create a example for this c++ code T& operator*() const { return ptr->value; }

T* operator->() const { return &(ptr->value); }

Zeig doch mal

Below is a tiny “demo” that shows how the `operator*()` and `operator->()` overloads of a **`LinkedList<T>::iterator`** (or `const_iterator`) are used in practice.

```cpp
#include <iostream>
#include "linked_list.hpp"      // your LinkedList header

int main()
{
    // 1. Create a list and populate it
    util::LinkedList<int> lst{10, 20, 30};

    // 2. Iterate with the *operator (dereference)
    std::cout << "Values via operator*(): ";
    for (auto it = lst.begin(); it != lst.end(); ++it) {
        int v = *it;          // T& operator*() const
        std::cout << v << ' ';
    }
    std::cout << '\n';

    // 3. Iterate with the ->operator (member access)
    std::cout << "Values via operator->(): ";
    for (auto it = lst.begin(); it != lst.end(); ++it) {
        int* p = it.operator->();   // T* operator->() const
```

```
    std::cout << *p << ' ';    // or simply: std::cout << (*it).value;
  }
  std::cout << '\n';

  // 4. Use the iterator with a range-based for (works because of the above operators)
  std::cout << "Range based loop: ";
  for (int v : lst) {          // *v is called implicitly
    std::cout << v << ' ';
  }
  std::cout << '\n';

  return 0;
}
```

What happens under the hood

Code snippet	What it calls
`*it`	`T& operator*() const { return ptr->value; }`
`it.operator->()`	`T* operator->() const { return &(ptr->value); }`

The iterator holds a raw pointer (`Node<T>* ptr`).

- `operator*()` simply returns a reference to the node's stored value.
- `operator->()` returns a *pointer* to that value, allowing you to write `it->member` if `T` were a class type with members.

Because these overloads are defined, the iterator can be used seamlessly in range-based for loops, standard algorithms, and any other code that expects an STL-style forward iterator.