SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220418<A|D>

Replace <A|D> with this section's letter

Generic programming

CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

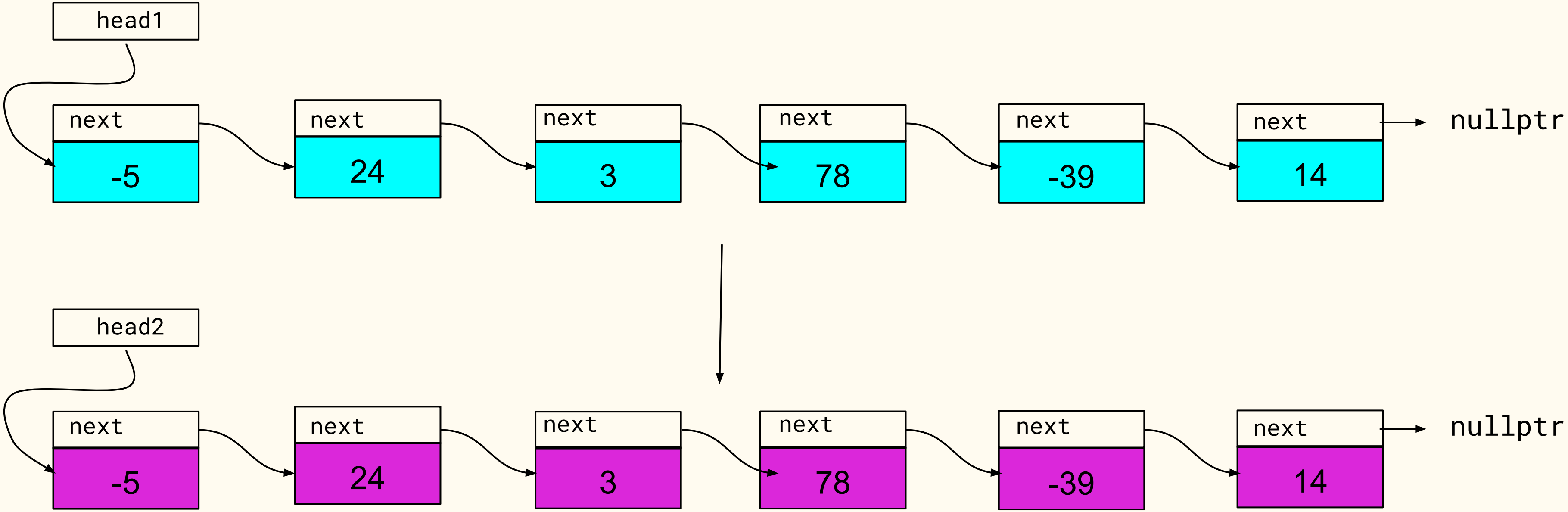
- Finishing linked lists
- Background
- Iterators

Agenda

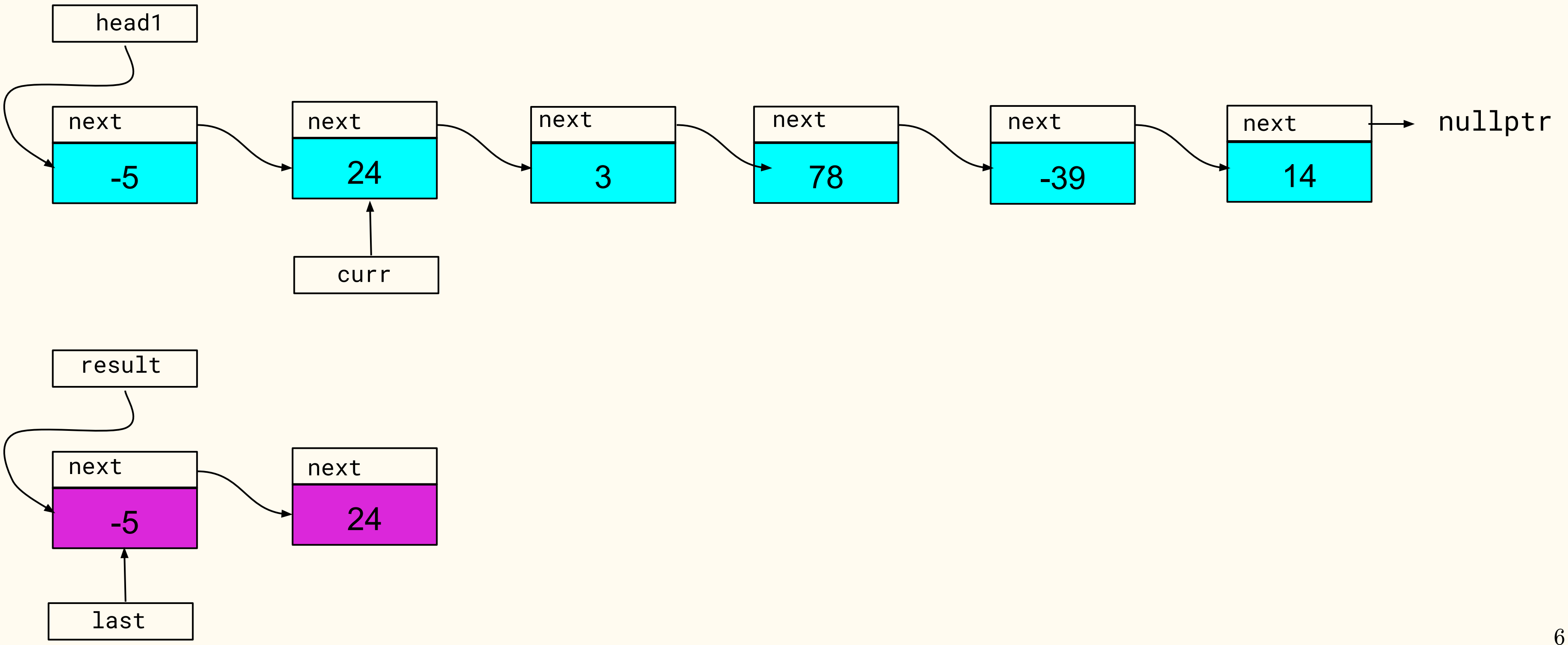
- Finishing linked lists
- Background
- Iterators
- Review of `Vector` class
- In-class problem

Duplicating a list

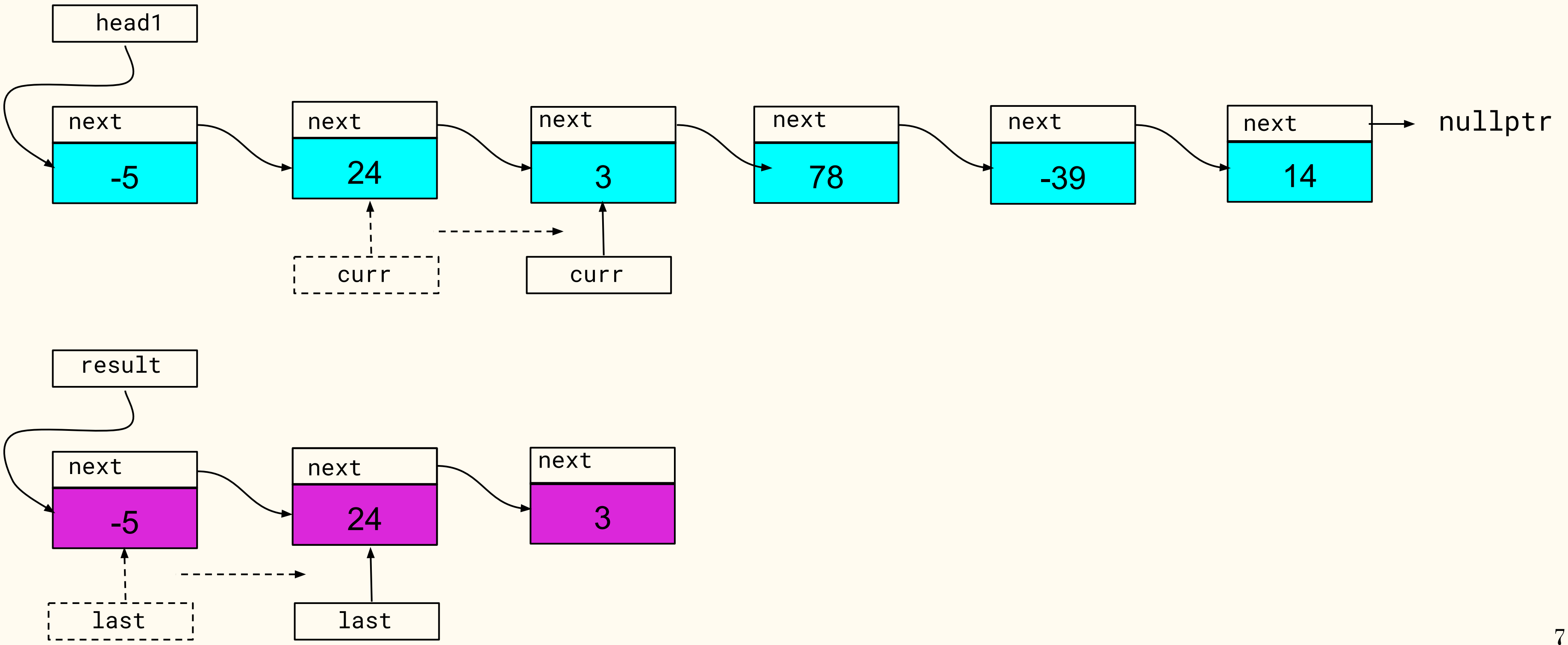
Duplicating a list



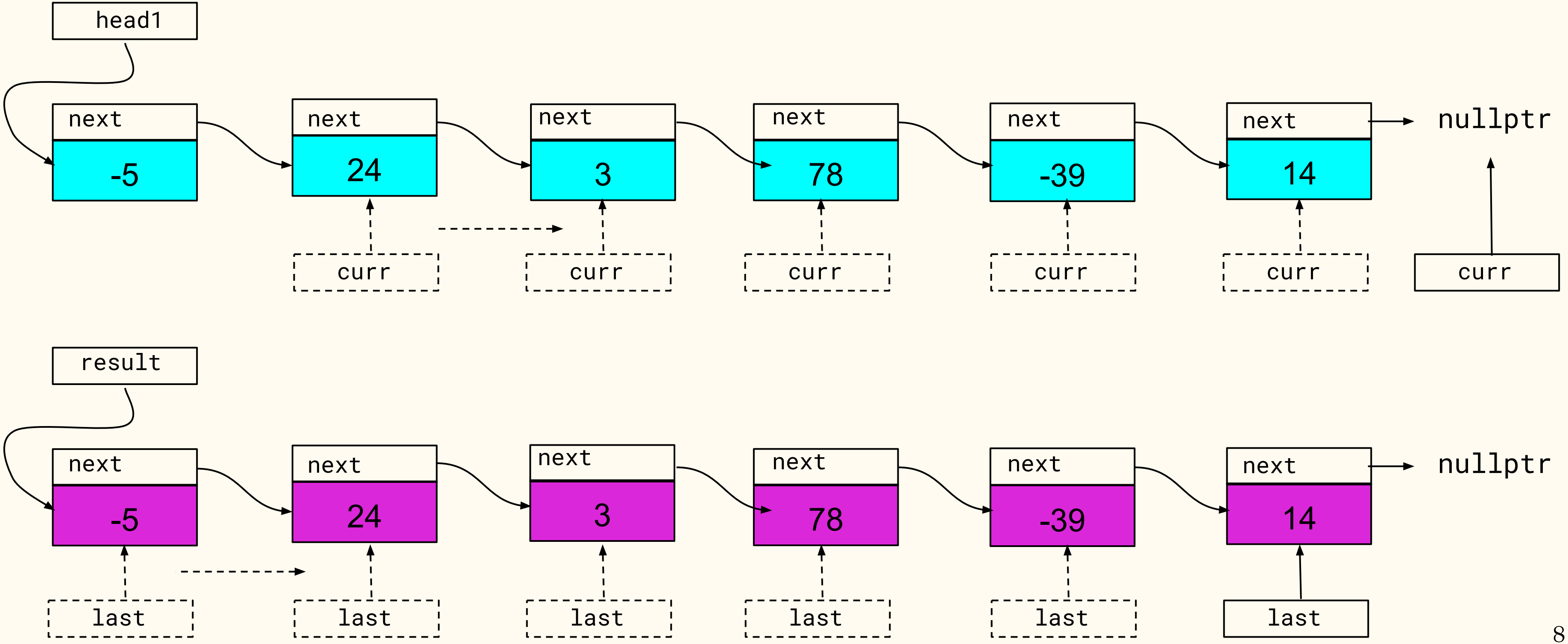
Duplicating a list



Duplicating a list



Duplicating a list

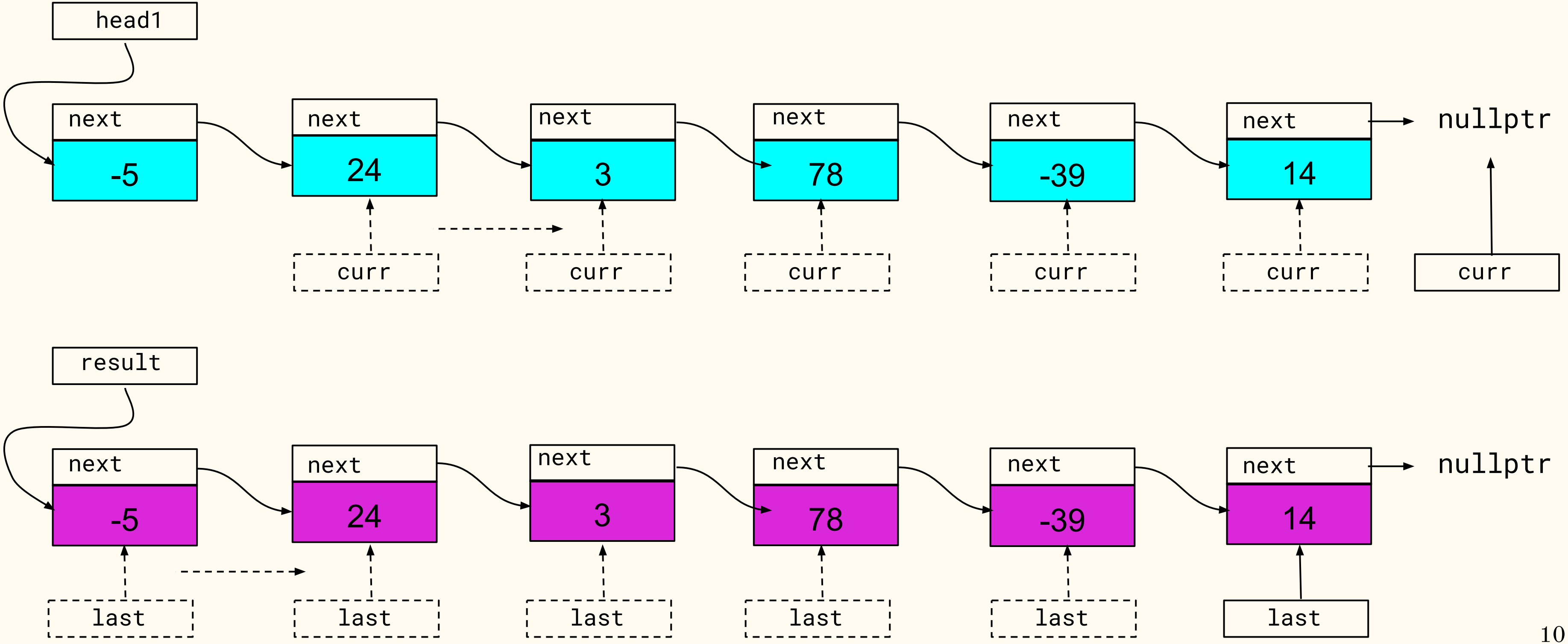


TurningPoint

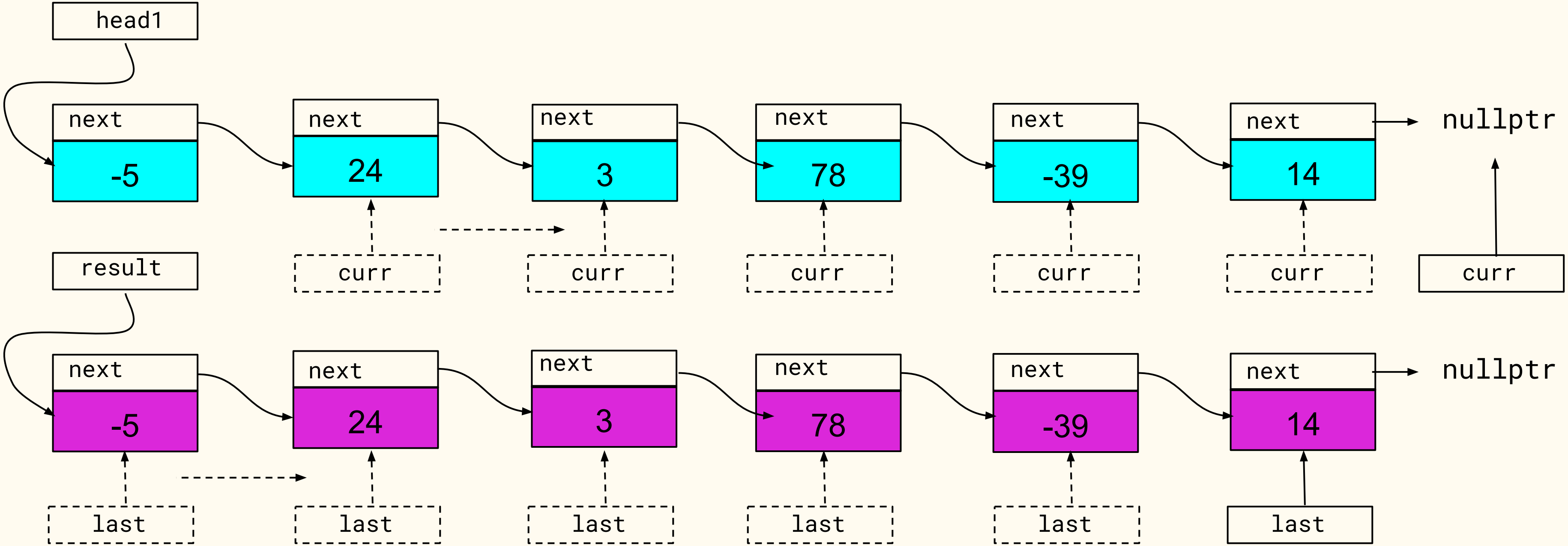
SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220418<A|D>

Replace <A|D> with this section's letter

Which condition will indicate the the full list has been duplicated?



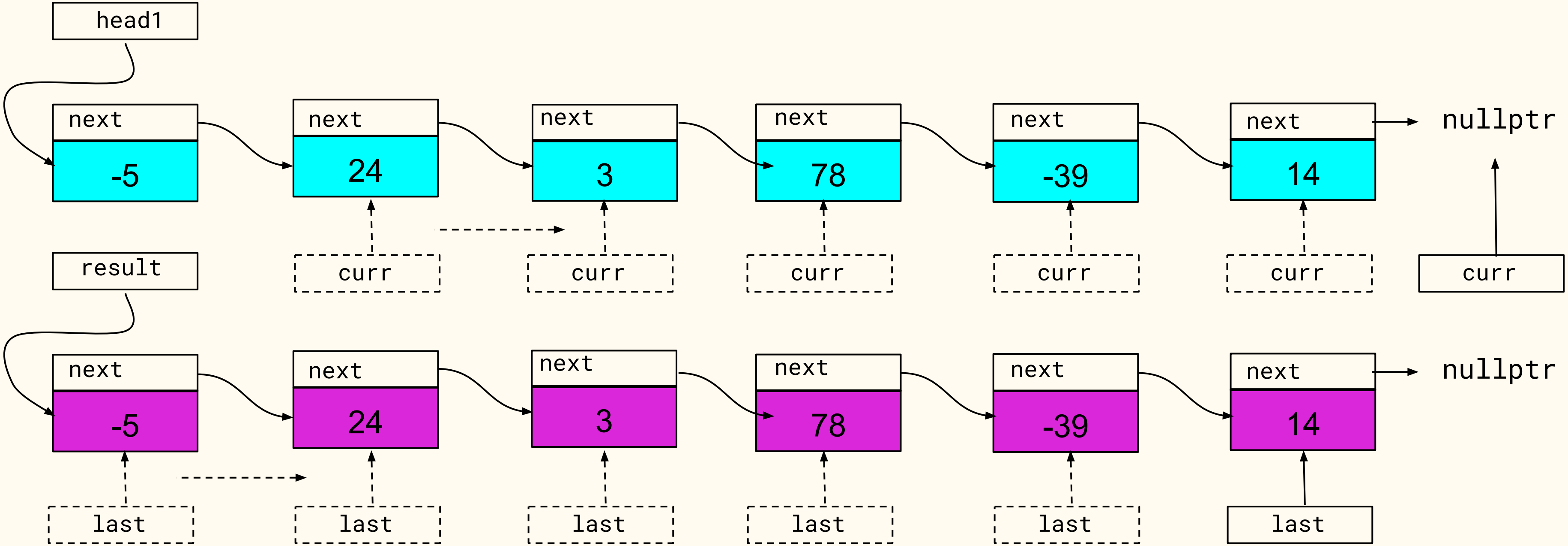
Duplicating a list



```
___ duplicate_list() { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

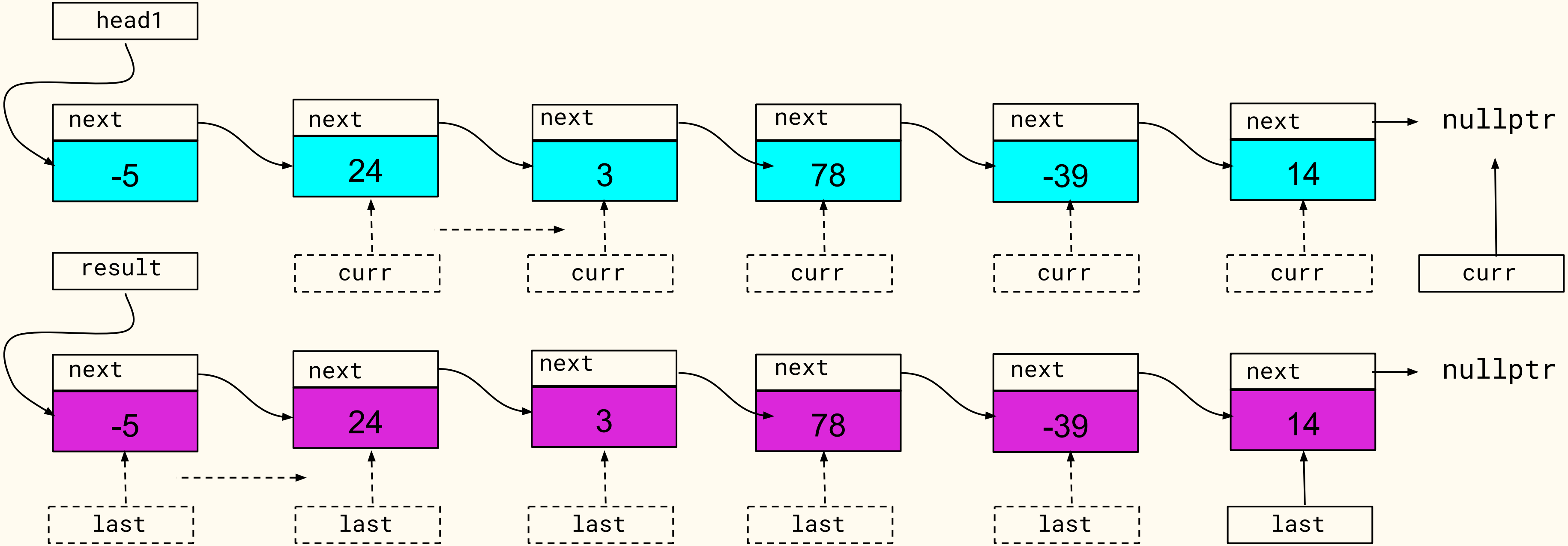
Duplicating a list



```
___ duplicate_list(___ ___) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

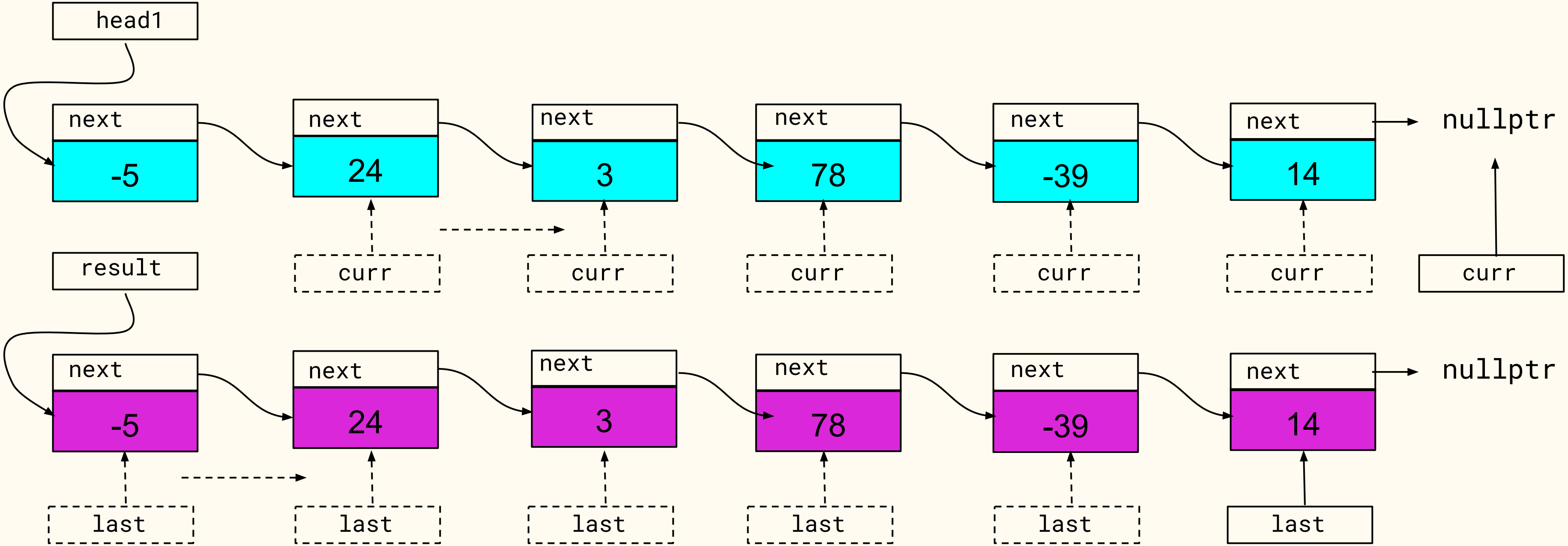
Duplicating a list



```
___ duplicate_list(Node* head_ptr) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

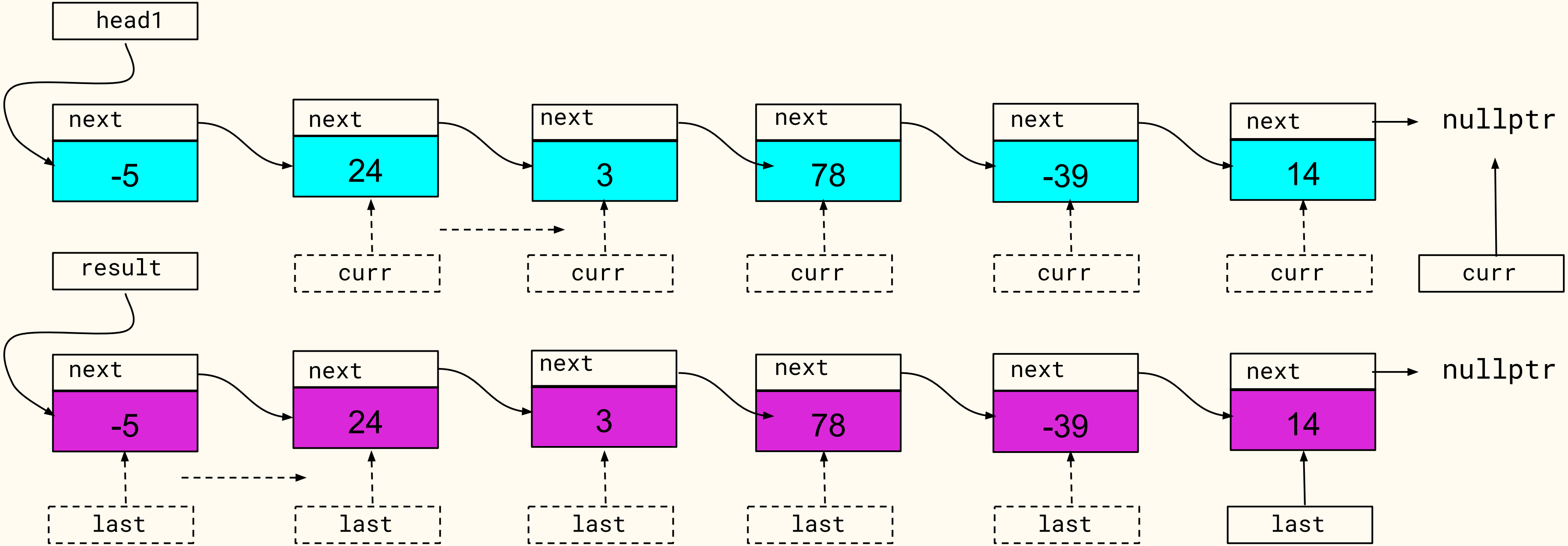
Duplicating a list



```
___ duplicate_list(___ Node* head_ptr) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

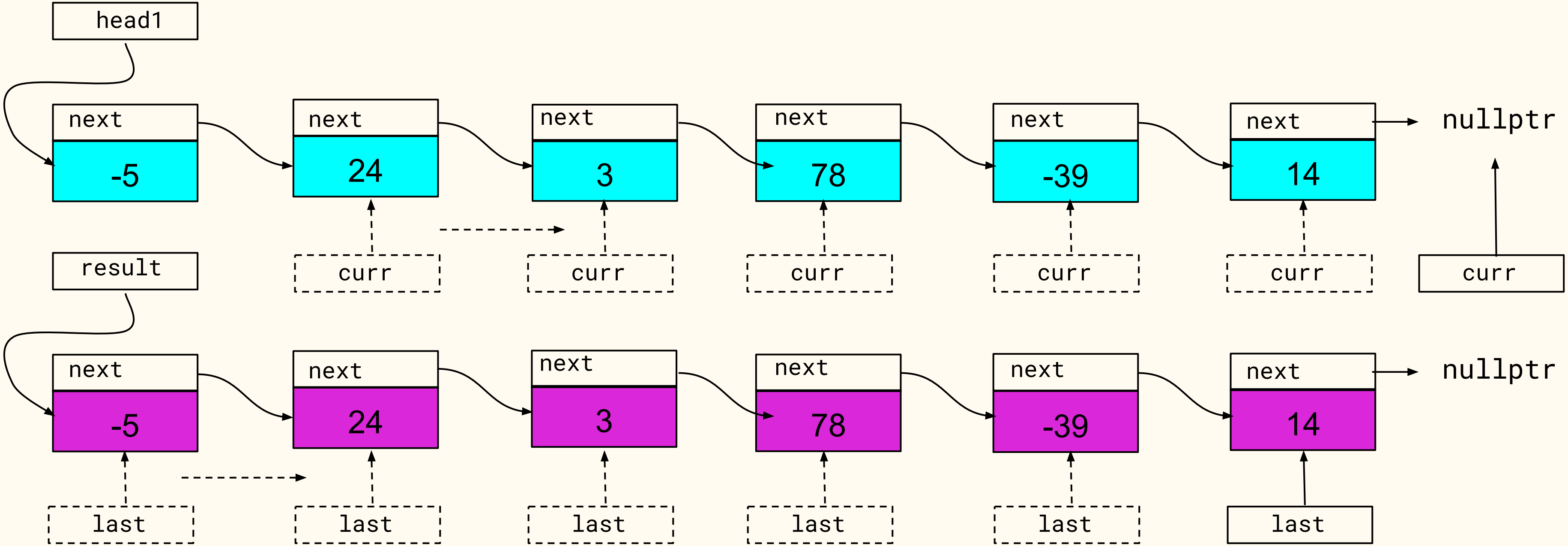
Duplicating a list



```
___ duplicate_list(_57_ Node* head_ptr) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

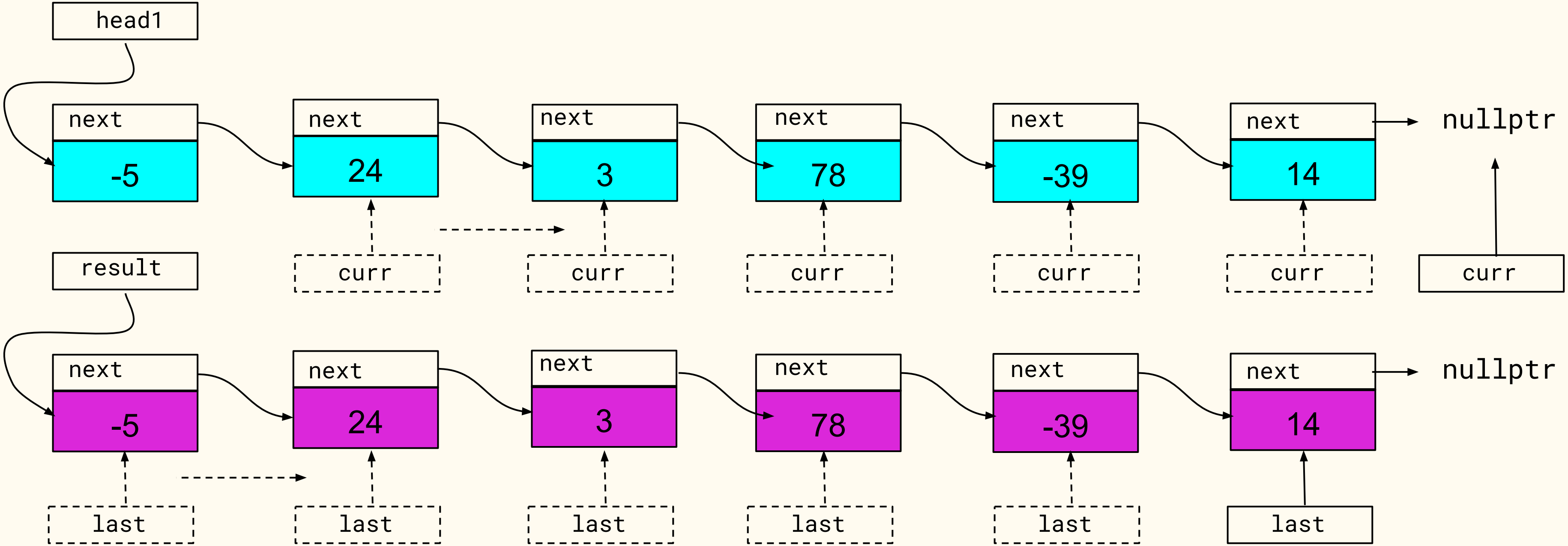
Which keyword replaces blank #57 to ensure the list to be duplicated is not modified?



```
___ duplicate_list(_57_ Node* head_ptr) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

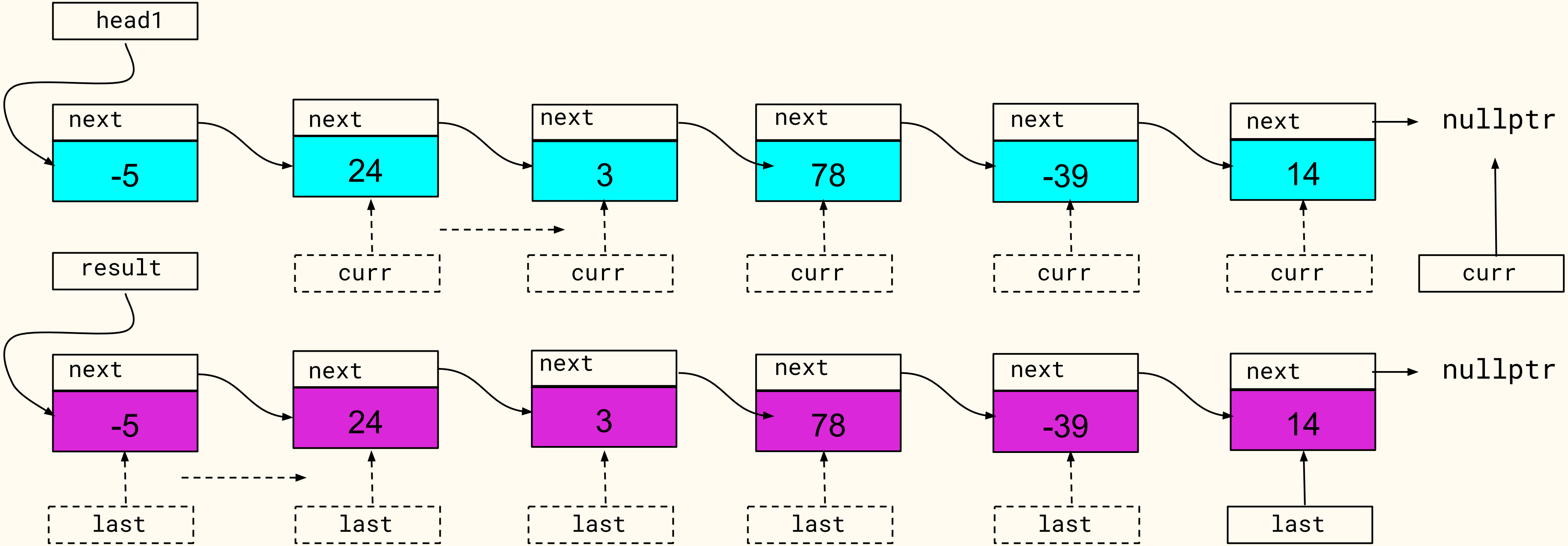
Duplicating a list



```
___ duplicate_list(const Node* head_ptr) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

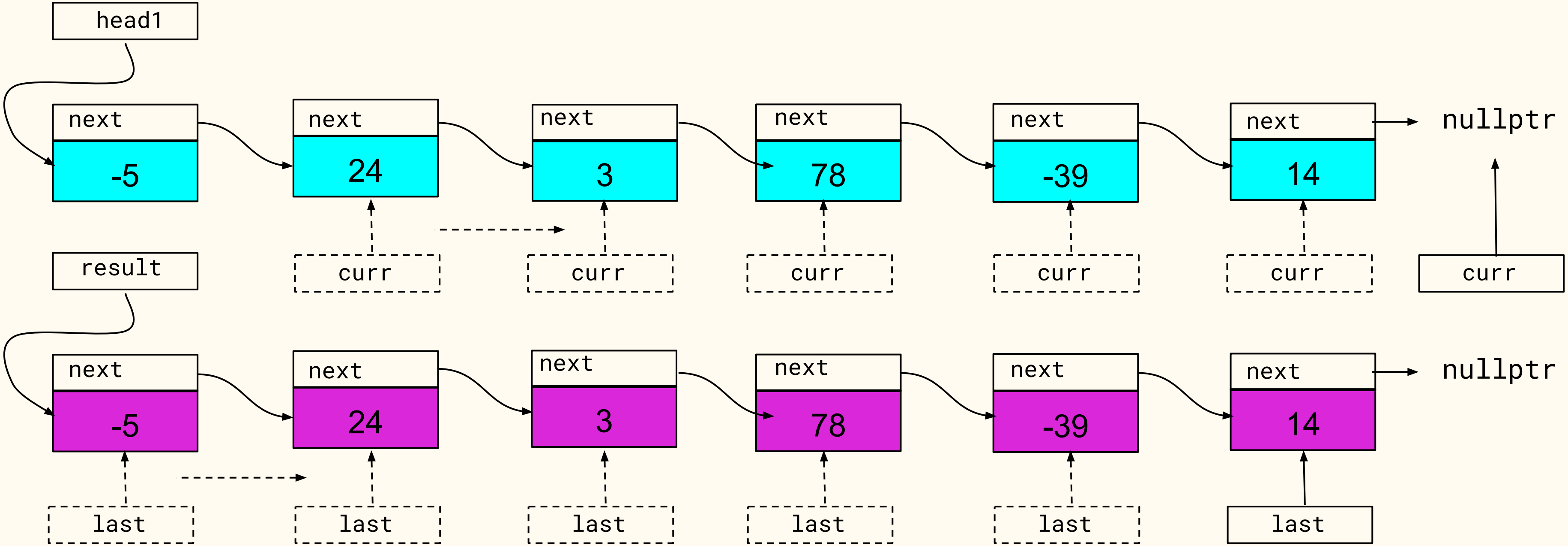
Duplicating a list



```
_58_ duplicate_list(const Node* head_ptr) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

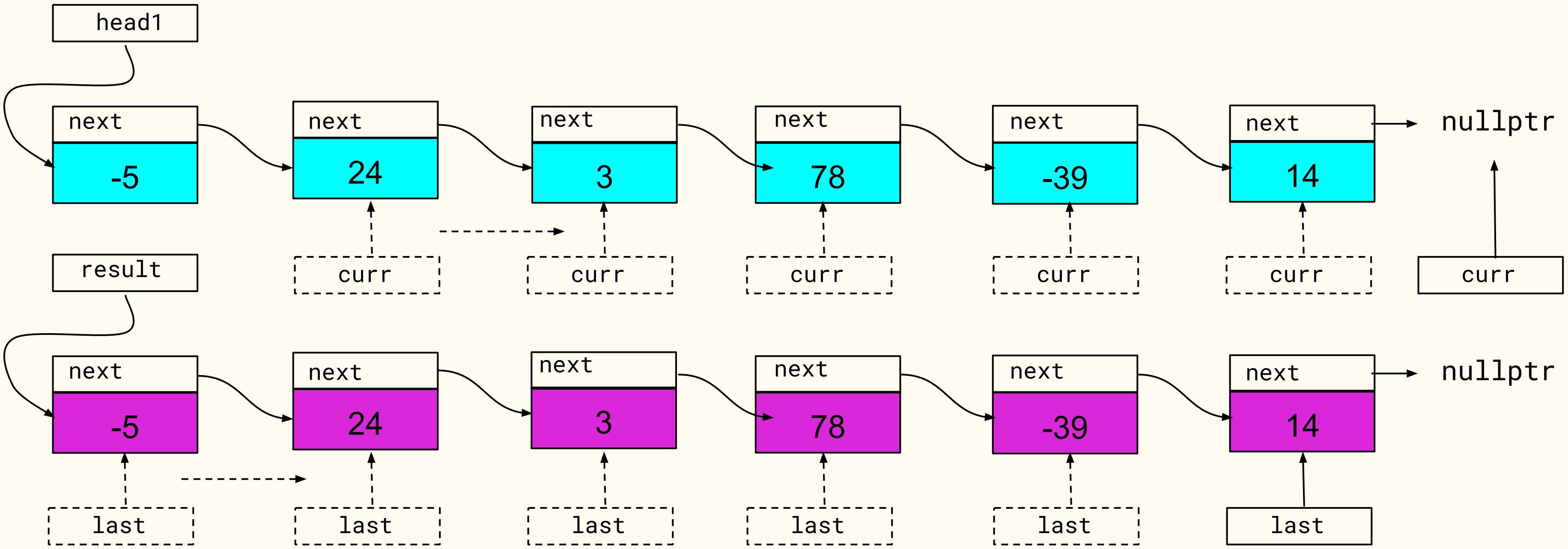
Which return type replaces blank #58 to return the duplicate list?



```
_58_ duplicate_list(const Node* head_ptr) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

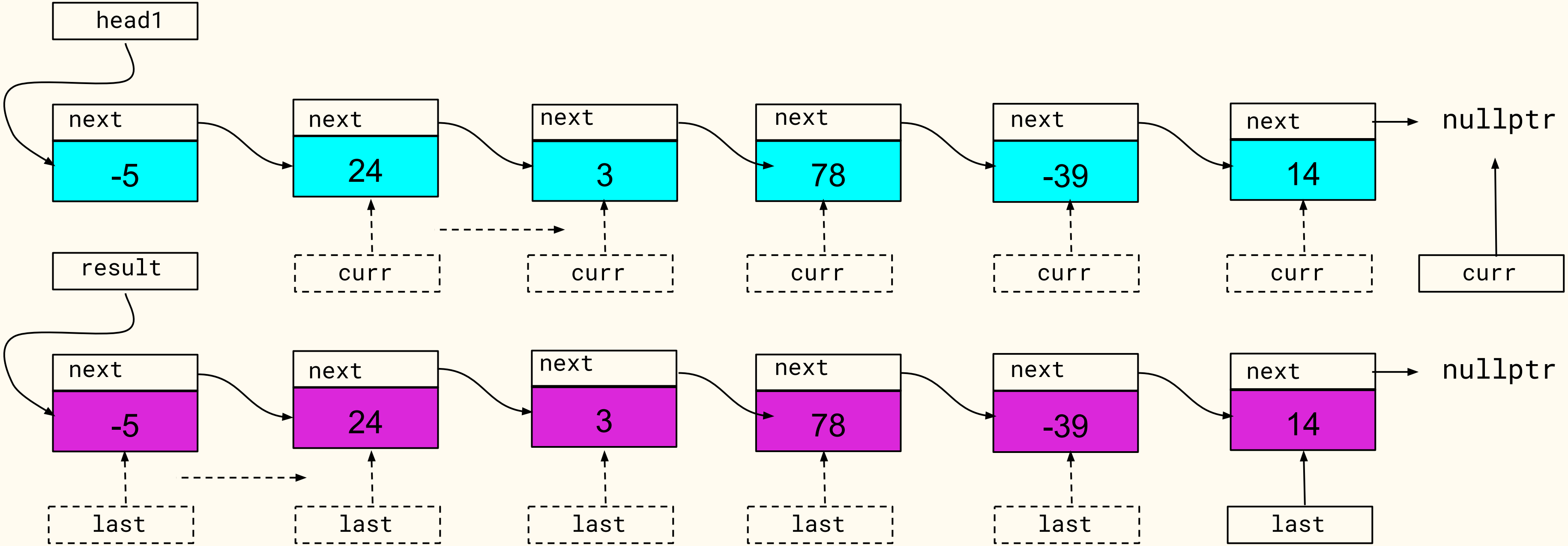
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

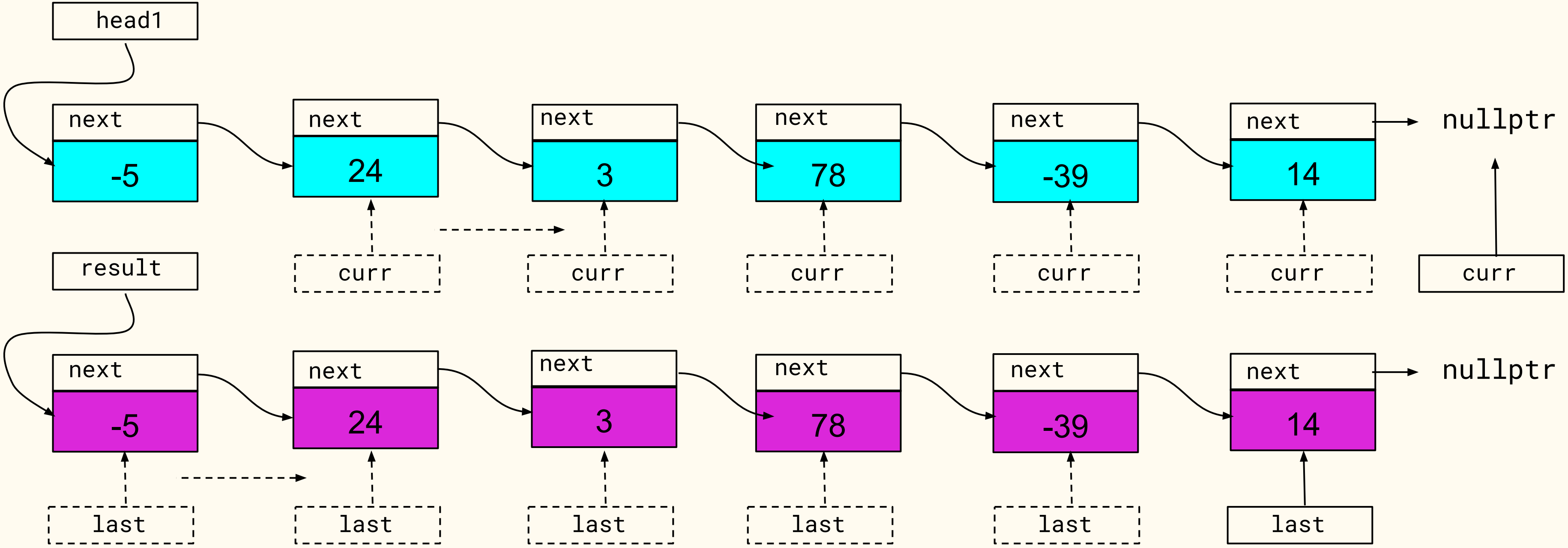
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    ___
}
```

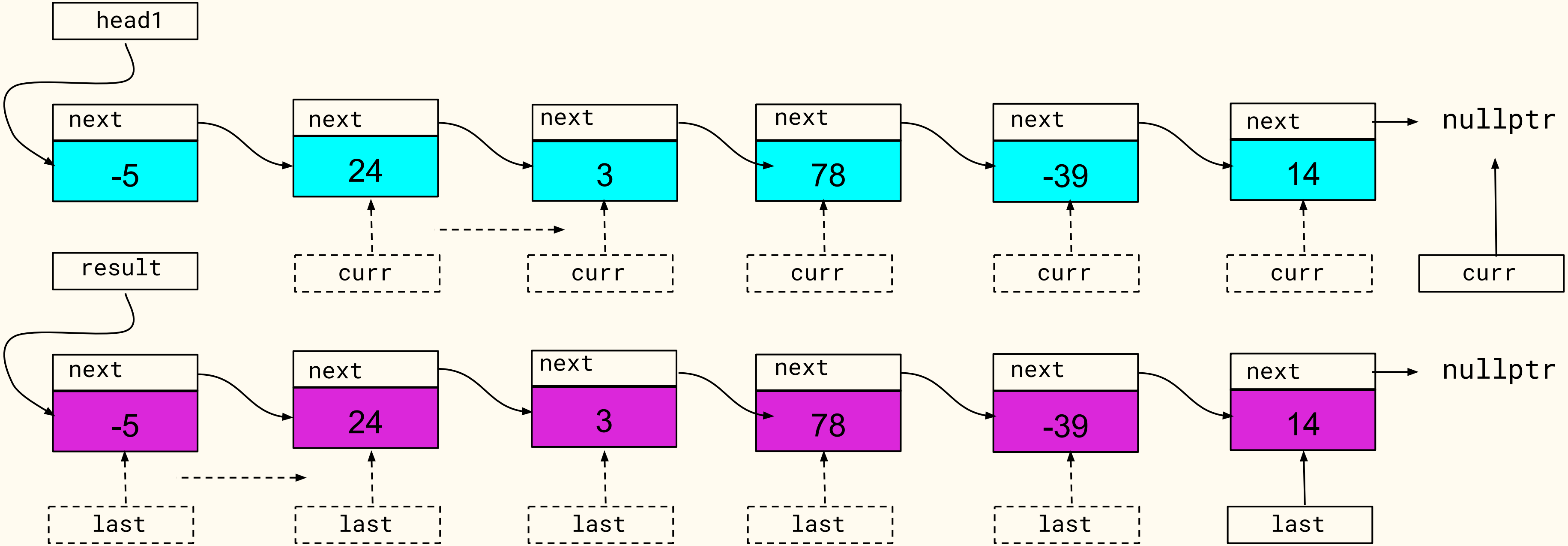
What should be returned when `nullptr` is passed as the argument to `head_ptr`?



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    ___
}
```

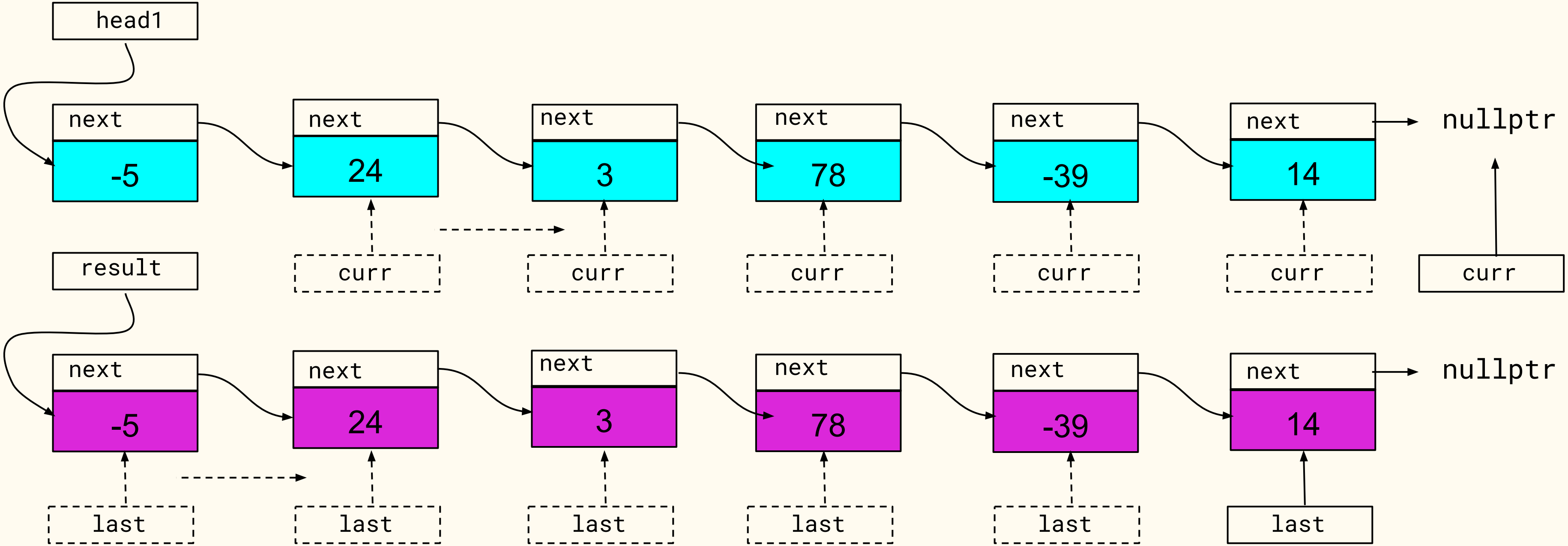
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
}
```

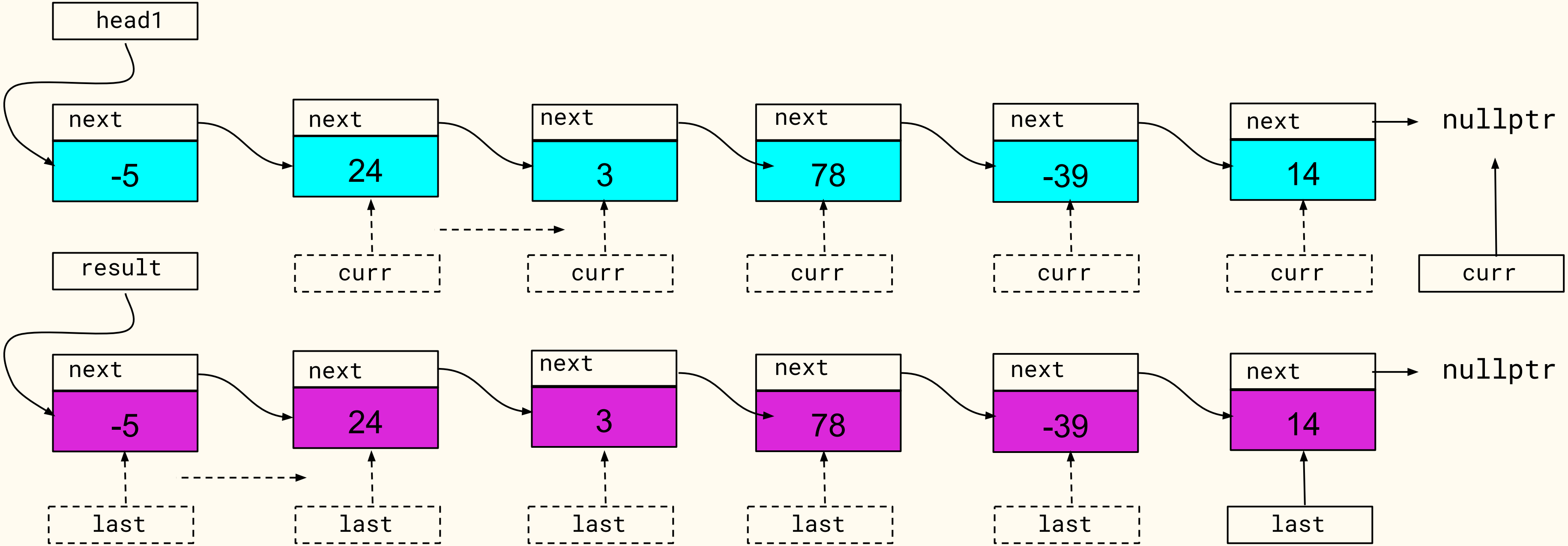
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
}
```

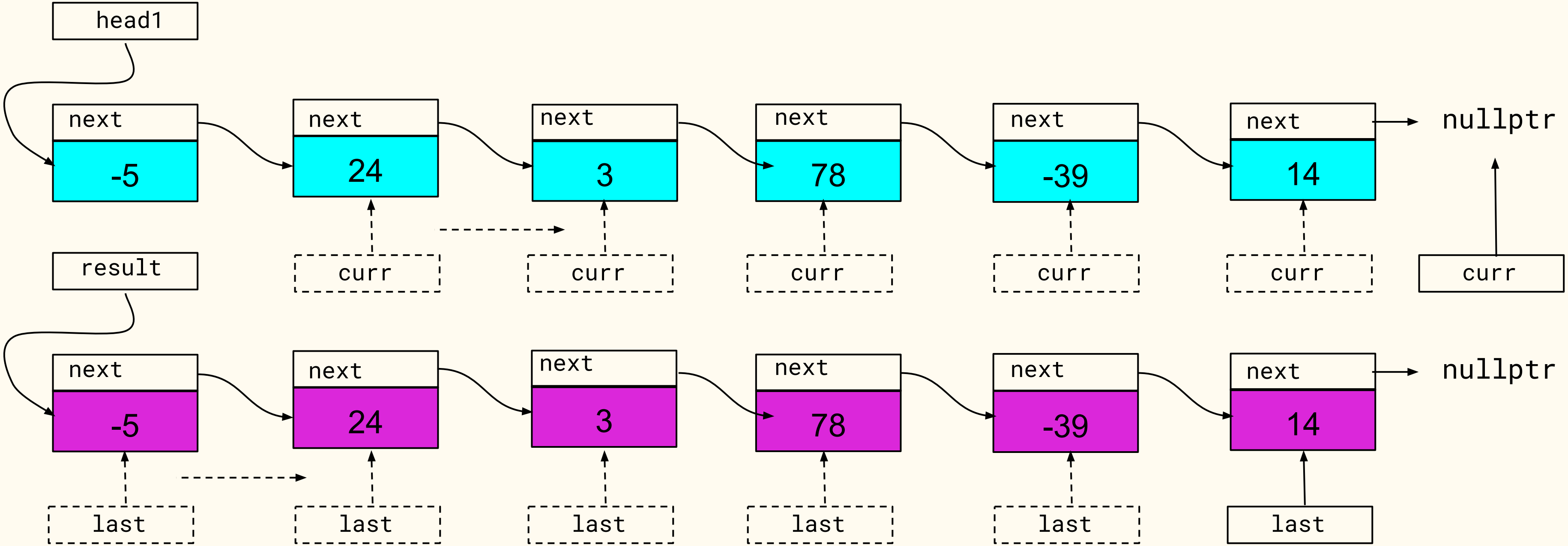
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    ---
}
```

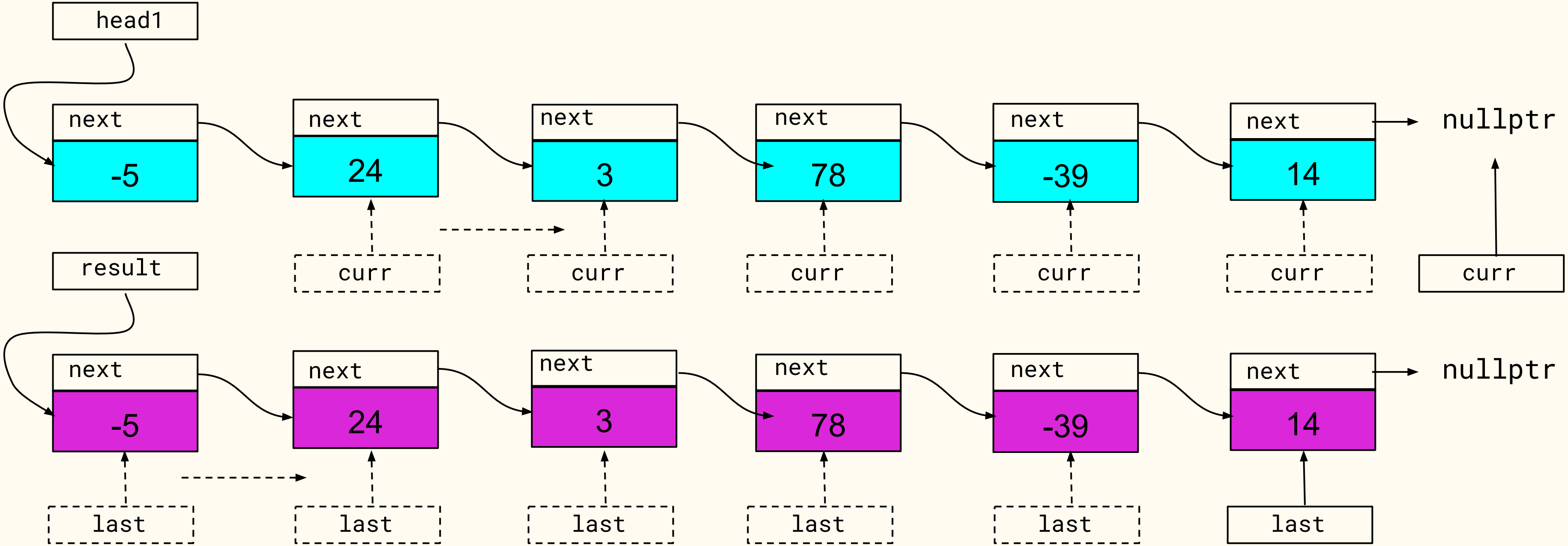
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = ___;
}
```

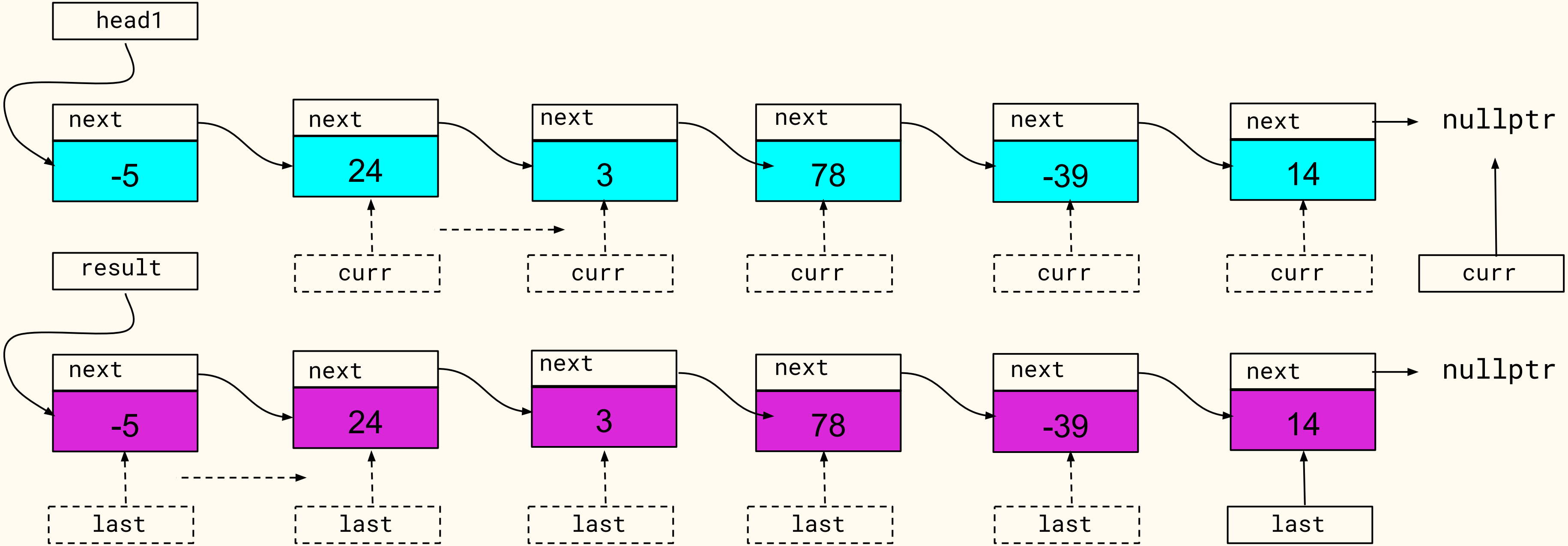
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = _59_;
}
```

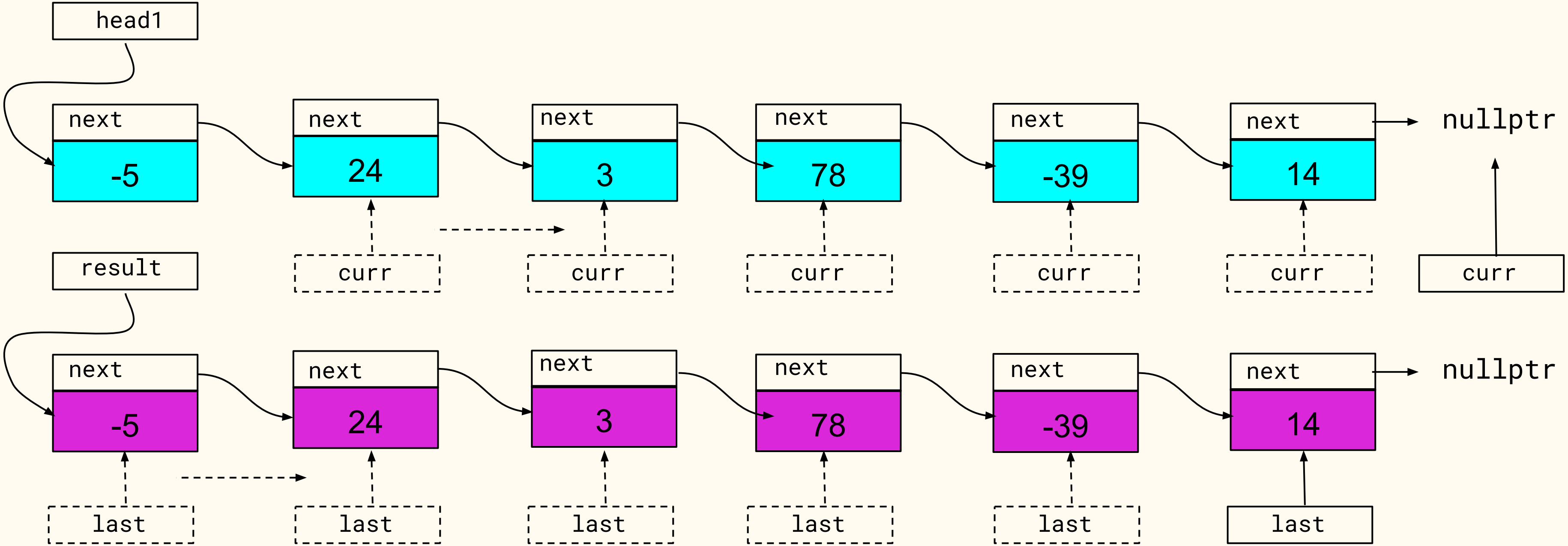
What replaces blank #59 to duplicate the head Node of the original list?



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = _59_;
}
```

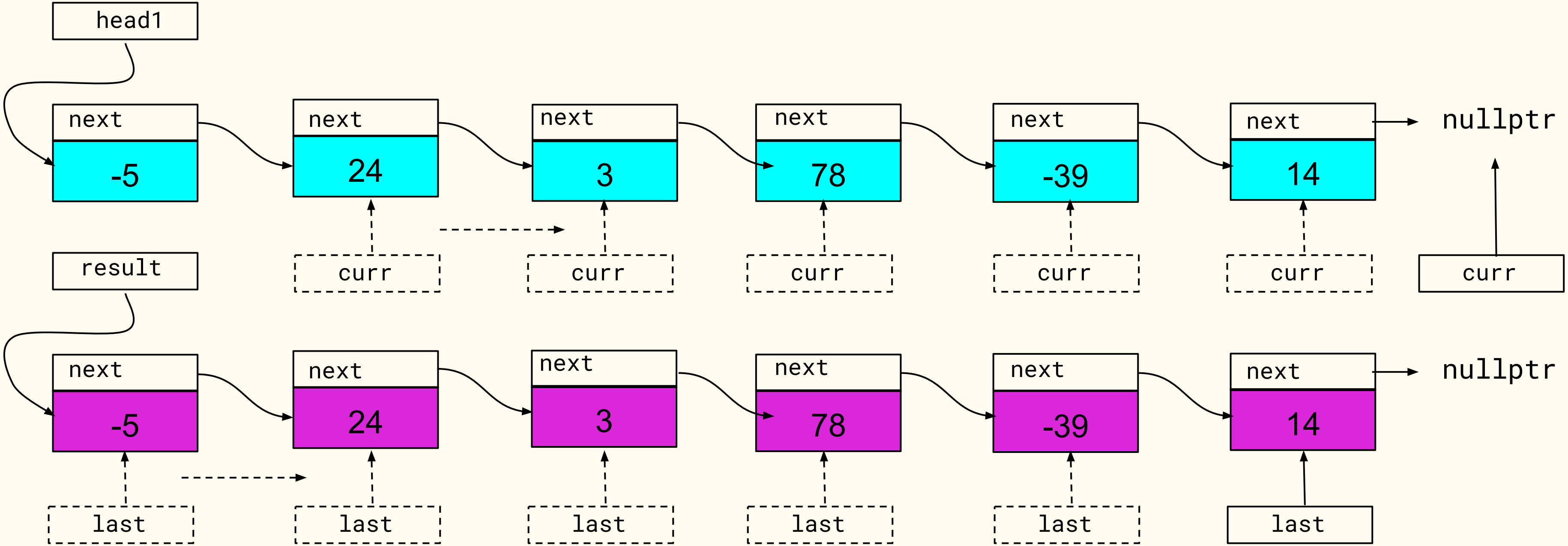
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
}
```

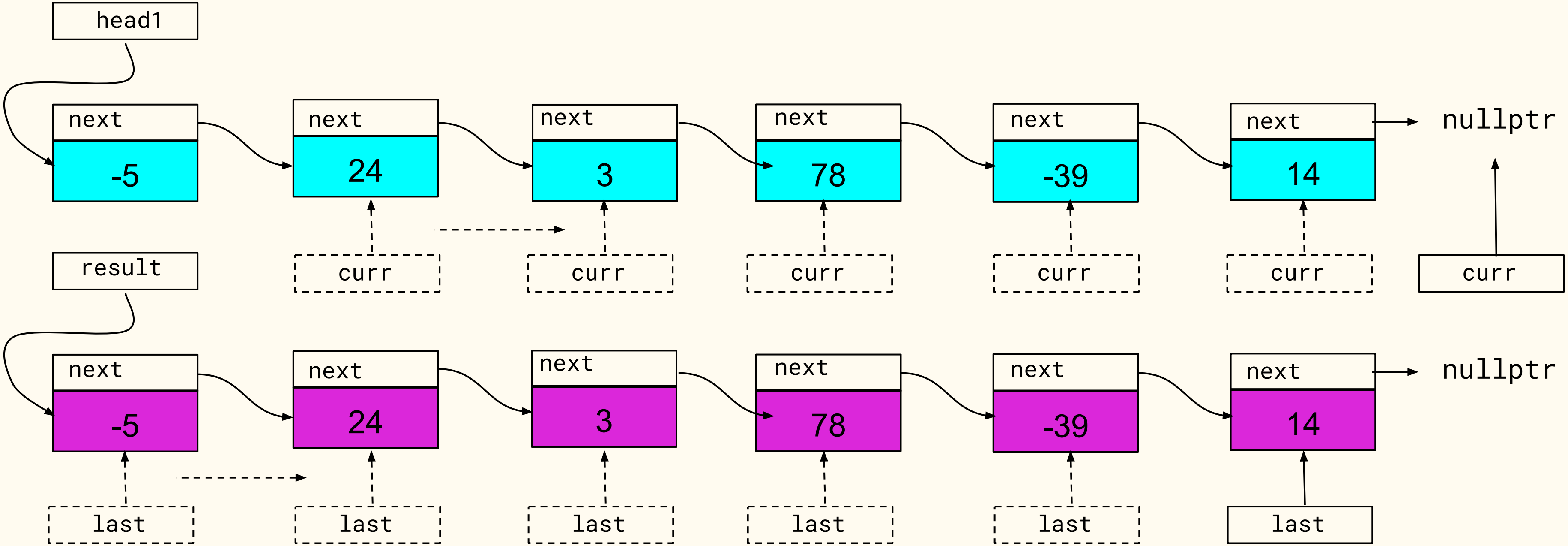
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    ___
}
```

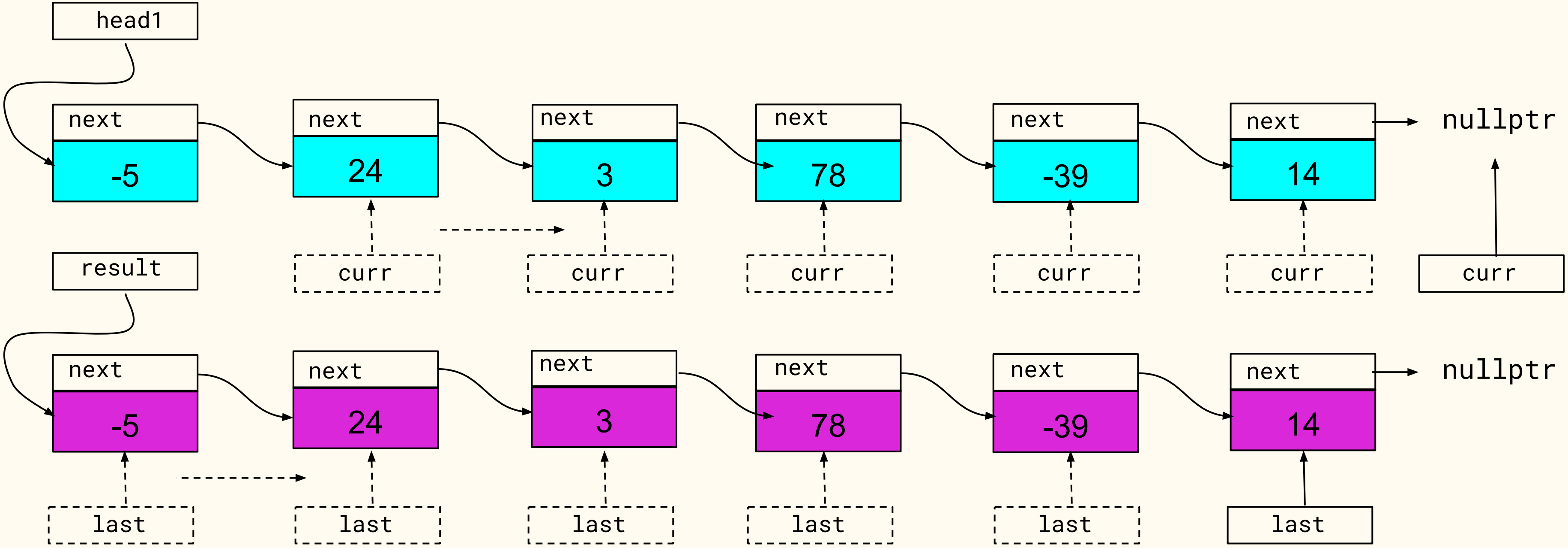
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = ___;
}
```

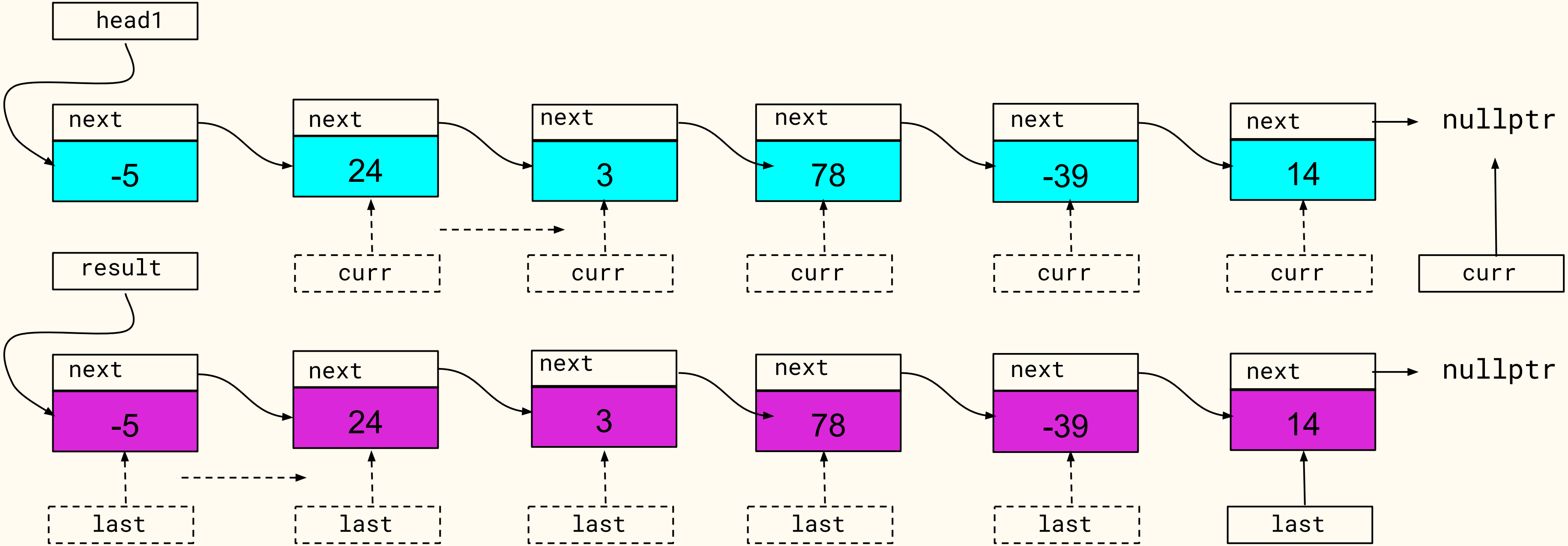
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = ___;
}
```

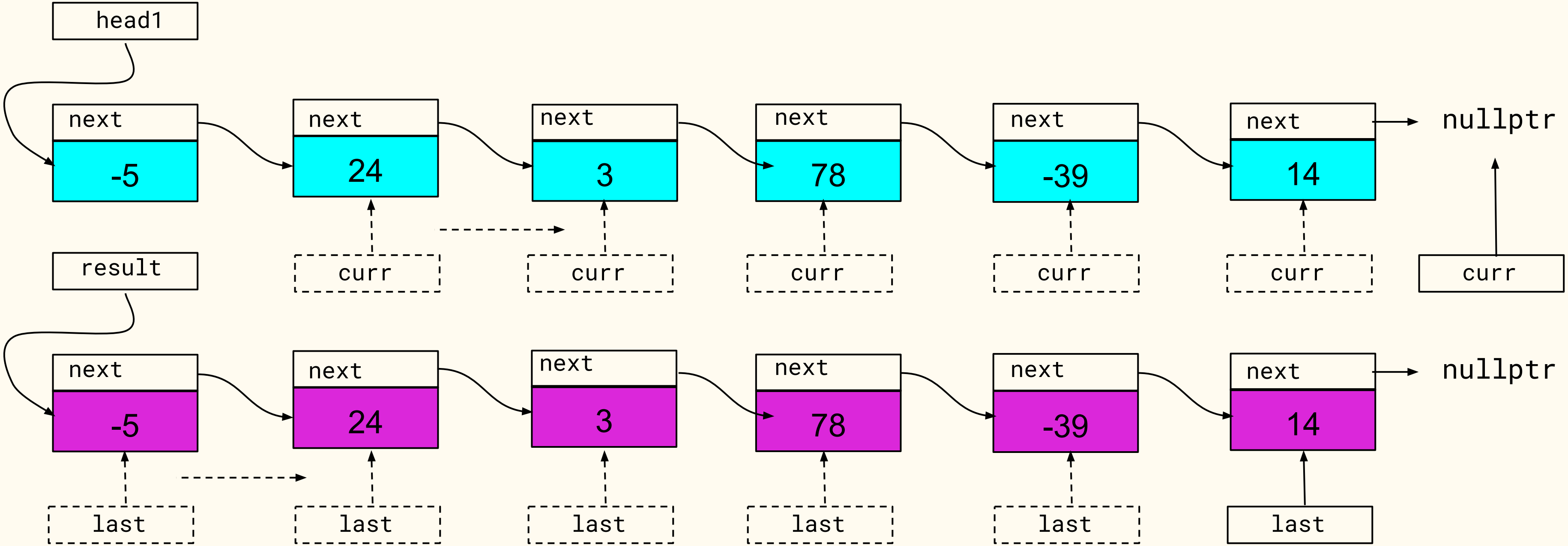
What replaces blank #60 to assign the address of the duplicated head `Node` to `last`?



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = _60_;
}
```

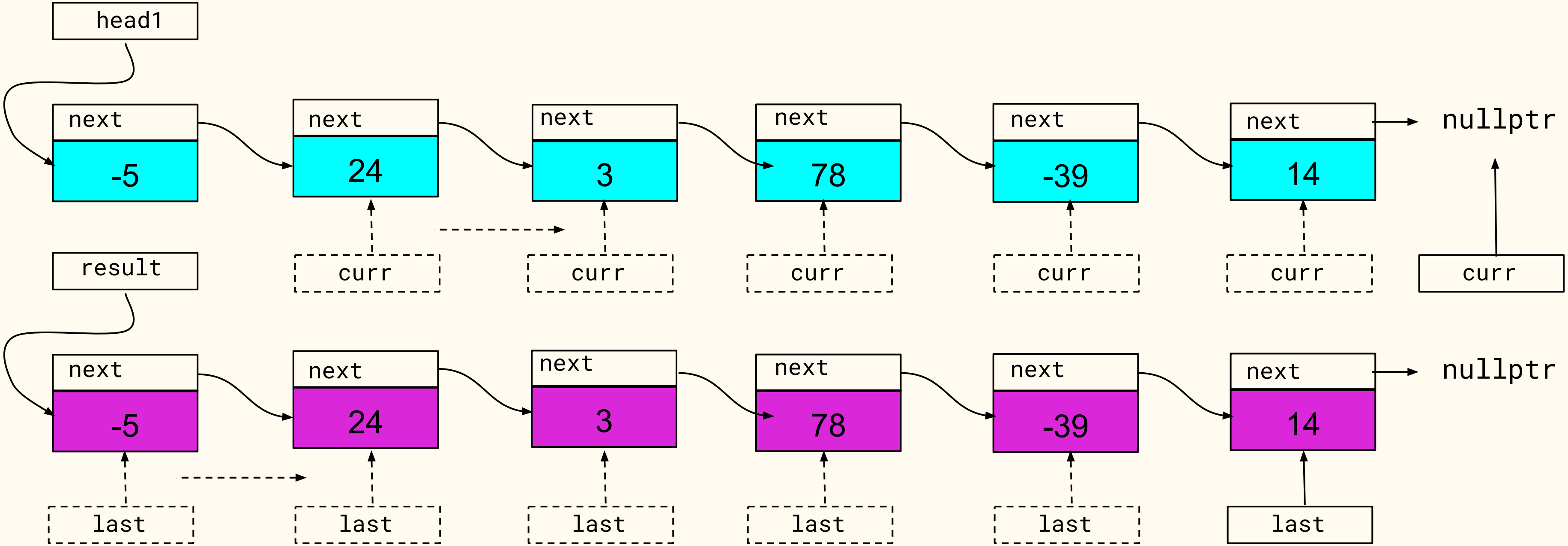
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
}
```

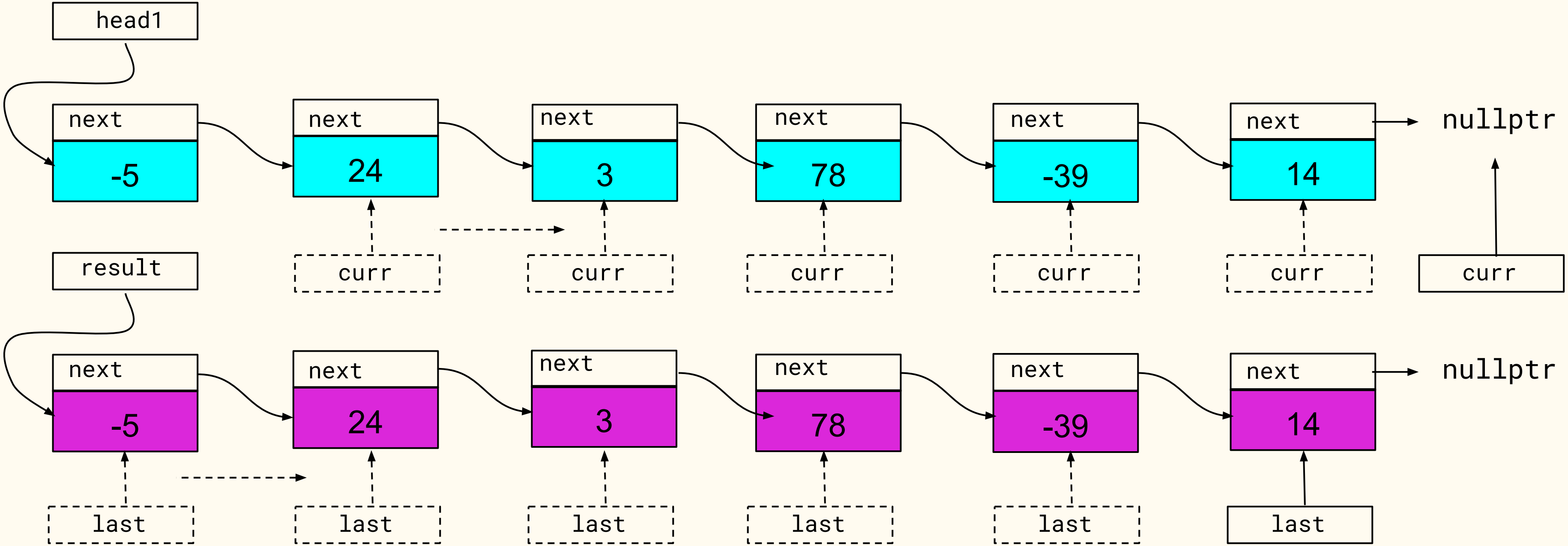
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;

    ___
}
```

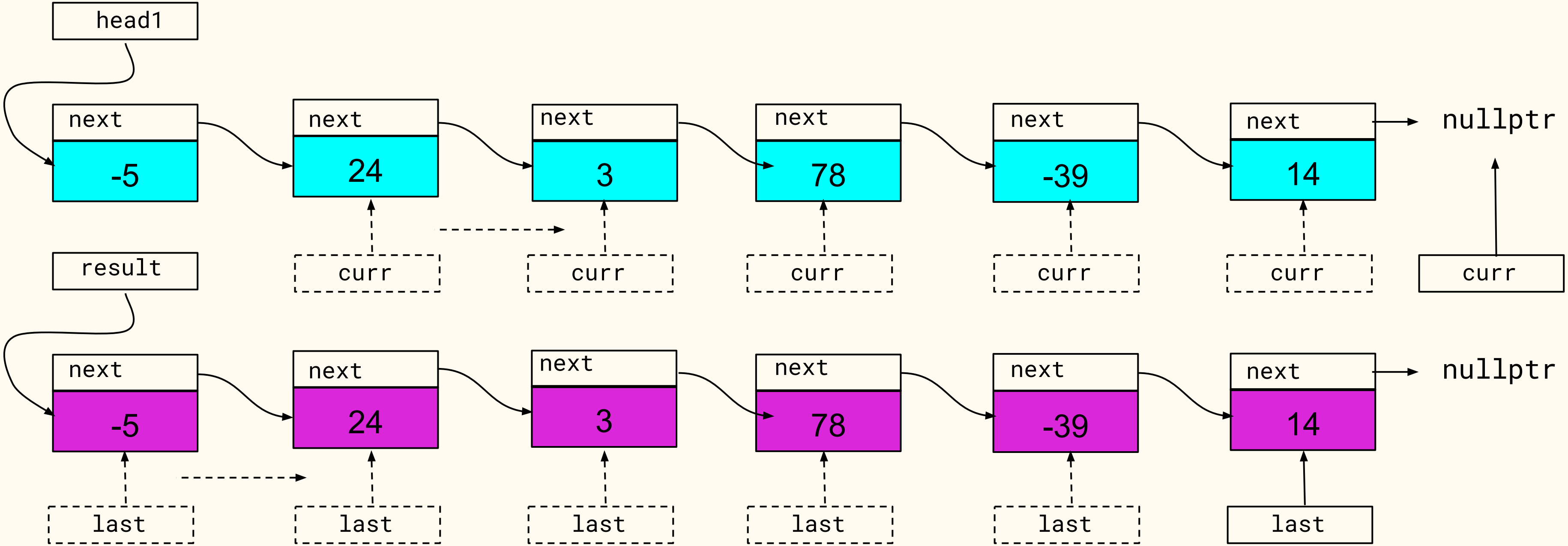
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    Node* curr = ___;
}
```

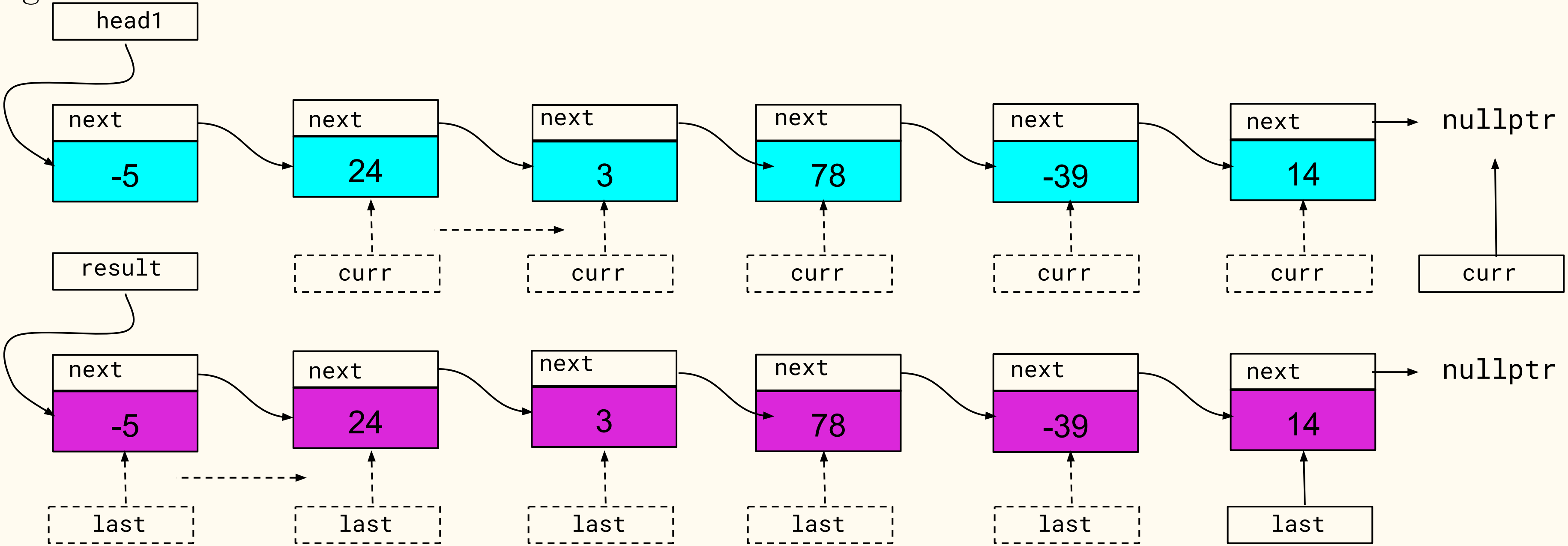
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    Node* curr = _61_;
}
```

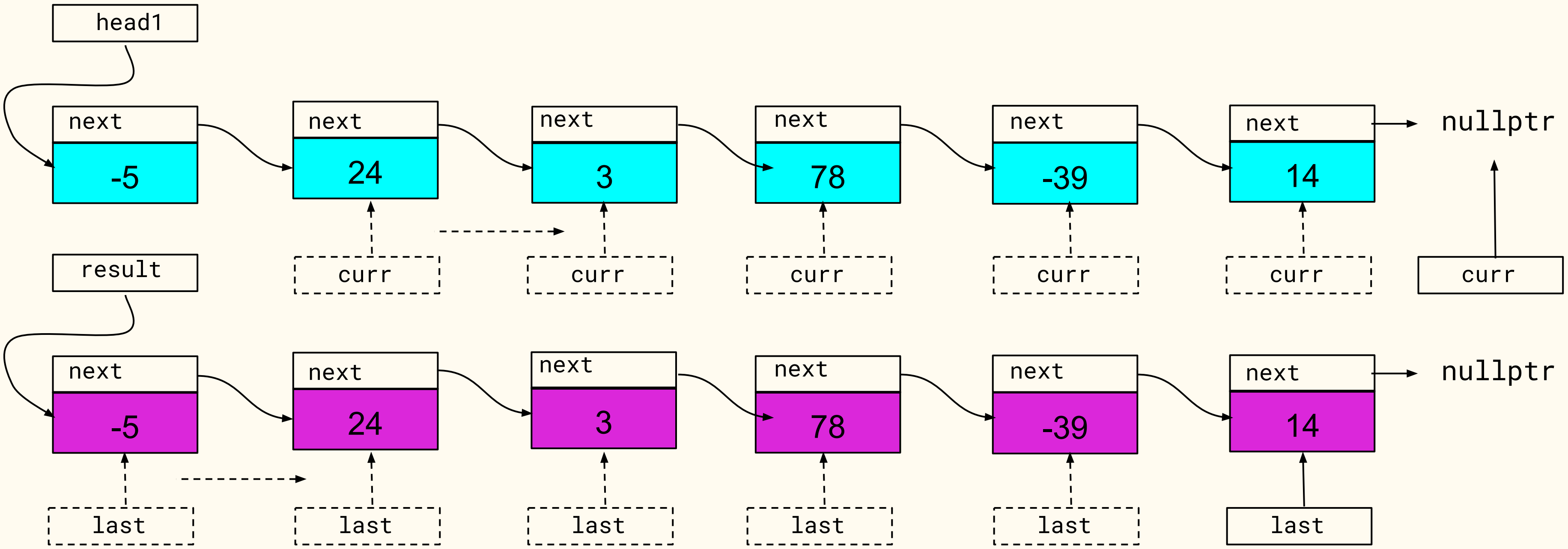
Which expression replaces blank #61 to point `curr` at the `Node` following the head `Node` in the original list?



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    Node* curr = _61_;
}
```

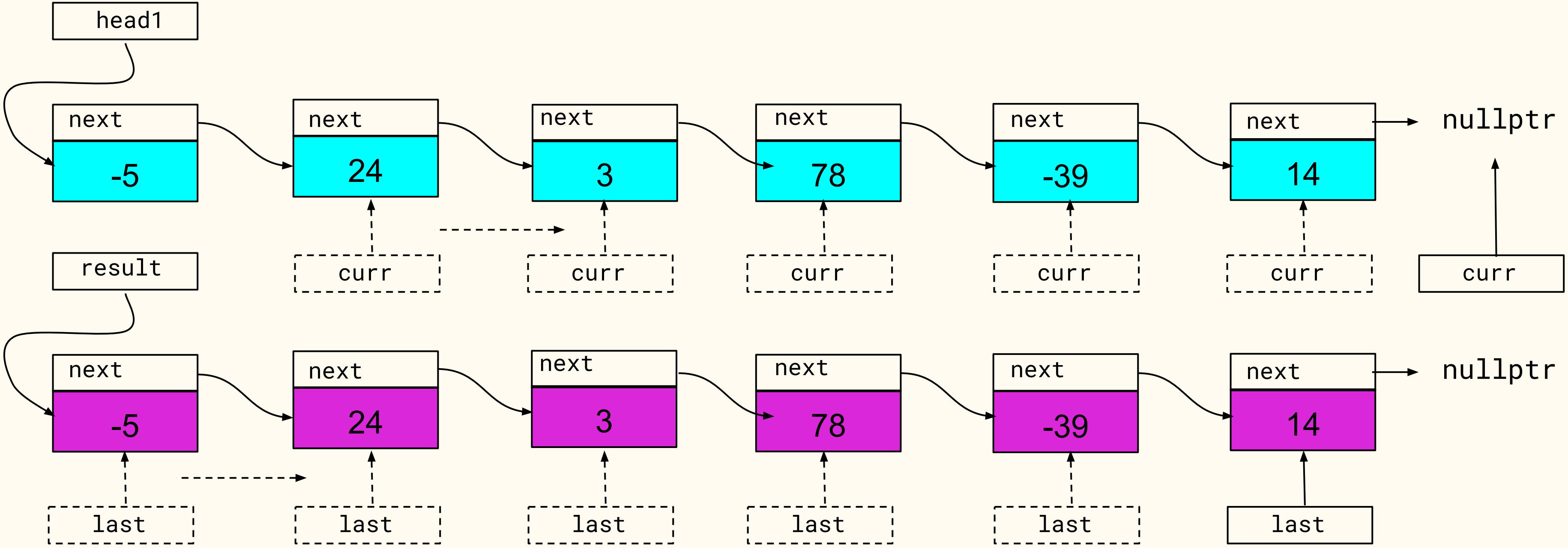
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;
}
```

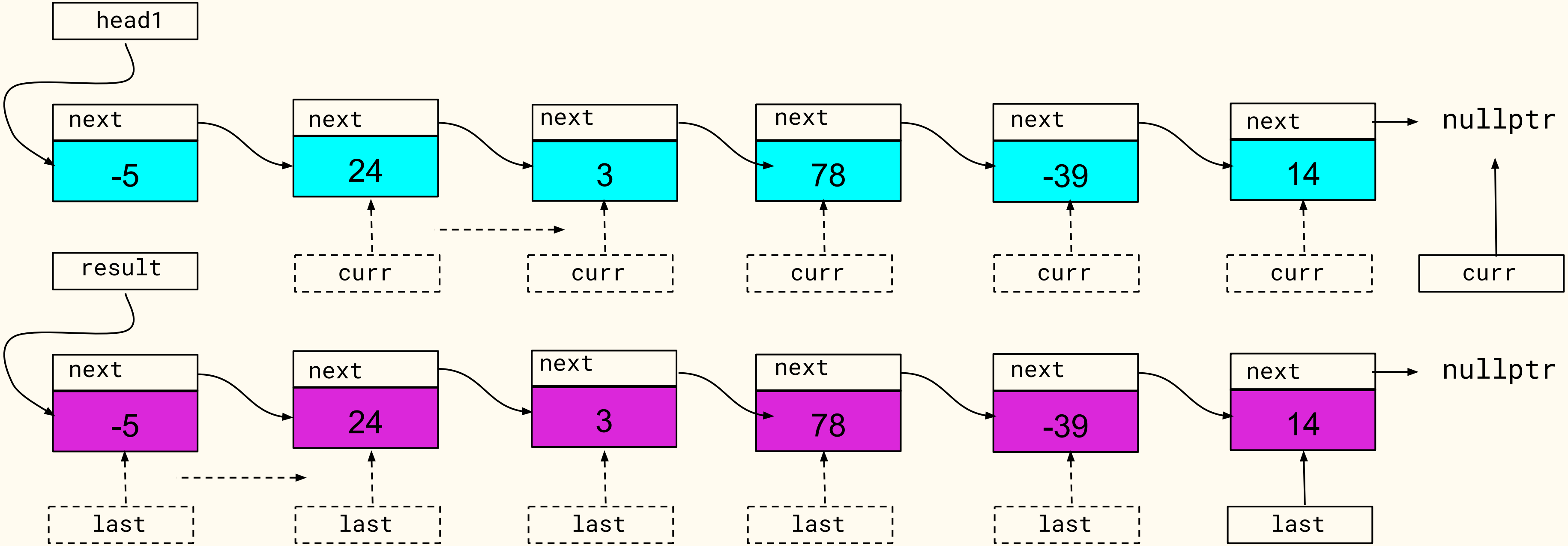
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;


    ---
}
```

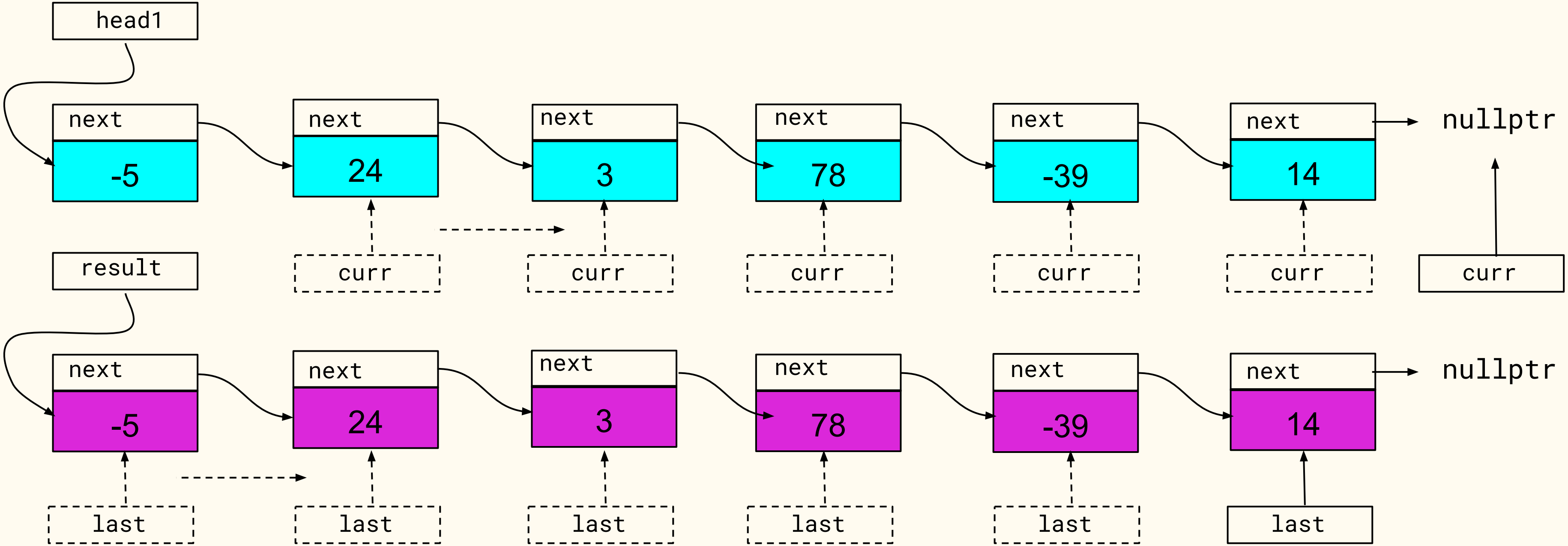
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (___) { }
}
```

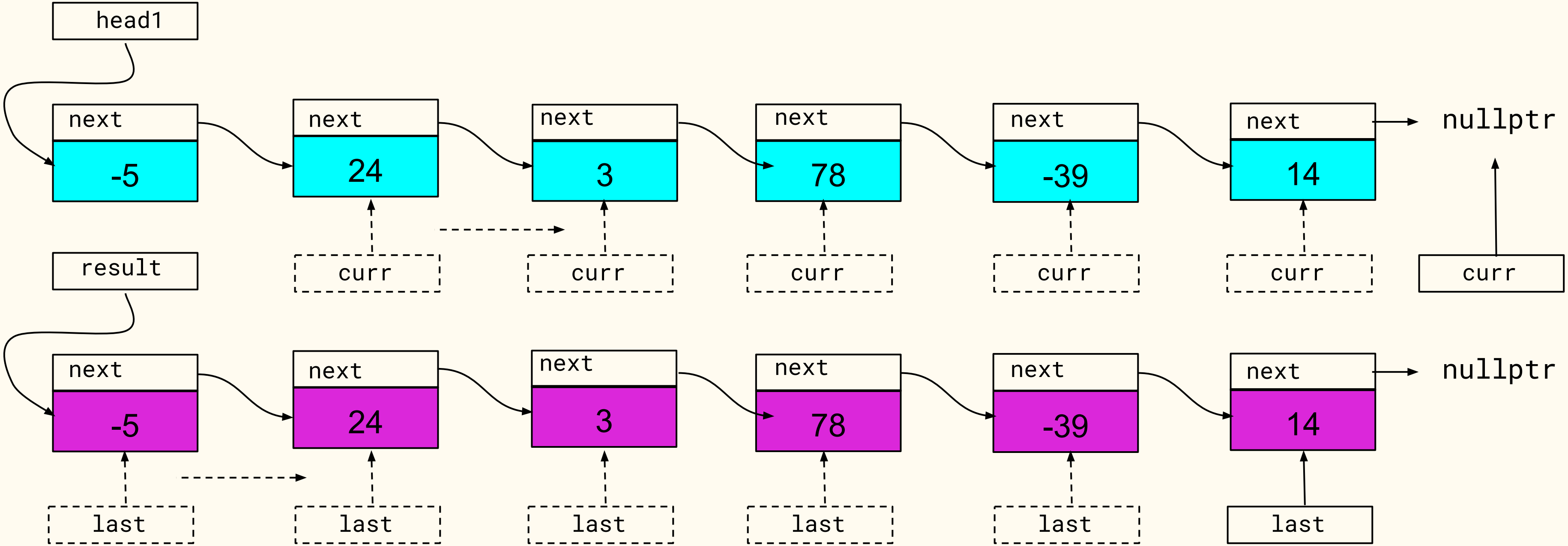
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (_61_) { }
}
```

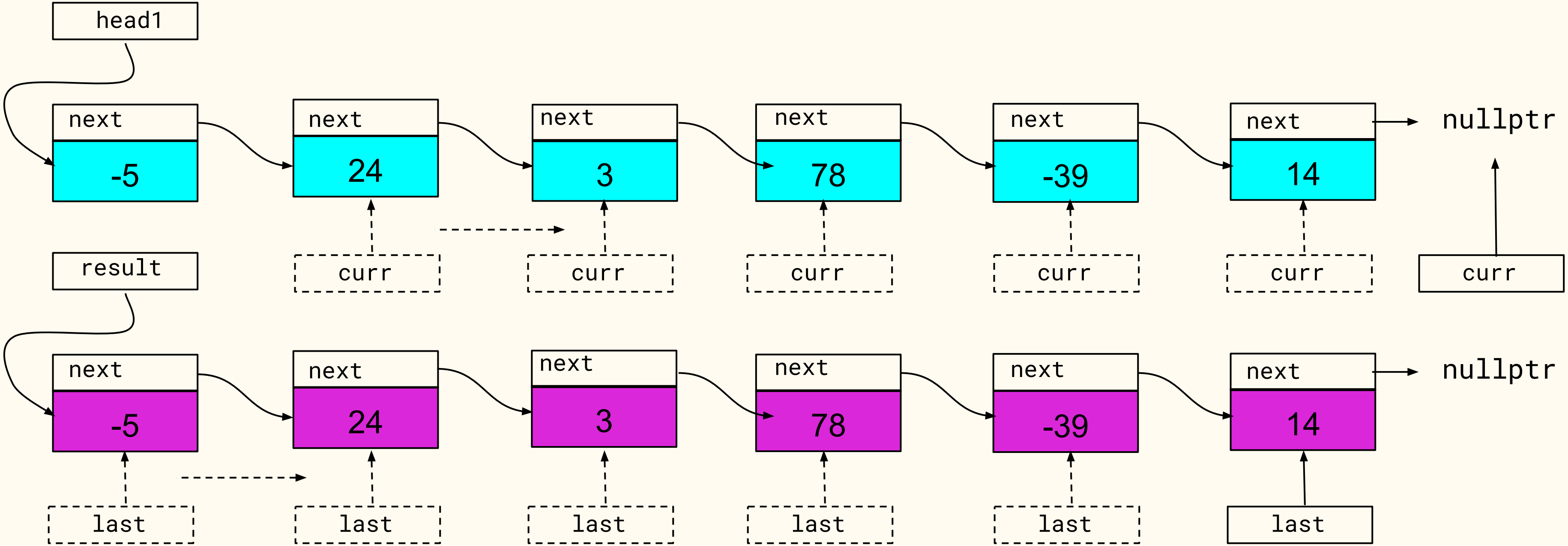
Which boolean expression replaces blank #61 to terminate the loop when the list is duplicated?



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (_61_) { }
}
```

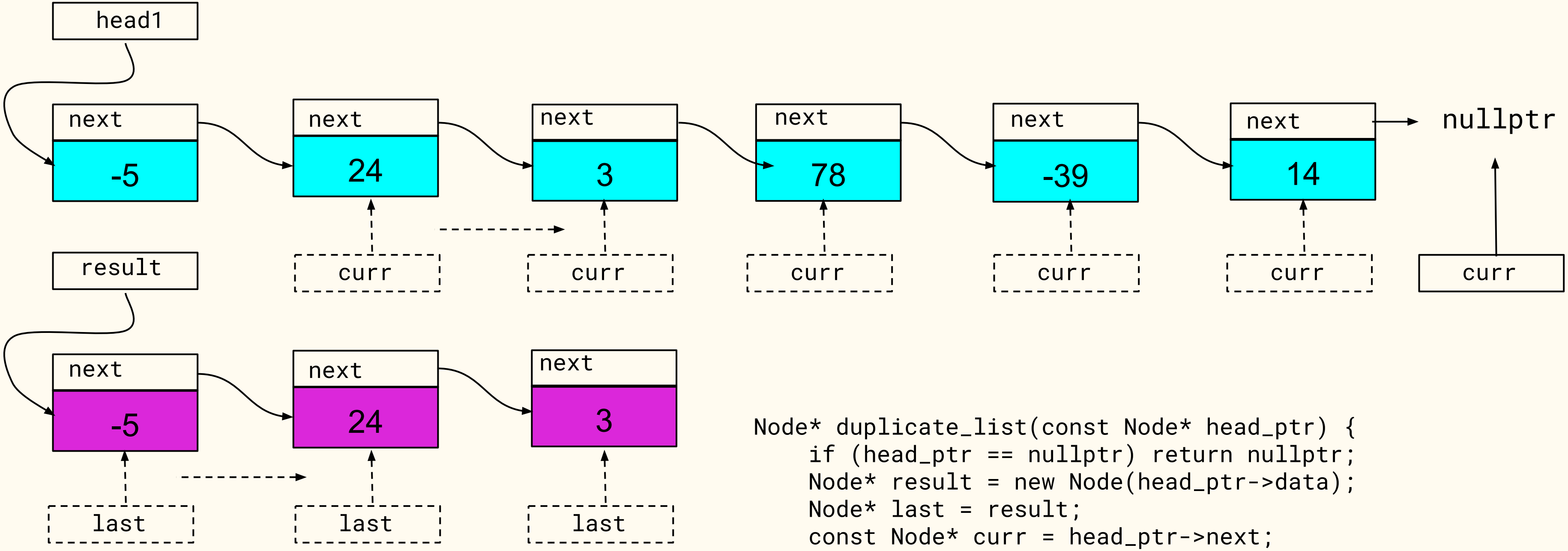
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) { }
}
```

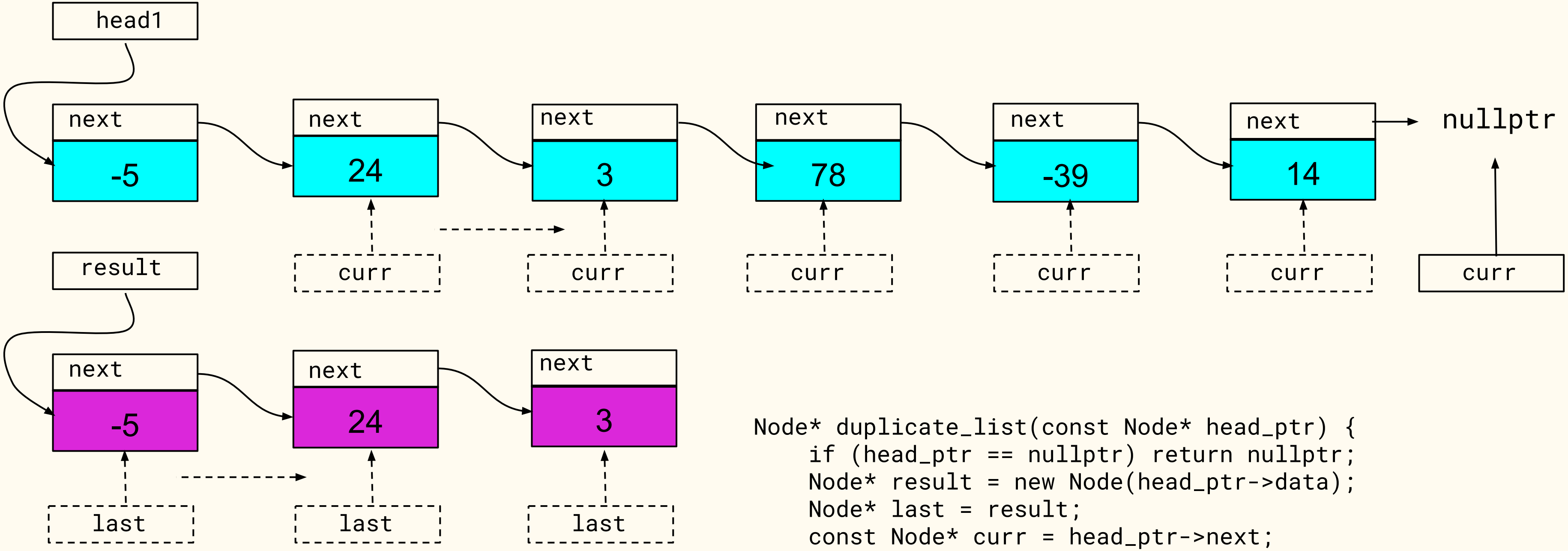
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {

    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

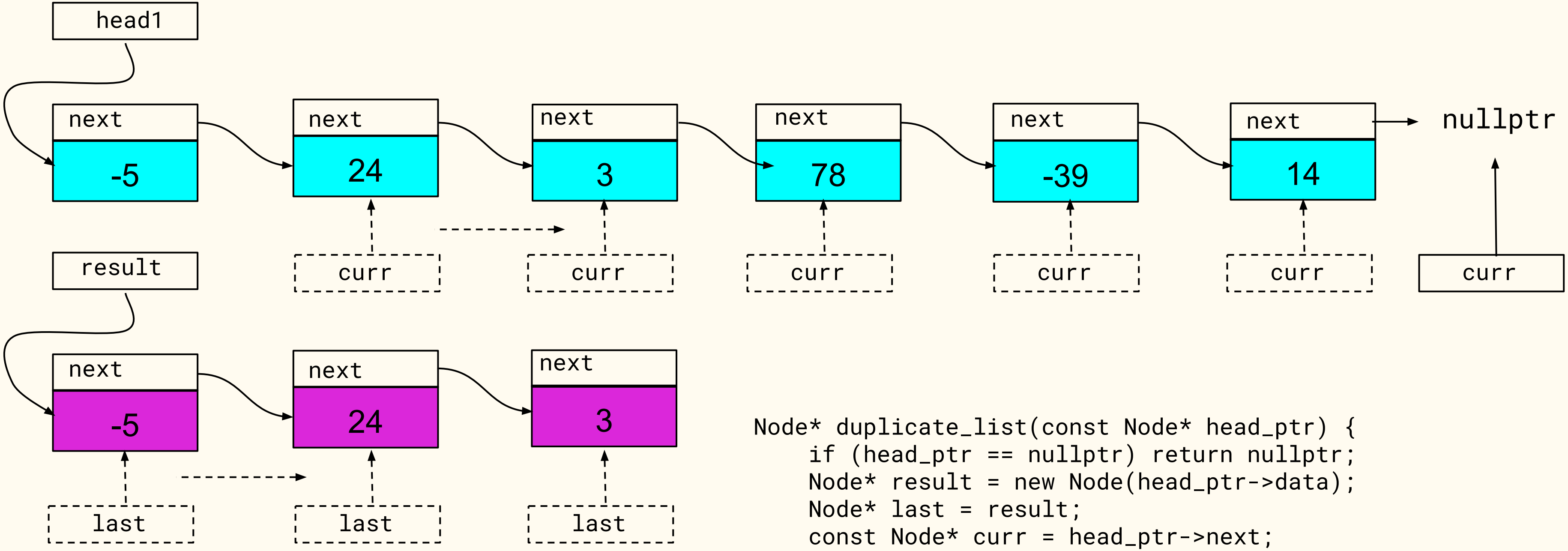
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = ___;
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

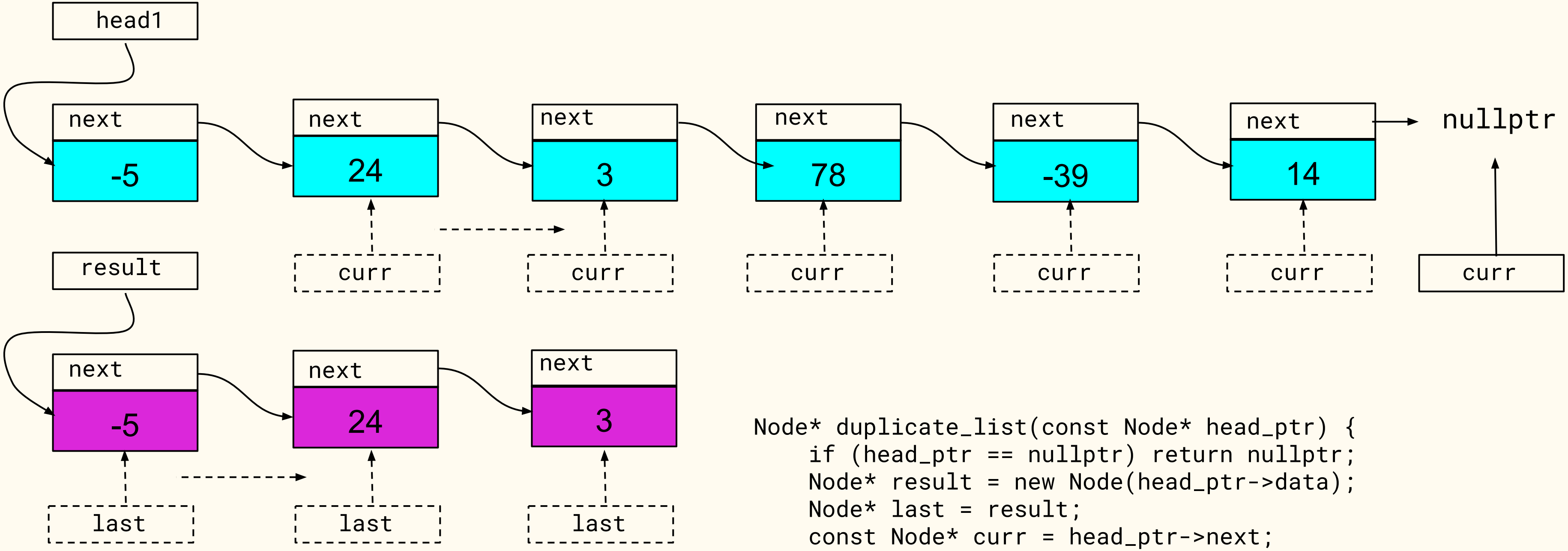
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = _62_;
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

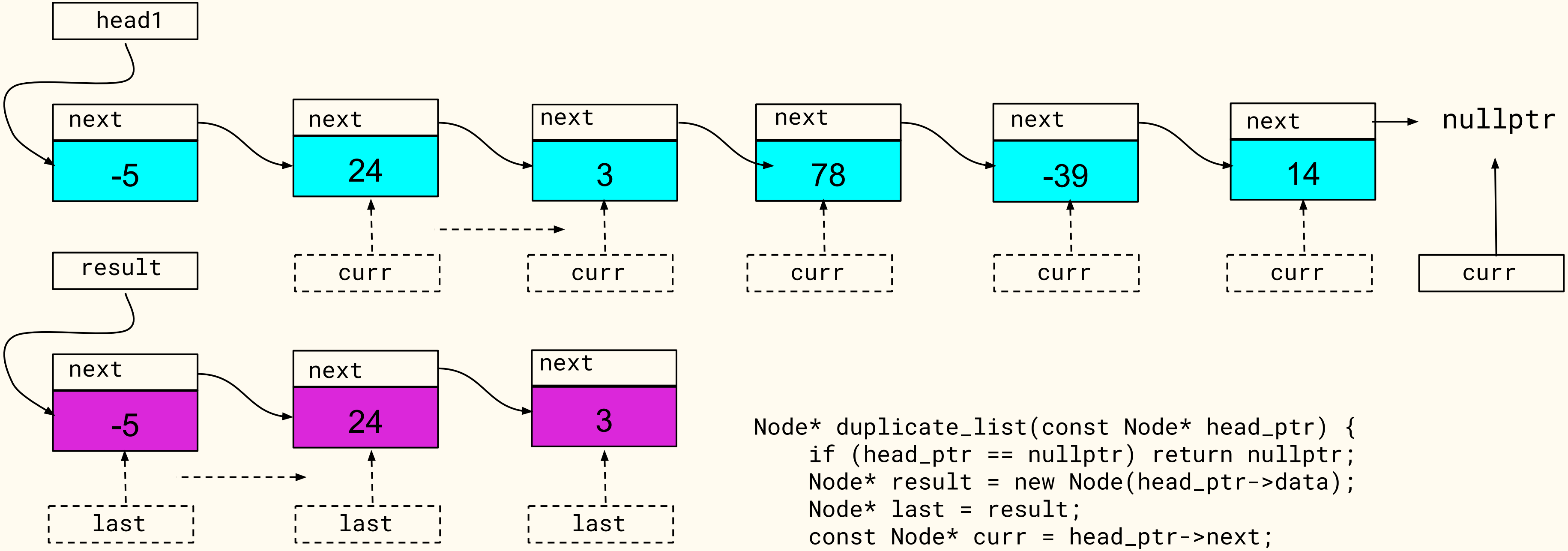
Which expression creates a copy of the current `Node` from the original list and returns its address?



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = _62_;
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

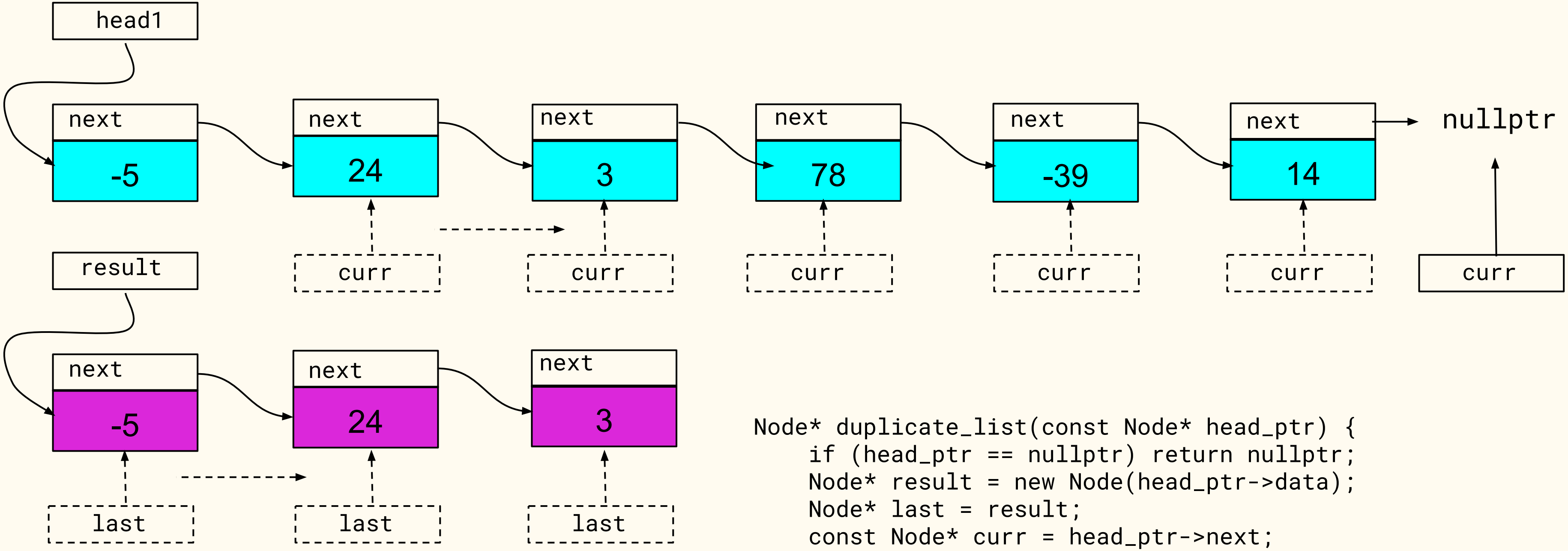
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

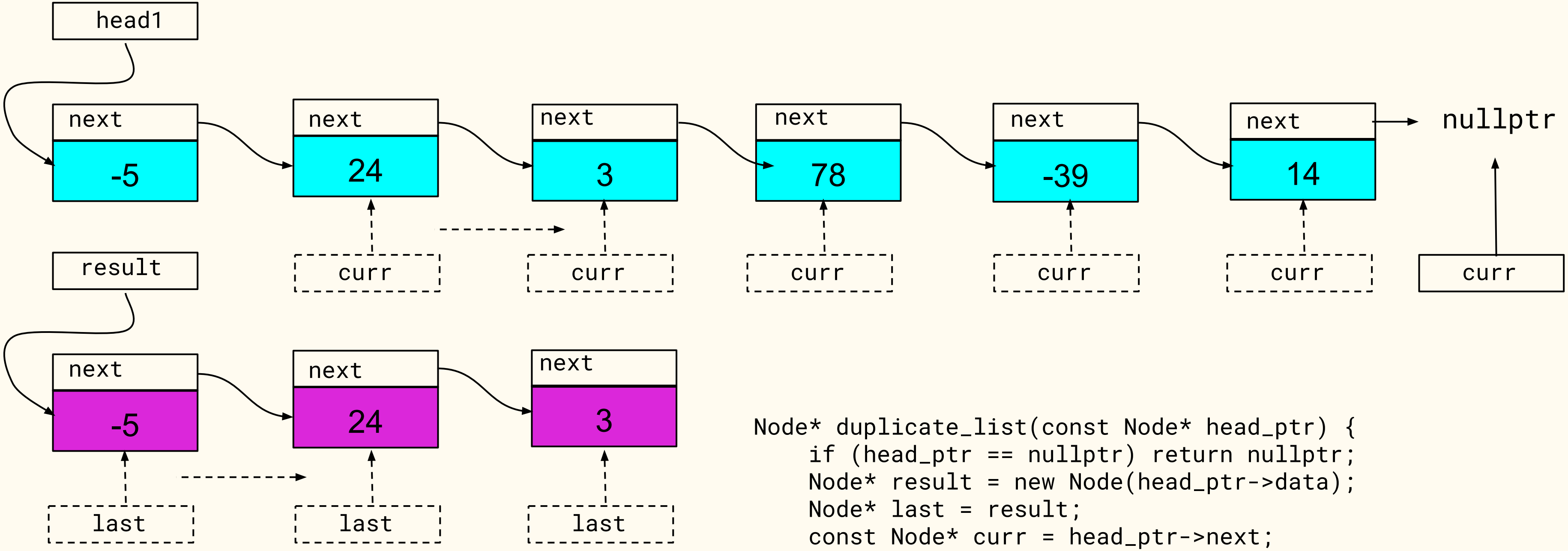
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        ---
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

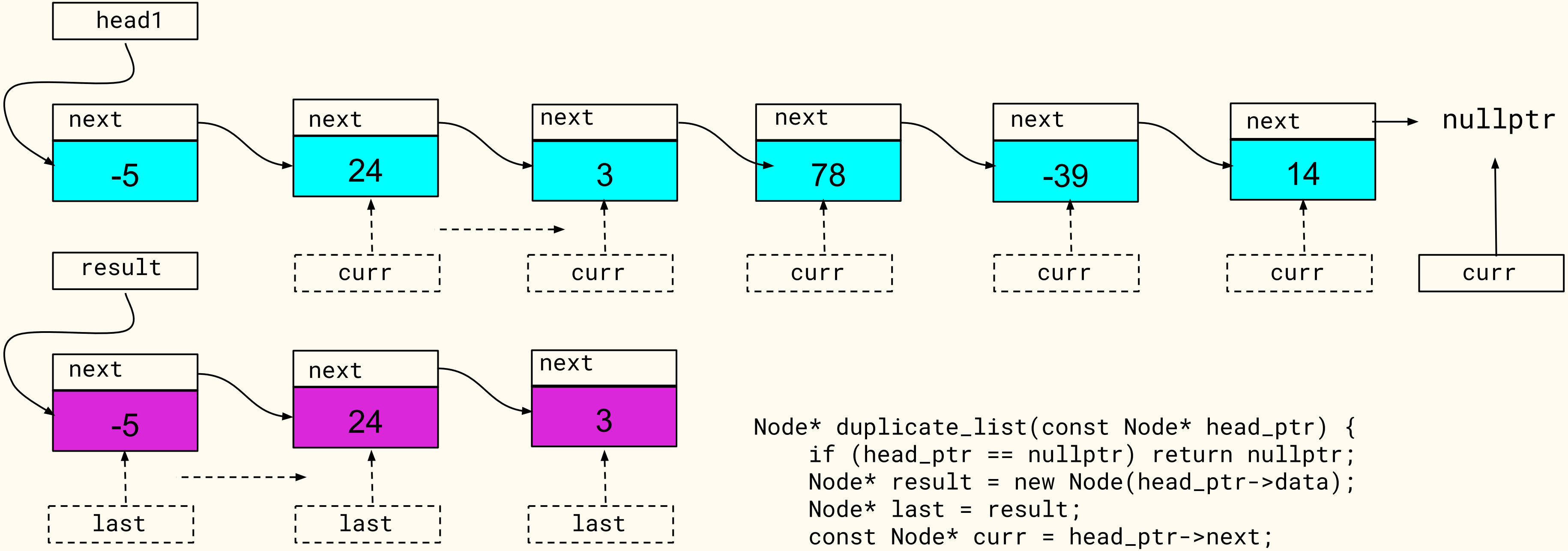
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = ___;
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

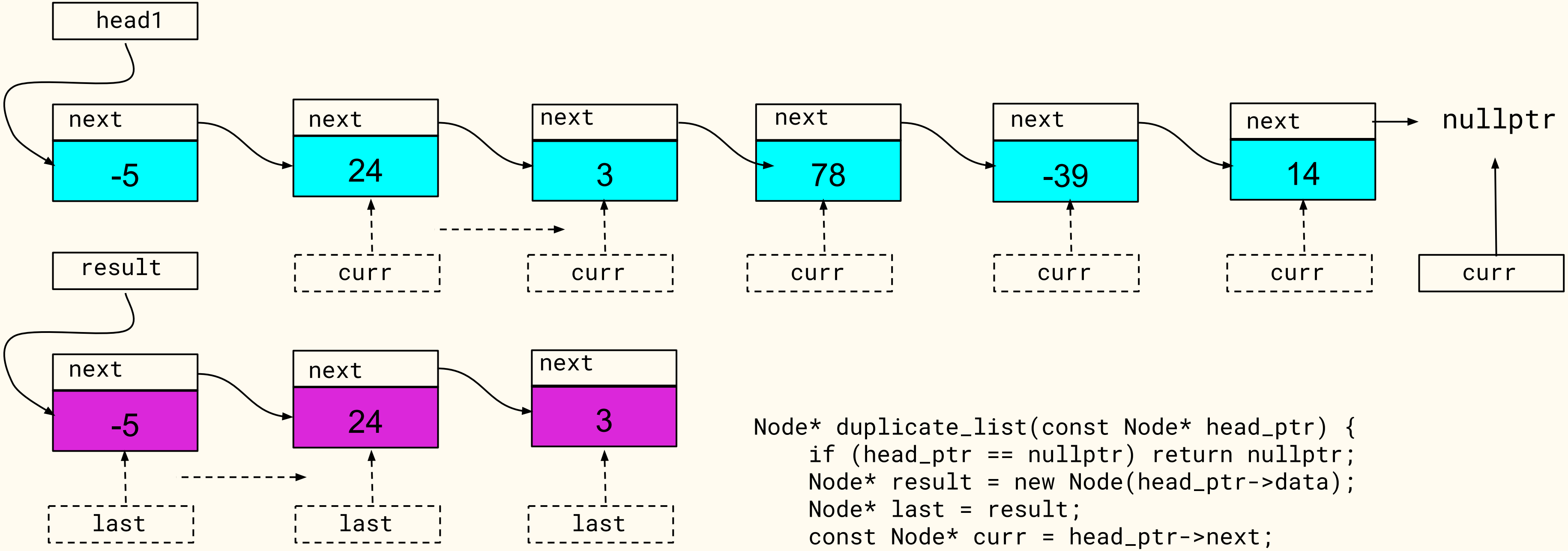
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = _61_;
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

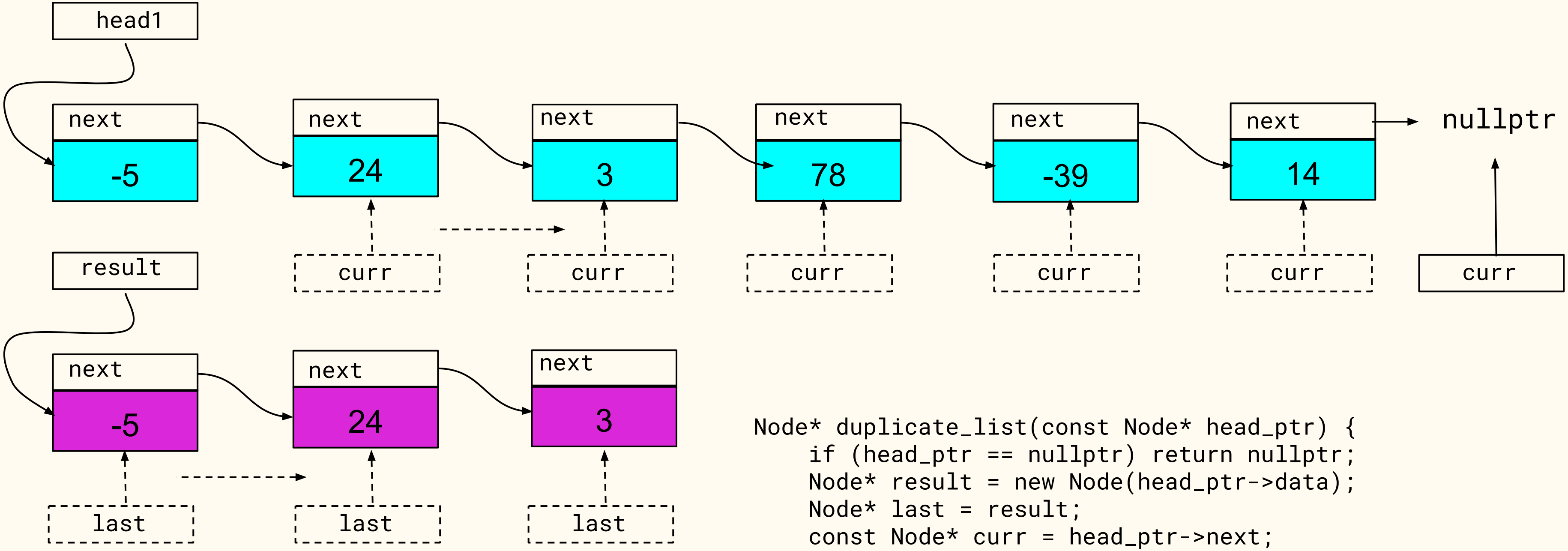
Which expression replaces blank #61 to advance the `last` pointer for the duplicate list?



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = _61_;
    }
}
```

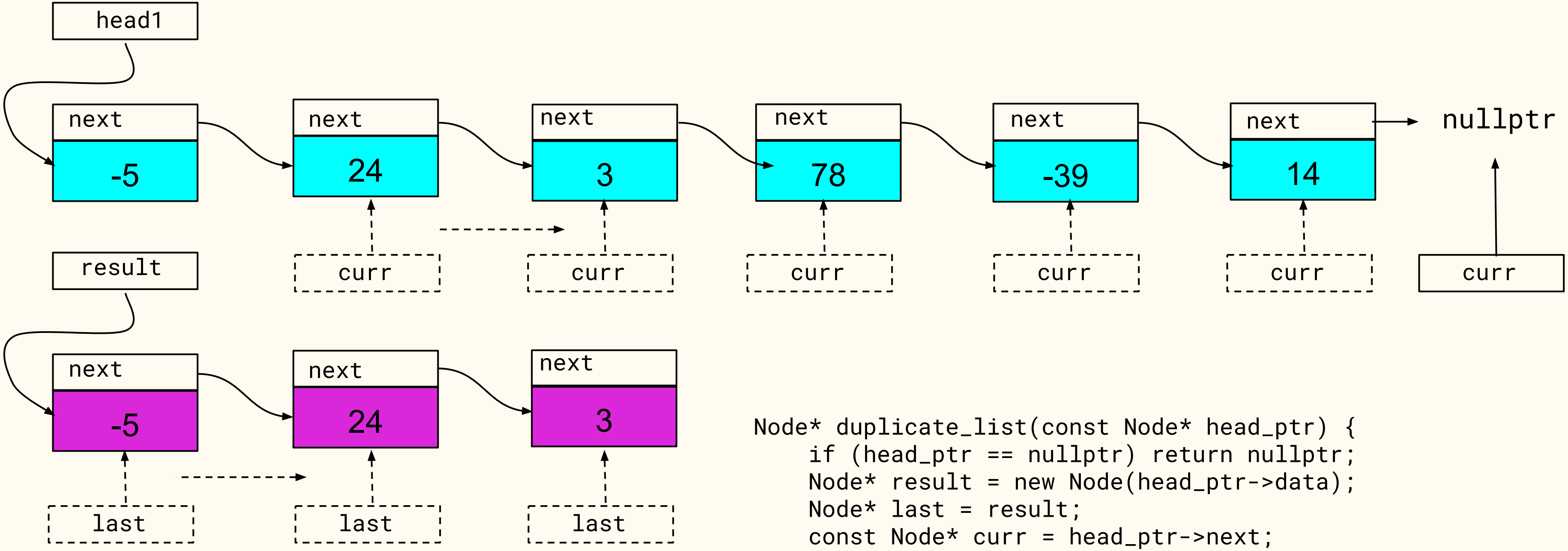
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = last->next;
    }
}
```

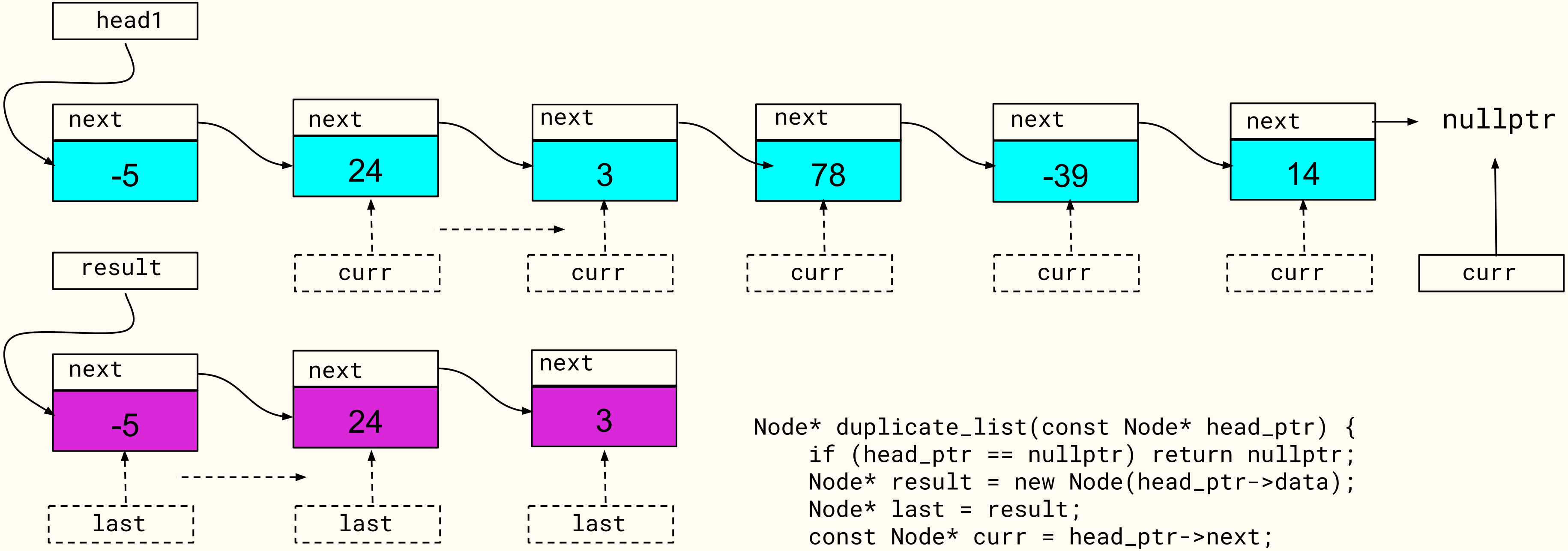
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = last->next;
        ---
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

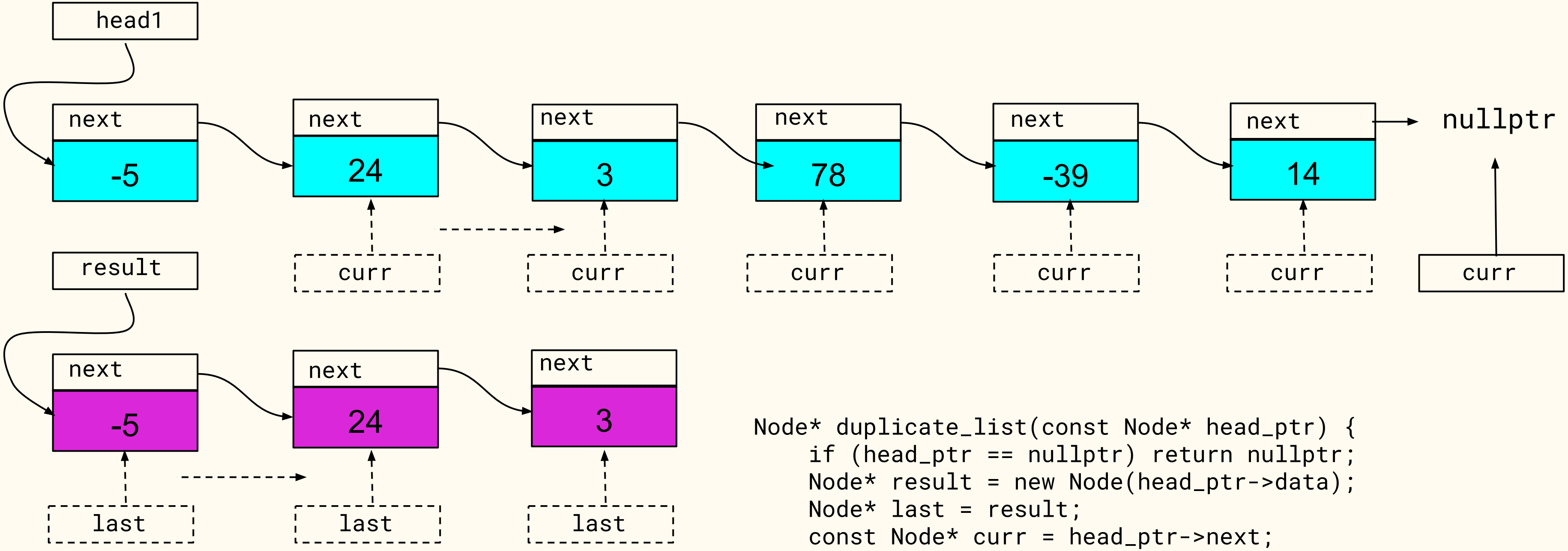
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = last->next;
        curr = ___;
    }
}
```

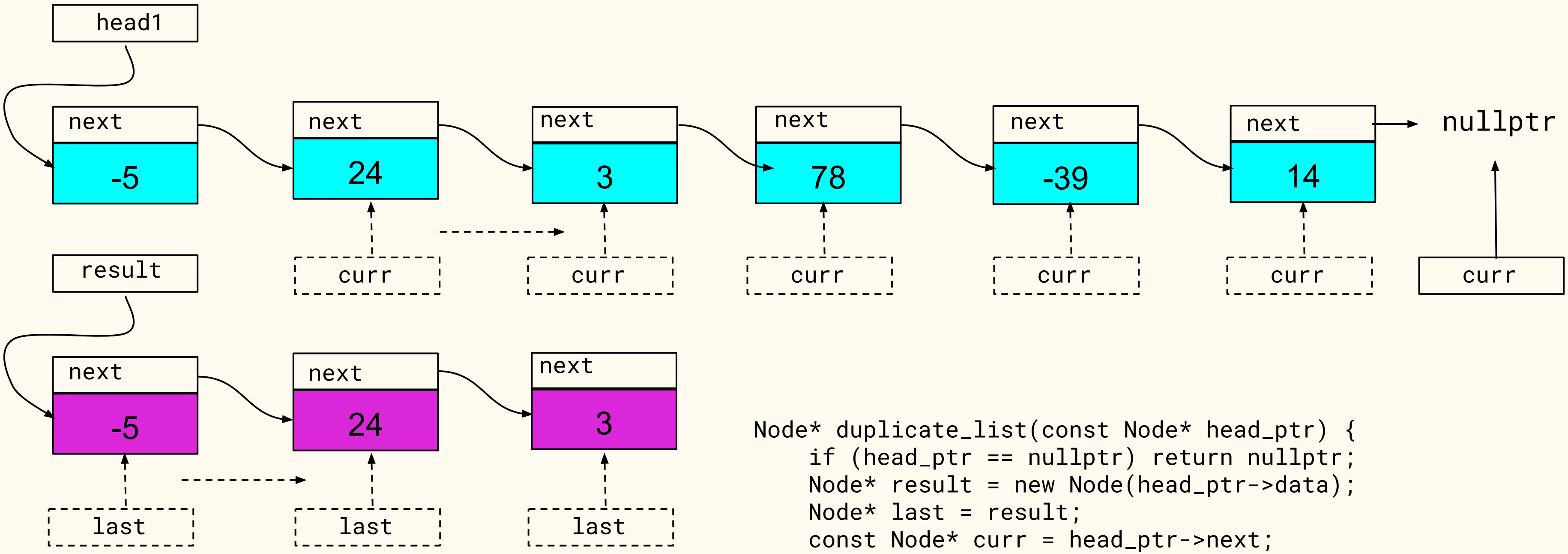
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = last->next;
        curr = _62_;
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

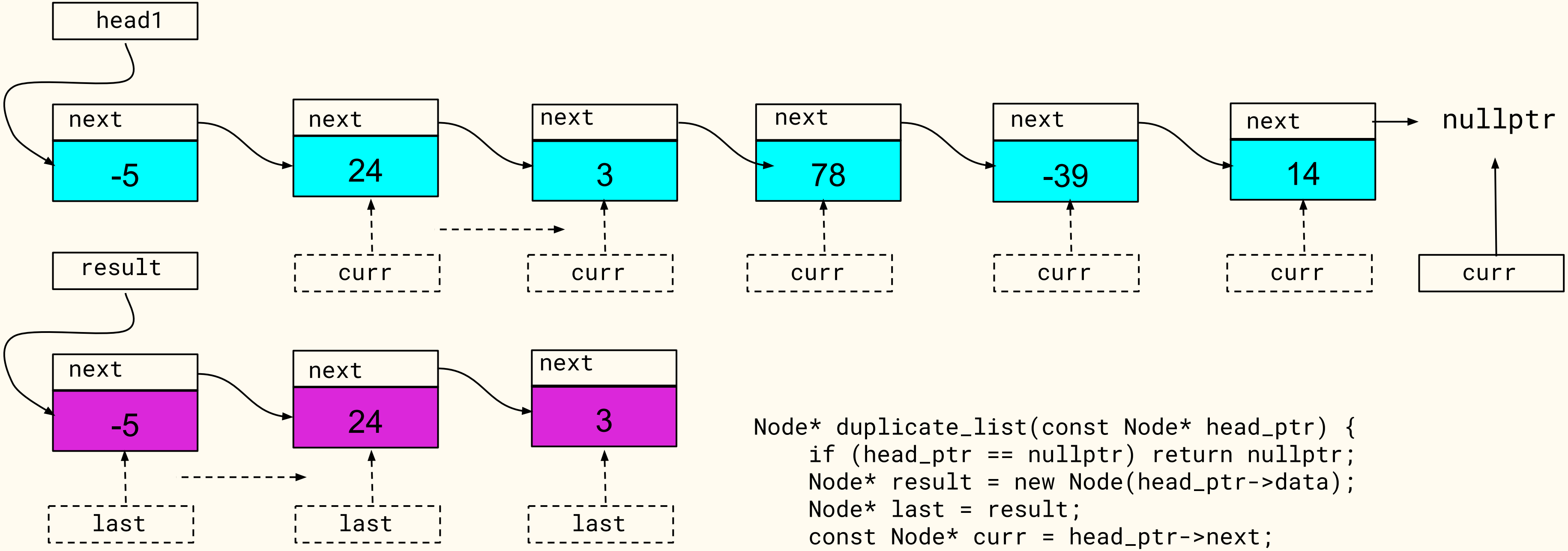
Which expression replaces blank #62 to advance the `curr` pointer for the original list?



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = last->next;
        curr = _62_;
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

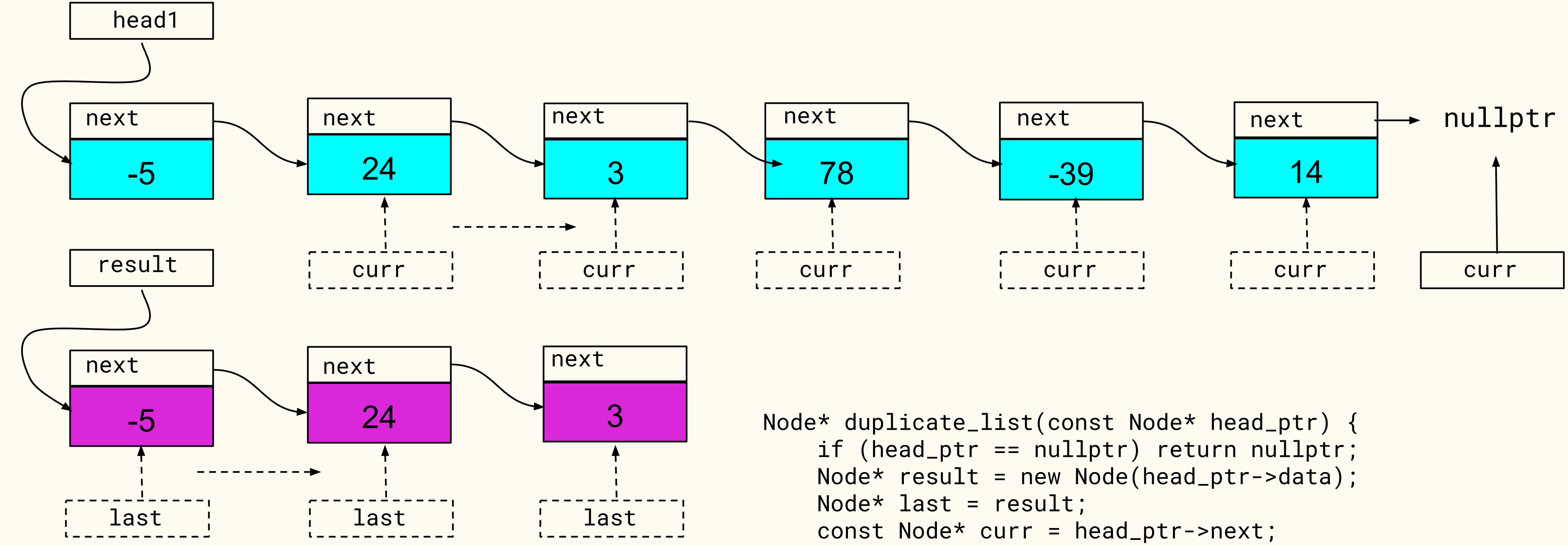
Duplicating a list



```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = last->next;
        curr = curr->next;
    }
    ---
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

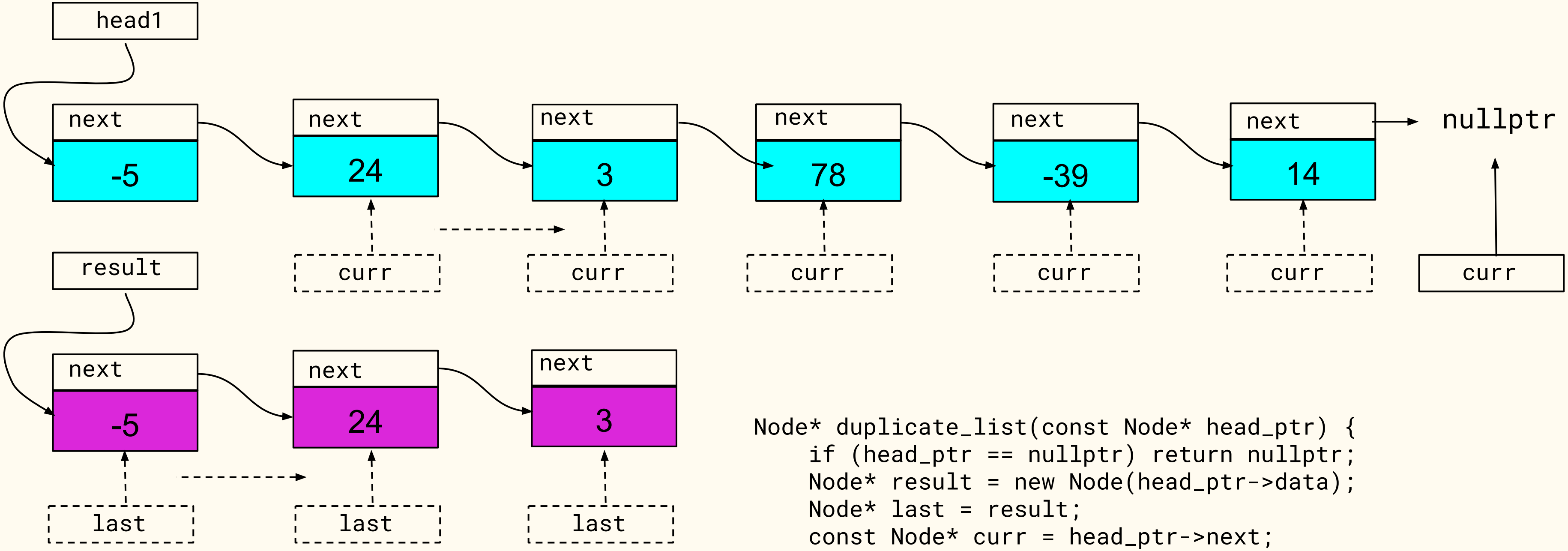
Duplicating a list



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = last->next;
        curr = curr->next;
    }
    _63_
}
```

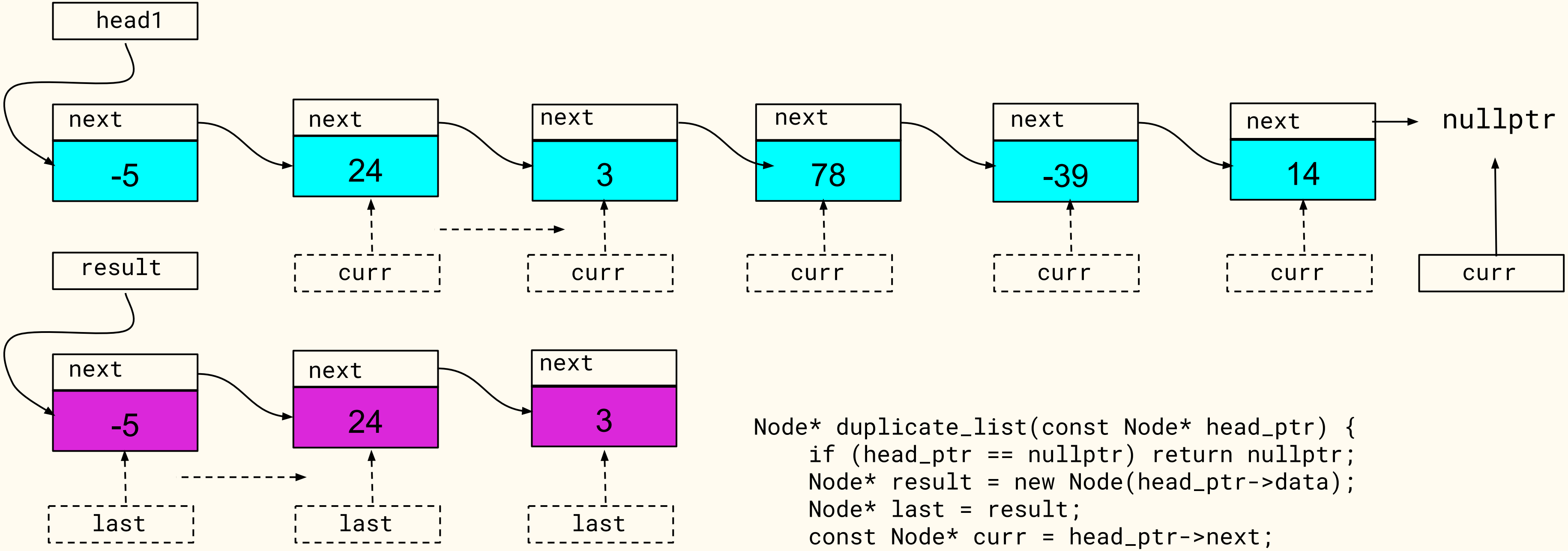
Which statement replaces blank #63 to return the address of the head `Node` of the duplicate list?



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = last->next;
        curr = curr->next;
    }
    _63_
}
```

Duplicating a list

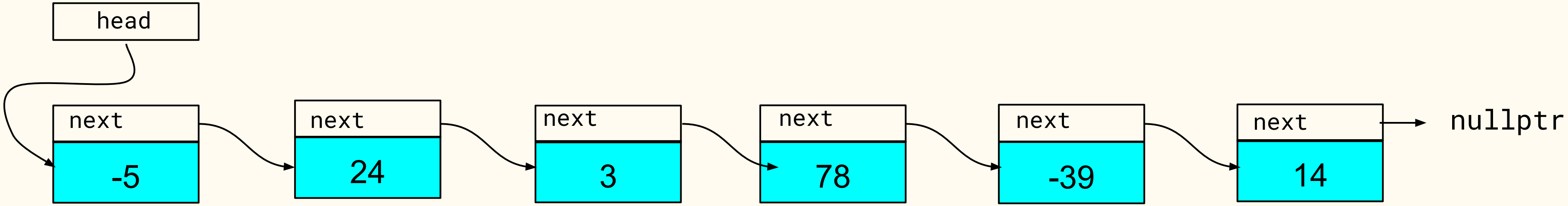


```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* duplicate_list(const Node* head_ptr) {
    if (head_ptr == nullptr) return nullptr;
    Node* result = new Node(head_ptr->data);
    Node* last = result;
    const Node* curr = head_ptr->next;

    while (curr != nullptr) {
        last->next = new Node(curr->data);
        last = last->next;
        curr = curr->next;
    }
    return result;
}
```

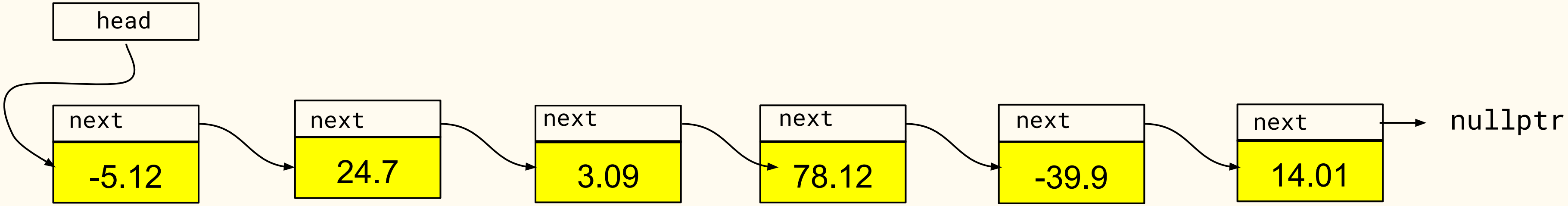
Background

Generic solutions to common programming problems



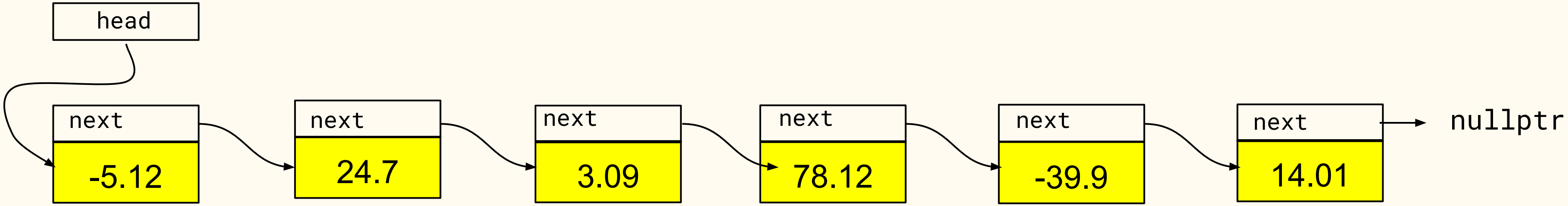
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

Generic solutions to common programming problems



```
struct ~~Node~~ {
    Node(~~int~~ data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    ~~int~~ data;
    ~~Node*~~ next;
};
```

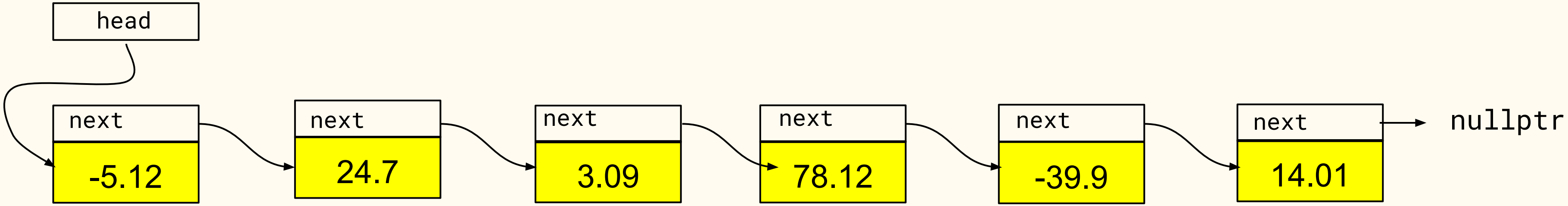
Generic solutions to common programming problems



```
struct DoubleNode {
    Node(double data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    double data;
    DoubleNode* next;
};
```

```
int calc_list_length(const ~~Node~~* head_ptr) {
    const ~~Node~~* ptr = head_ptr;
    int counter = 0;
    while (ptr != nullptr) {
        ++counter;
        ptr = ptr->next;
    }
    return counter;
}
```

Generic solutions to common programming problems



```
struct DoubleNode {
    Node(double data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    double data;
    DoubleNode* next;
};
```

```
int calc_list_length(const DoubleNode* head_ptr) {
    const DoubleNode* ptr = head_ptr;
    int counter = 0;
    while (ptr != nullptr) {
        ++counter;
        ptr = ptr->next;
    }
    return counter;
}
```

~~*Let's get started implementing each function...*~~

Generic solutions to common programming problems

- C++ supports *generic programming*
  - containers can be defined that only differ based on contained type
    - `std::vector<int>`
    - `std::vector<double>`
    - `std::vector<bool>`
    - etc
  - different containers implement similar functionality
    - default constructor
    - copy constructor
    - `size()` method
    - `empty()` method
    - etc
  - generic algorithms supporting different types
    - `sort()`
    - `max()`
    - `count()`
    - etc

Generic solutions to common programming problems

- Standard Template Library (STL) included with C++
  - provides useful container classes (including)
    - stack
    - queue
    - vector
    - deque
    - list
    - set
    - map
  - provides useful algorithms that work with containers
    - copying, searching, sorting, etc
    - `#include <algorithm>`
  - provides iterators
    - generalization of pointers

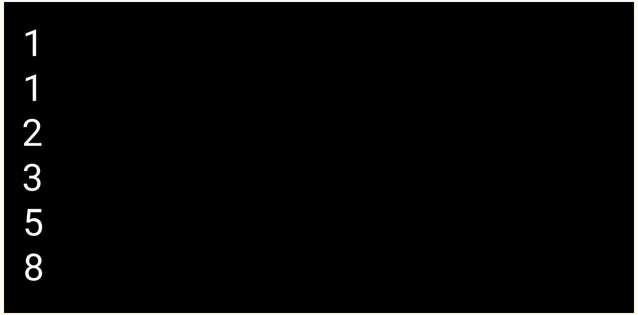
Generic solutions to common programming problems

- STL and generic programming deep topic
  - coverage will only scratch surface

Iterators

Traversing an array

```
int main() {
   const int SIZE = 6;
   int data[SIZE] = {1, 1, 2, 3, 5, 8};

   for (size_t i = 0; i < SIZE; ++i) {
       cout << data[i] << endl;
   }
}
```



```
1
1
2
3
5
8
```

Traversing an array (with pointers)

```
int main() {
   const int SIZE = 6;
   int data[SIZE] = {1, 1, 2, 3, 5, 8};

   for (int* ptr = data; ptr != data + SIZE; ++ptr) {
       cout << *ptr << endl;
   }
}
```

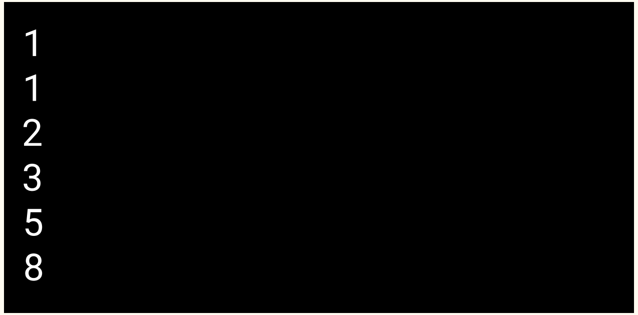
```
1
1
2
3
5
8
```

Traversing a list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_head_to_list(Node*& head_ptr, int data) {
    head_ptr  = new Node(data, head_ptr);
}
```

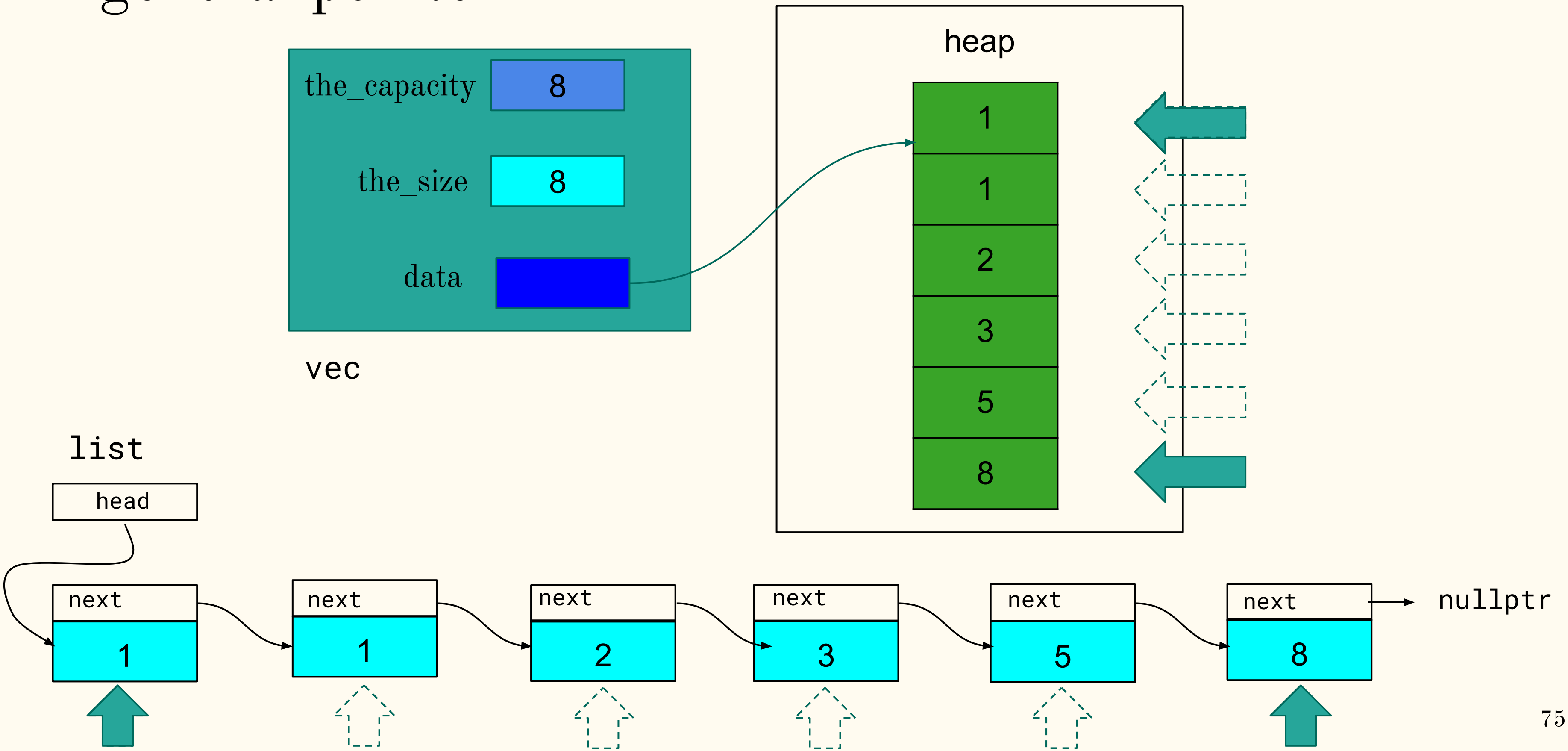
```
int main() {

    Node* head_ptr = new Node(8);
    add_head_to_list(head_ptr, 5);
    add_head_to_list(head_ptr, 3);
    add_head_to_list(head_ptr, 2);
    add_head_to_list(head_ptr, 1);
    add_head_to_list(head_ptr, 1);

    for (Node* ptr = head_ptr; ptr != nullptr; ptr = ptr->next) {
        cout << ptr->data << endl;
    }
}
```



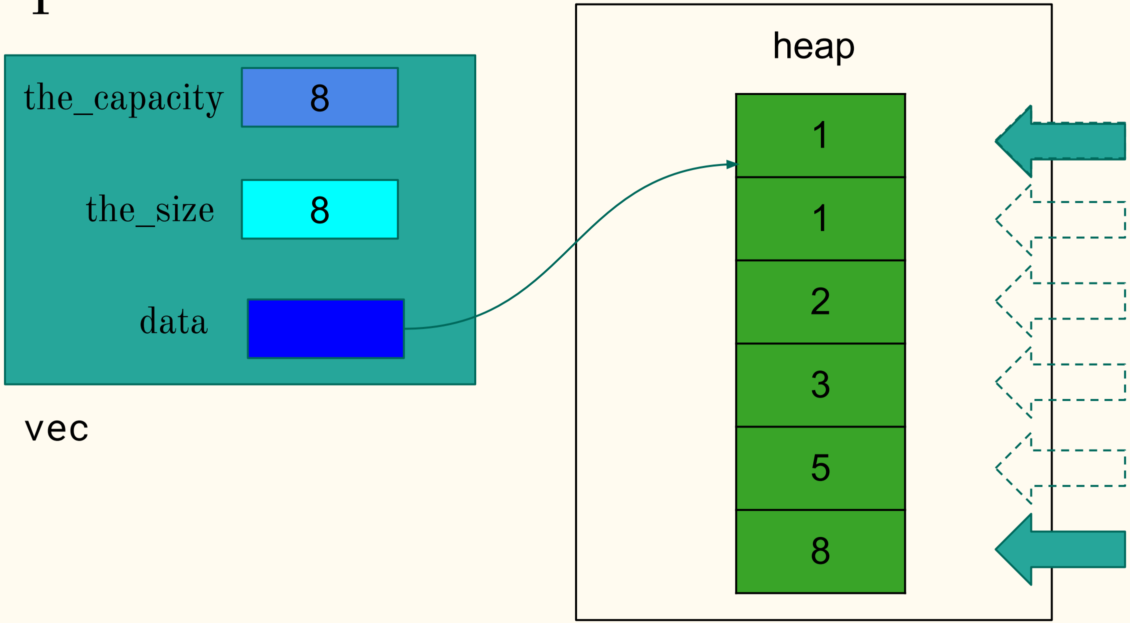
```
1
1
2
3
5
8
```

unified interface for traversing container would be nice...

A general pointer



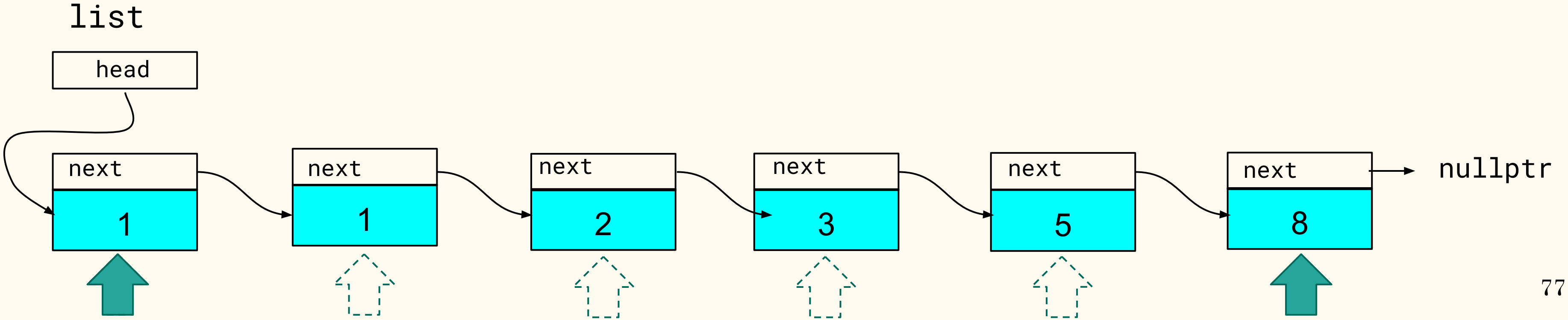
A general pointer



```
for (v_ptr = vec.begin(); v_ptr != vec.end(); ++v_ptr) {
    cout << *v_ptr << endl;
}
```

A general pointer

```
for (l_ptr = list.begin(); l_ptr != list.end(); ++l_ptr) {
    cout << *l_ptr << endl;
}
```



A general pointer

need operator to access value "pointed to"

```
for (l_ptr = list.begin(); l_ptr != list.end(); ++l_ptr) {
    cout << *l_ptr << endl;
}
```

need a type

need to define beginning

need to define end

need way to point to "next" element

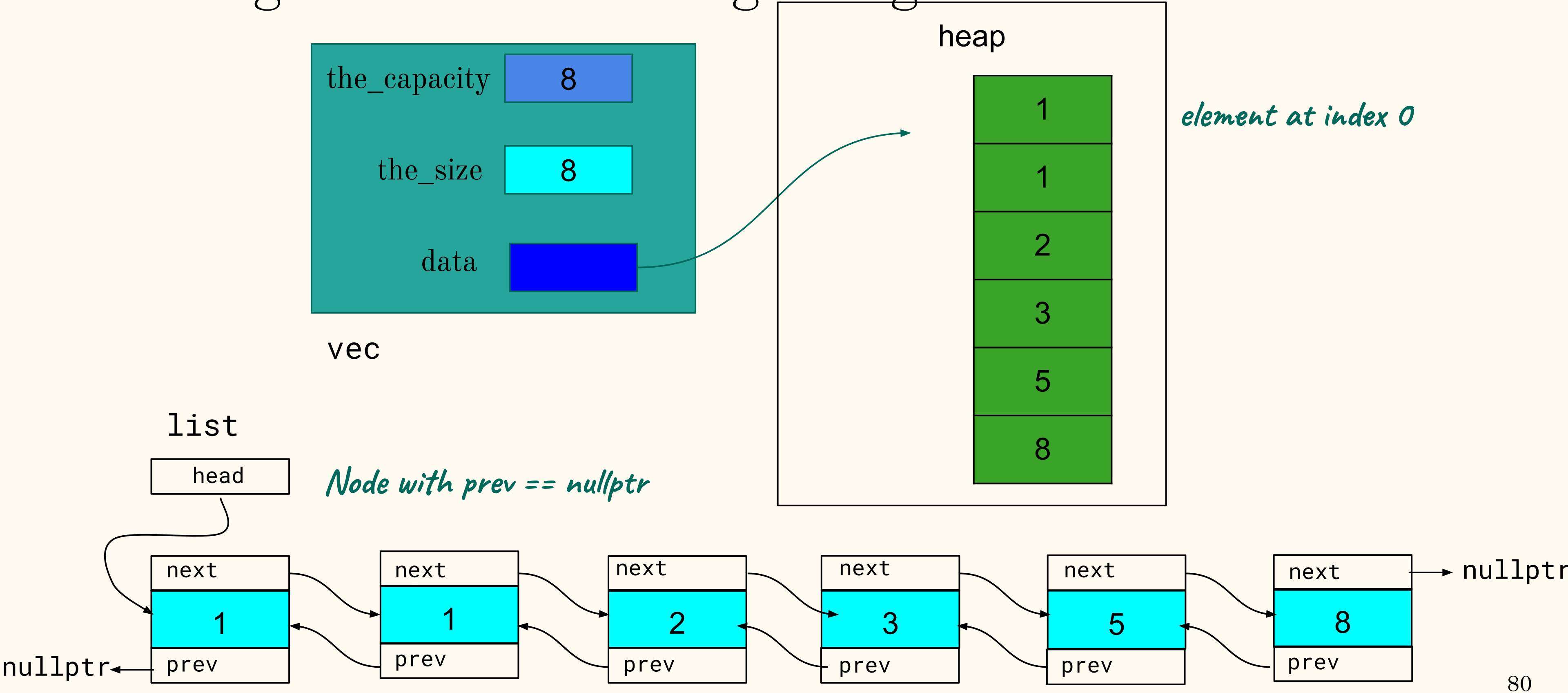
```
for (v_ptr = vec.begin(); v_ptr != vec.end(); ++v_ptr) {
    cout << *v_ptr << endl;
}
```

need operator to access value "pointed to"

Iterator - a general pointer

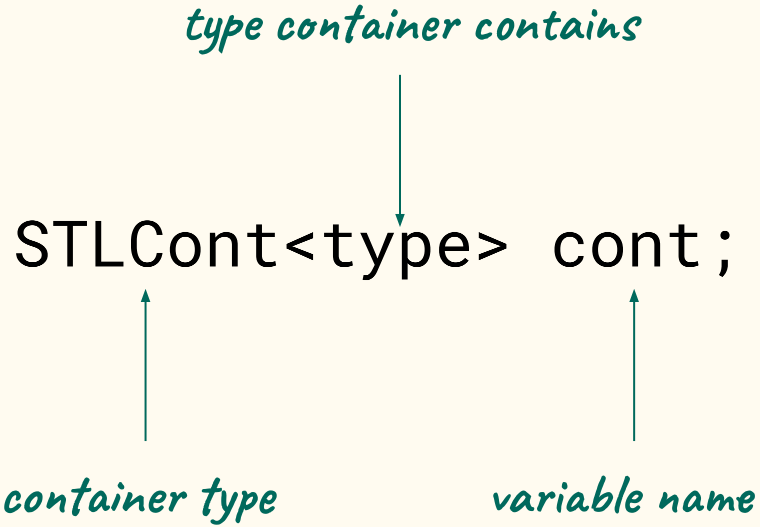
- STL defines generalization of pointers (an *iterator*)
- enables shared behavior for pointers to elements in different container types
- exist in both `const` and non-`const` varieties

Defining the containers beginning



Determining the containers beginning

Note: conceptual code
(will not compile)



type container contains

```
STLCont<type> cont;
```

container type

variable name

e.g. vector<int> vec;

Determining the containers beginning

Note: conceptual code
(will not compile)

```
STLCont<type> cont;

iter = cont.begin();
```

$$\texttt{iter = }\underbrace{\texttt{cont.begin()}}_{\text{points to first element in container}}\texttt{;}$$

Determining the containers beginning

Note: conceptual code
(will not compile)

```
STLCont<type> cont;

?? iter = cont.begin();
```

type?

points to first
element in container

Iterator types

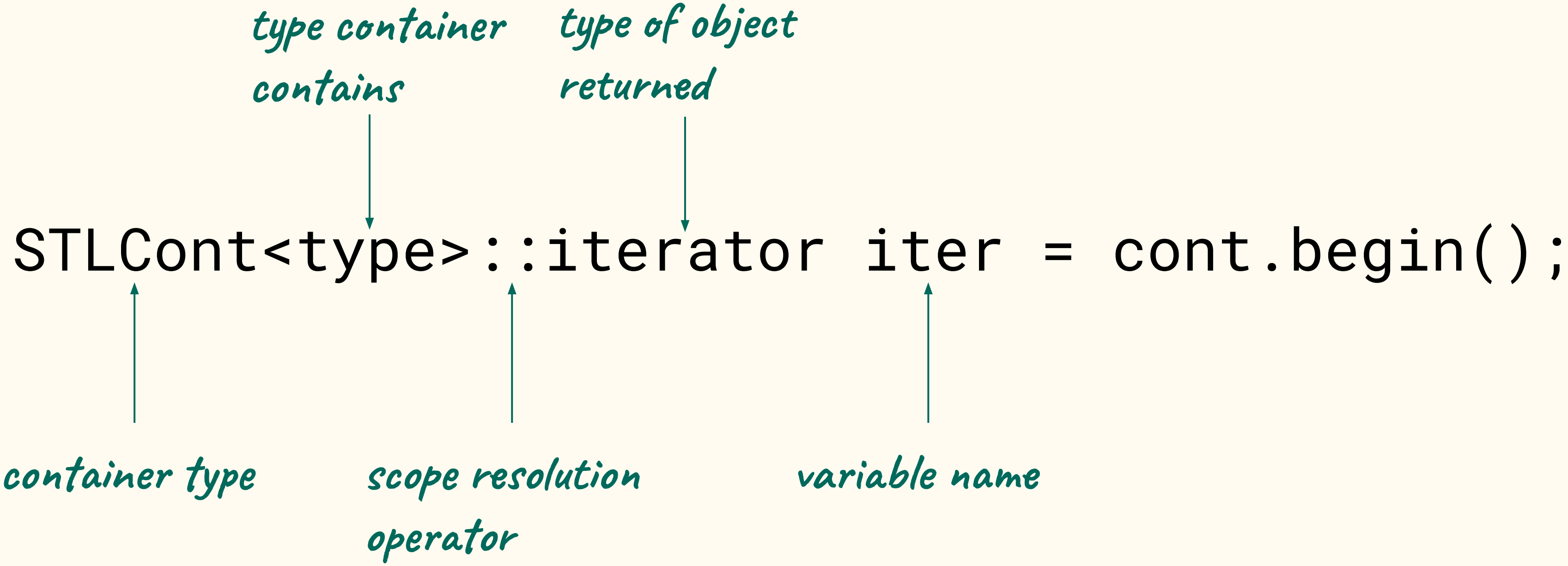
- each container provides an iterator
- iterator type available via container (using scope resolution operator)

Note: conceptual code (will not compile)

```
STLCont<type> cont;

STLCont<type>::iterator iter = cont.begin();
```

Iterator types

- each container provides an iterator
- iterator type available via container (using scope resolution operator)



```
STLCont<type>::iterator iter = cont.begin();
```

type container contains

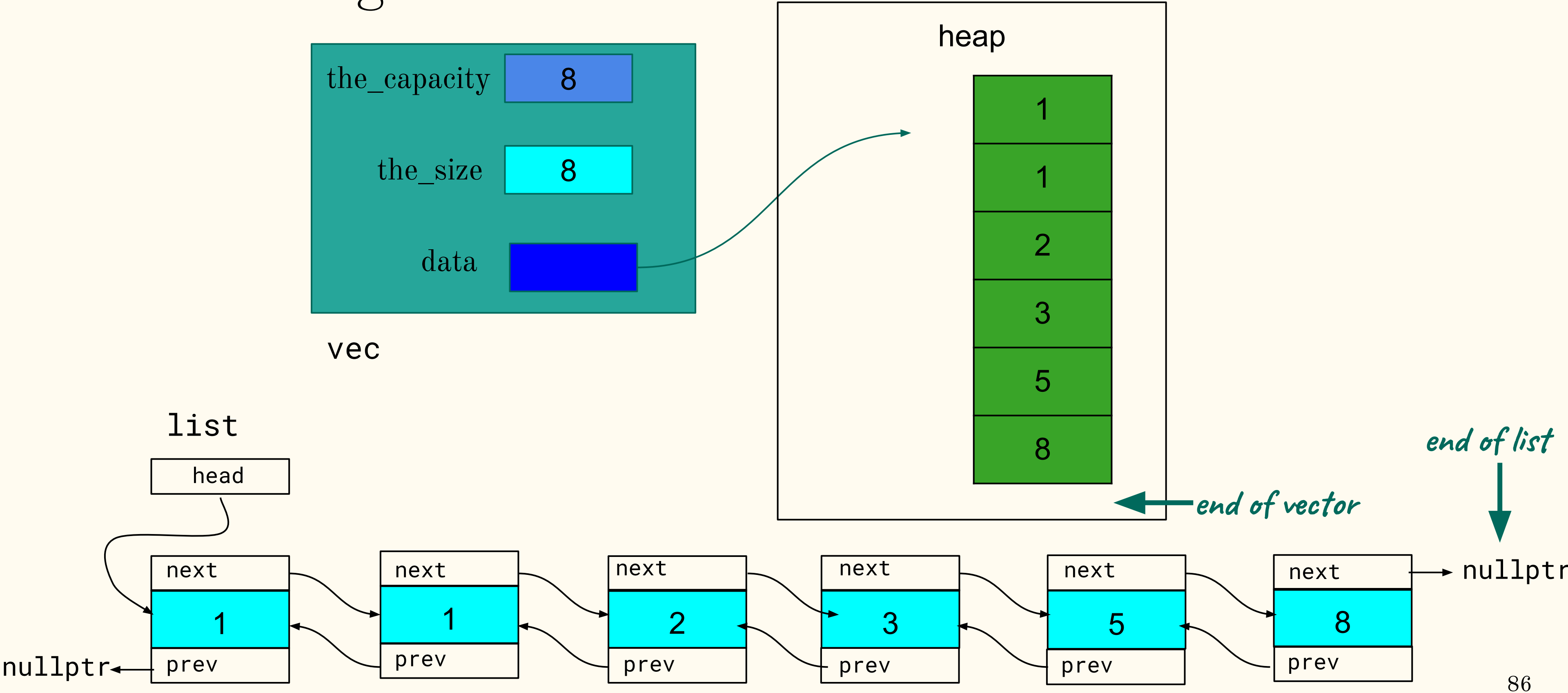
type of object returned

container type

scope resolution operator

variable name

Determining the containers end



Determining the containers end

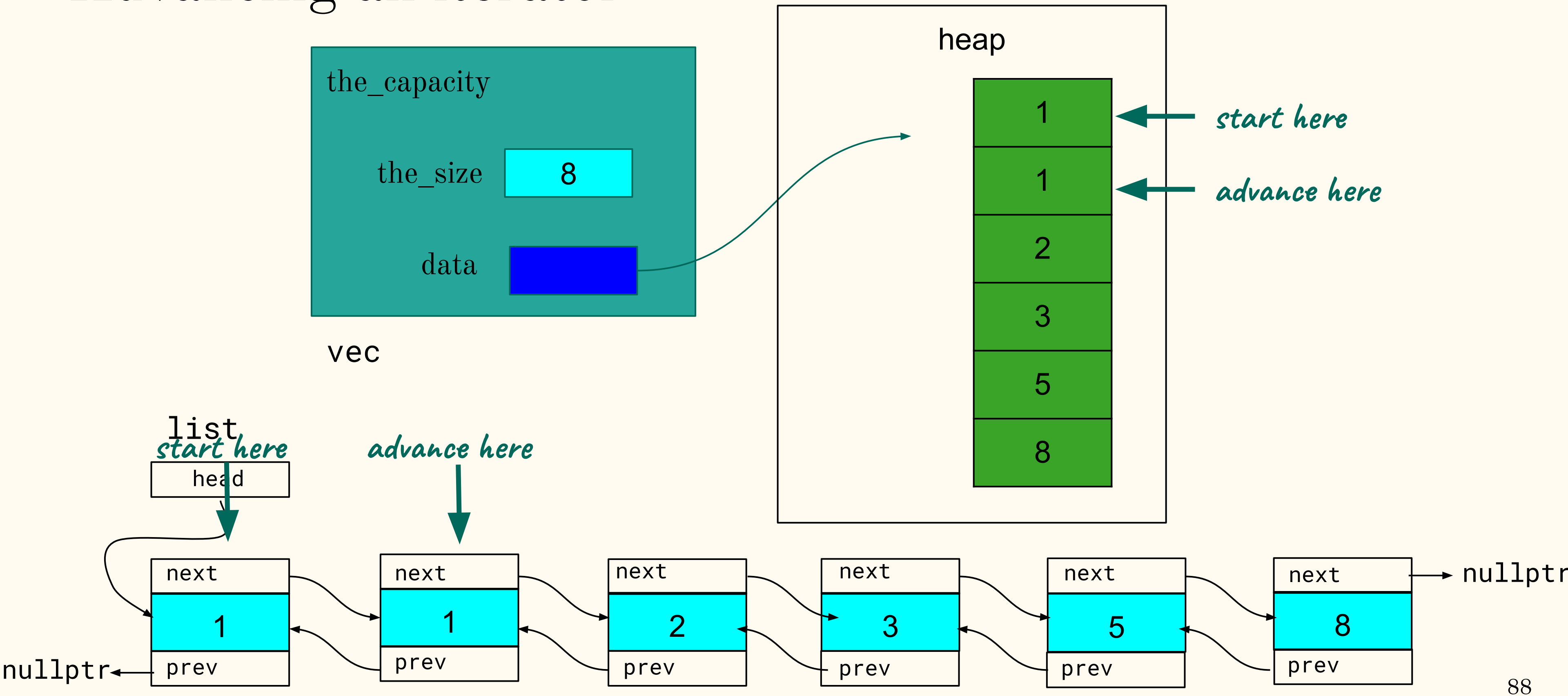
- `end()` returns an iterator that cannot be dereferenced
- useful in conditions to test that end of container reached

Note: conceptual code
(will not compile)

```
STLCont<type> cont;

STLCont<type>::iterator iter = cont.end();
```

returns an iterator pointing to "end" of container

Advancing an iterator



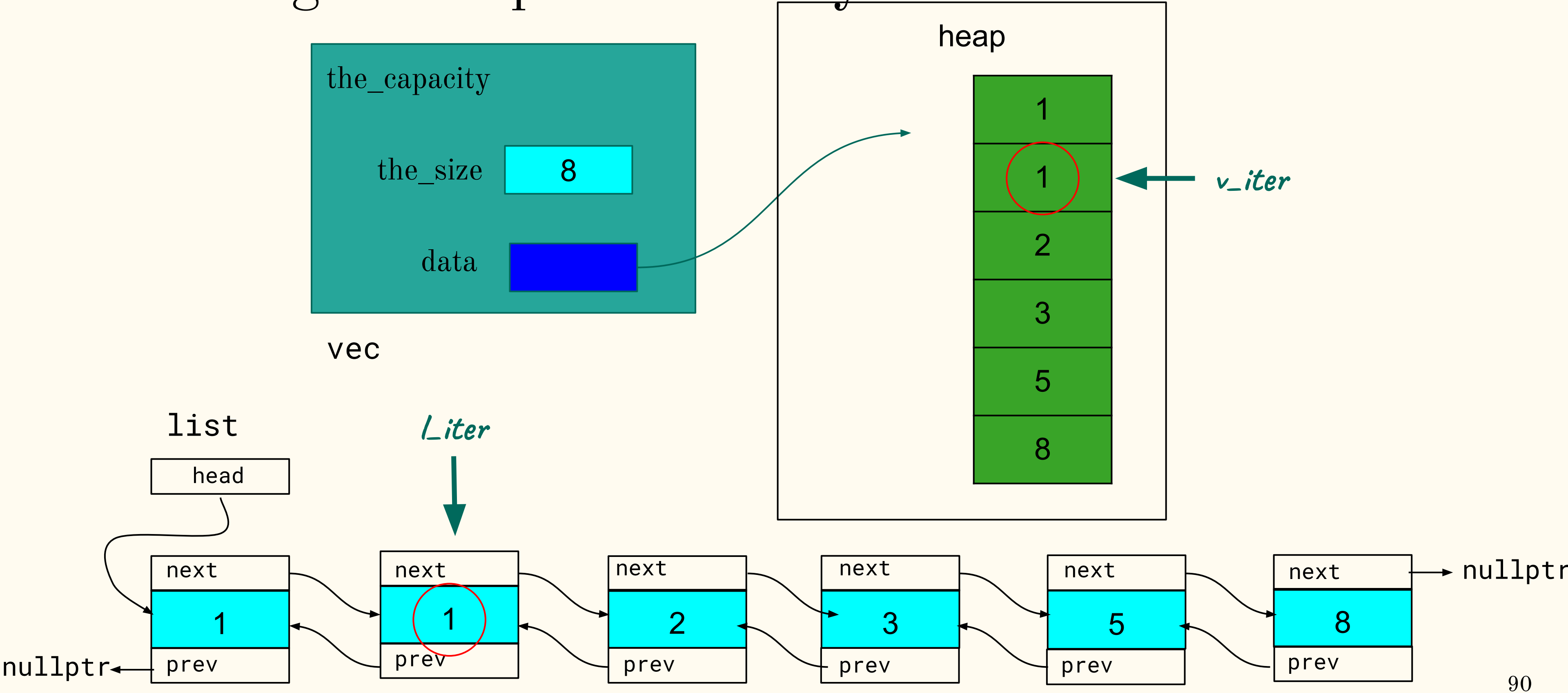
Advancing an iterator

```
STLCont<type> cont;

STLCont<type>::iterator iter = cont.begin();

++iter;
```

iter points to element second from beginning

Note: conceptual code (will not compile)

Accessing value "pointed to" by iterator



Accessing value "pointed to" by iterator

```
STLCont<type> cont;

STLCont<type>::iterator iter = cont.begin();

++iter;

type elem = *iter;

type other;

*iter = other;
```

Note: conceptual code
(will not compile)

elem assigned second
element in container

second element now
same value as other

const iterators

```
#include <vector>
using namespace std;

int main() {
    vector<int> vec{1, 1, 2, 3, 5, 8};
    vector<int>::iterator iter = vec.begin();

    ++iter;

    int elem = *iter;

    int other = 6;

    *iter = other;
}
```

vec => {1, 6, 2, 3, 5, 8}

`const` iterators

```
#include <vector>
using namespace std;

int main() {
     vector<int> vec{1, 1, 2, 3, 5, 8};
     vector<int>::iterator iter = vec.begin();

     ++iter;

     int elem = *iter;

     int other = 6;

     *iter = other;
}
```

need to change type of iterator

What if we don't want to allow modification through iterator??

const iterators

```
#include <vector>
using namespace std;

int main() {
     vector<int> vec{1, 1, 2, 3, 5, 8};
     vector<int>::const_iterator iter = vec.begin();

     ++iter;

     int elem = *iter;  ✓

     int other = 6;

     *iter = other;  compilation error
}
```

Review of `Vector` class

CS2124 `Vector` constructor

```
class Vector {
public:
    Vector(size_t size = 0, int value = 0) {
        the_size = size;
        the_capacity = size;
        data = new int[size];
        for (size_t i = 0; i < the_size; ++i) {
            data[i] = value;
        }
    }

private:
    int* data;
    size_t the_size, the_capacity;
};
```

CS2124 `Vector` destructor

```
class Vector {
public:
    Vector(size_t size = 0, int value = 0) {
        the_size = size;
        the_capacity = size;
        data = new int[size];
        for (size_t i = 0; i < the_size; ++i) {
            data[i] = value;
        }
    }

    ~Vector() { delete [] data; }
private:
    int* data;
    size_t the_size, the_capacity;
};
```

CS2124 `Vector` copy constructor

```
class Vector {
public:
    ...

    Vector(const Vector& rhs) {
        the_size = rhs.the_size;
        the_capacity = rhs.the_capacity;
        data = new int[the_capacity];
        for (size_t i = 0; i < the_size; ++i) {
            data[i] = rhs.data[i];
        }
    }
private:
    int* data;
    size_t the_size, the_capacity;
};
```

CS2124 Vector assignment operator

```
class Vector {
public:
    ...
    Vector& operator=(const Vector& rhs) {
        if (this != &rhs) {
            delete [] data;
            the_size = rhs.the_size;
            the_capacity = rhs.the_capacity;
            data = new int[the_capacity];
            for (size_t i = 0; i < the_size; ++i) {
                data[i] = rhs.data[i];
            }
        }
        return *this;
    }
    ...
};
```

CS2124 Vector push_back() method

```
class Vector {
public:
    ...
    void push_back(int val) {
        if (the_capacity == 0) {
            delete [] data;
            ++the_capacity;
            data = new int[the_capacity];
        }
        if (the_size == the_capacity) {
            int* new_data = new int[2 * the_capacity];
            for (size_t i = 0; i < the_size; ++i) {
                new_data[i] = data[i];
            }
            delete [] data;
            data = new_data;
            the_capacity *= 2;
        }
        data[the_size] = val;
        ++the_size;
    }
private:
    int* data;
    size_t the_size, the_capacity;
};
```

CS2124 Vector other methods

```
class Vector {
public:
    ...
    size_t size() const { return the_size; }

    int operator[](size_t i) const { return data[i]; }

    int& operator[](size_t i) { return data[i]; }

    void clear() { the_size = 0; }

    void pop_back() { --the_size; }
private:
    int* data;
    size_t the_size, the_capacity;
};
```

CS2124 Vector begin() and end() methods

```
class Vector {
public:
    ...

    int* begin() { return data; }
    int* end() { return data + the_size; }

    const int* begin() const { return data; }
    const int* end() const { return data + the_size; }


private:
    int* data;
    size_t the_size, the_capacity;
};
```