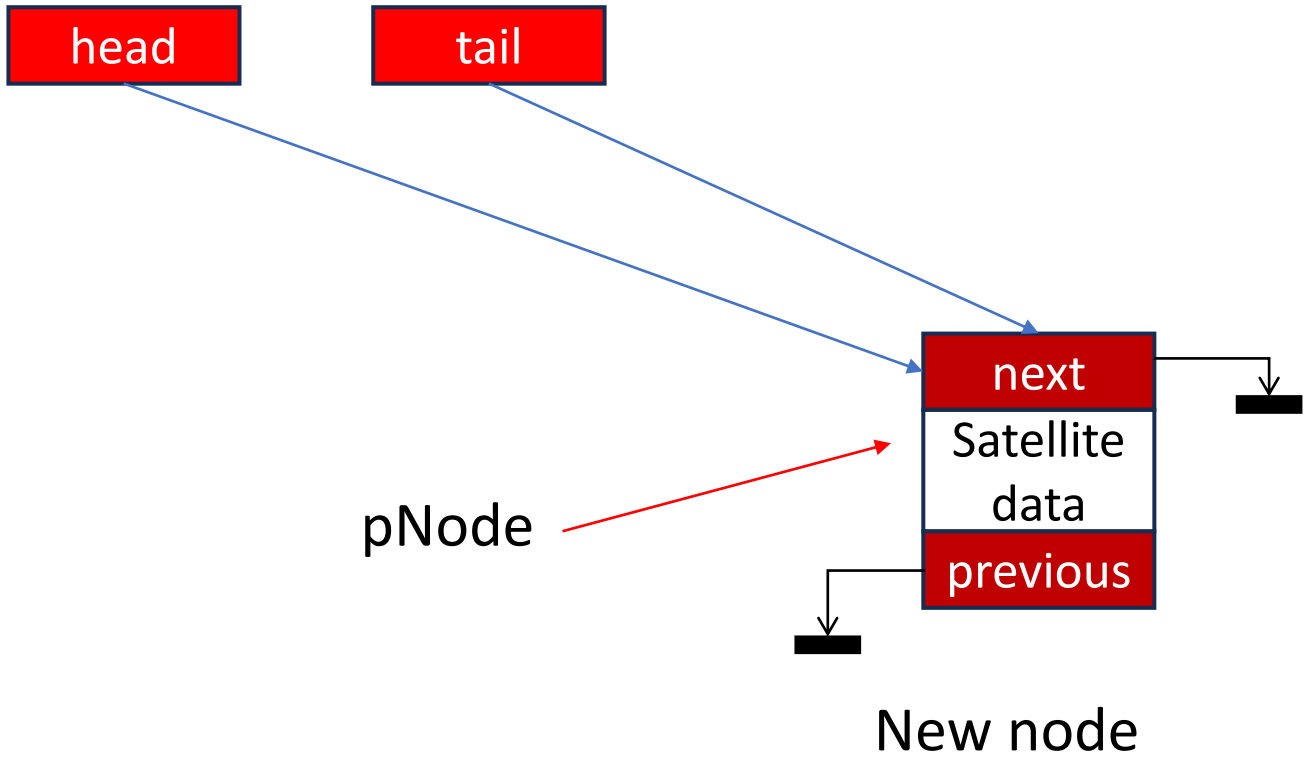


New node

Linked list initial operations: append, insert, and remove

- Append to the empty list

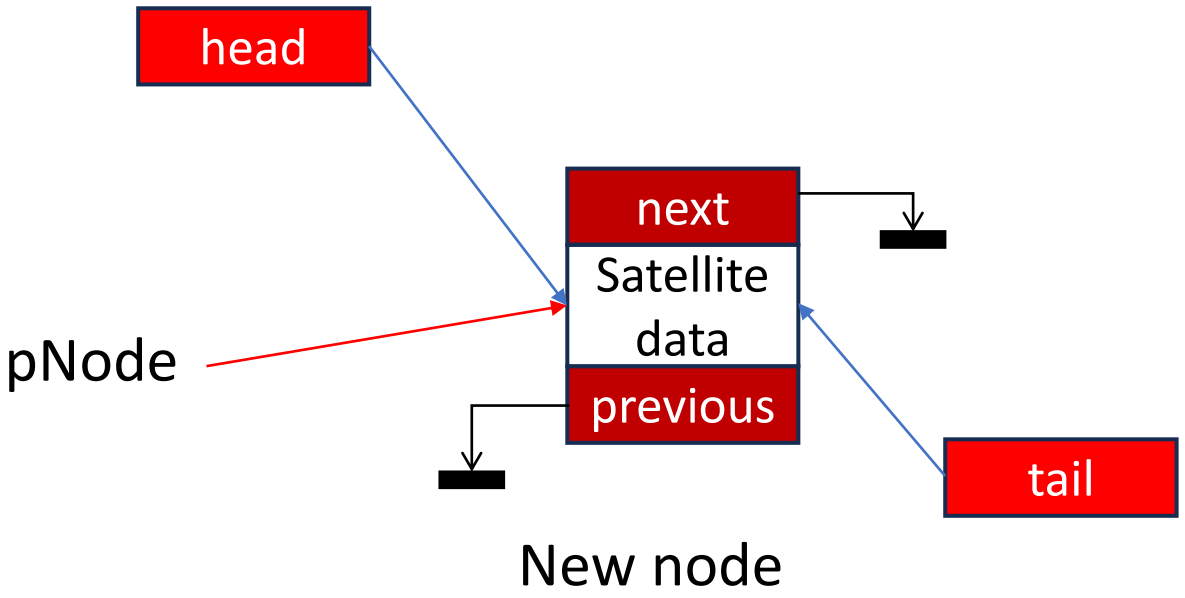
```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



Linked list initial operation: append

- Append to the empty list

```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



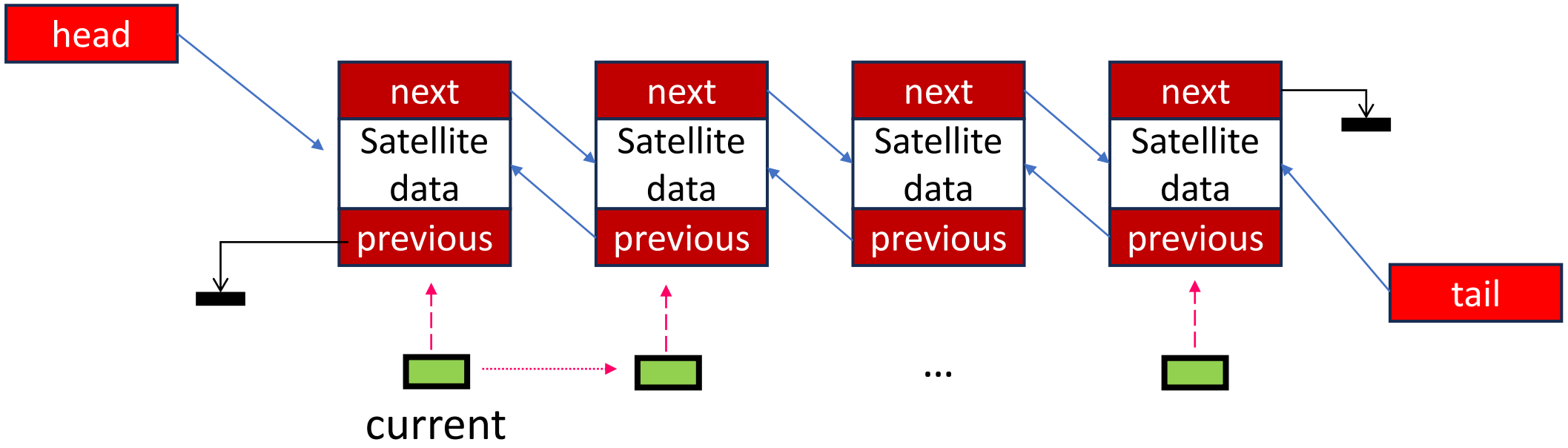
linked list: Special member functions

- Default constructor
- Copy constructor
- Copy assignment
- Move constructor
- Move assignment
- Destructor

```
template<class T> class LinkedList {
    Node<T>* head; // pointer to first element
    Node<T>* tail; // pointer to last element
public:
    LinkedList();
    LinkedList(const LinkedList&);
    LinkedList& operator=(const LinkedList&);
    LinkedList(LinkedList&&);
    LinkedList& operator=(LinkedList&&);
    ~LinkedList();
};
```

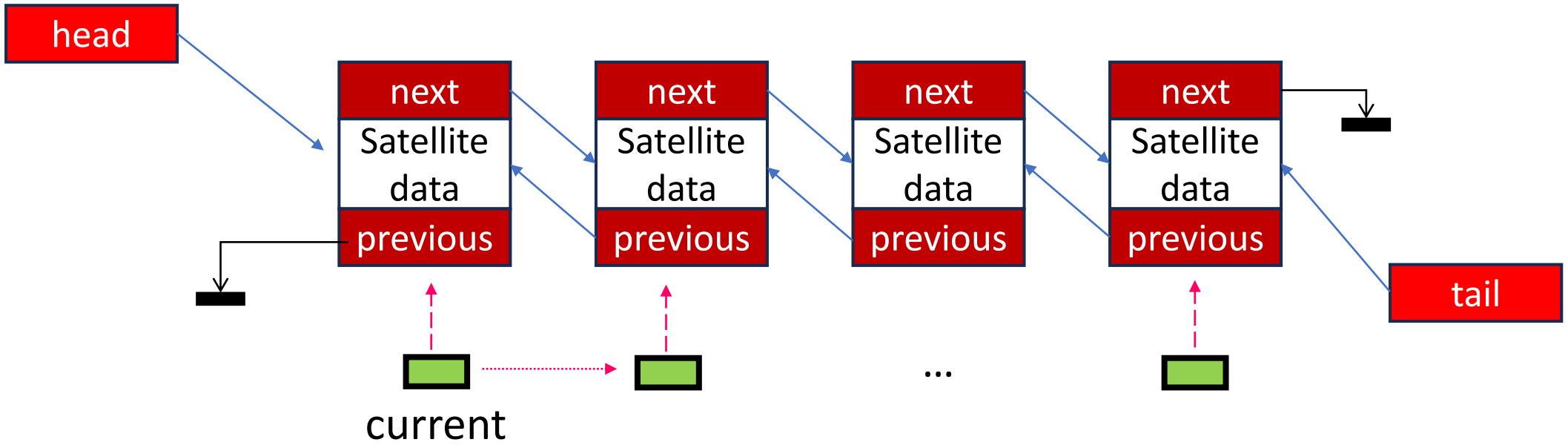
special member functions: Destructor

- Traversing the linked list and deleting nodes: for loop vs. while

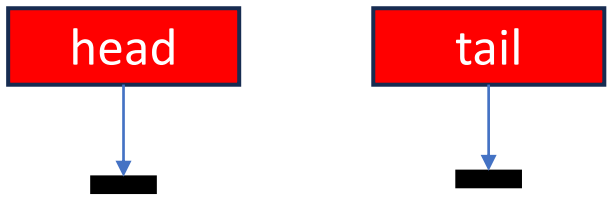


special member functions: Destructor

- Traversing the linked list and deleting nodes: for loop vs. while



- Destructed list



Traversing and deleting list nodes: for vs. while statements

```
decltype(head_) next;
for (auto current = head; current != nullptr;) {
    next = current->next;
    delete current;
    current = next;
}
head_ = tail_ = nullptr;
```

vs.

```
auto current = head_;
while (current != nullptr) {
    auto next = current->next_;
    delete current;
    current = next;
}
head_ = tail_ = nullptr;
```

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant