**SRS Setup**
**Login: student.turningtechnologies.com**
**Session ID: 20220425<A|D>**

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

# Generic Algorithms Recursion

—

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

# Agenda

- Generic algorithms
- Recursion

# Generic algorithms

# Swapping values

```
int main() {
   int a_val = 17, b_val = 42;
   cout << a_val << ' ' << b_val << endl;

   a_val, b_val = b_val, a_val;
   cout << a_val << ' ' << b_val << endl;
}
```

```
17 42
```

# TurningPoint

**SRS Setup**
**Login: student.turningtechnologies.com**
**Session ID: 20220425<A|D>**

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

# What is output by the second output statement?

```
int main() {
    int a_val = 17, b_val = 42;
    cout << a_val << ' ' << b_val << endl;

    a_val, b_val = b_val, a_val;
    cout << a_val << ' ' << b_val << endl;
}
```

# Swapping values

```
int main() {
    int a_val = 17, b_val = 42;
    cout << a_val << ' ' << b_val << endl;

    a_val, b_val = b_val, a_val;
    cout << a_val << ' ' << b_val << endl;
}
```

```
17 42
17 42
```

# Swapping values

```
int main() {
    int a_val = 17, b_val = 42;
    cout << a_val << ' ' << b_val << endl;

    a_val, b_val = b_val, a_val;
    cout << a_val << ' ' << b_val << endl;
}
```

*works in Python...*

*...not in C++*

```
17 42
17 42
```

# Swapping values

*assignment evaluated first*

```
a_val, b_val = b_val, a_val;
```

*comma (,) is an operator*

*lowest operator precedence*

# Swapping values

```
void swap_int(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int a_val = 17, b_val = 42;
    cout << a_val << ' ' << b_val << endl;

    a_val, b_val = b_val, a_val;
    cout << a_val << ' ' << b_val << endl;
}
```

# Swapping values

```
void swap_int(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int a_val = 17, b_val = 42;
    cout << a_val << ' ' << b_val << endl;

    swap_int(a_val, b_val);
    cout << a_val << ' ' << b_val << endl;
}
```

```
17 42
42 17
```

# Swapping values

```
void swap_int(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

*What about doubles?*

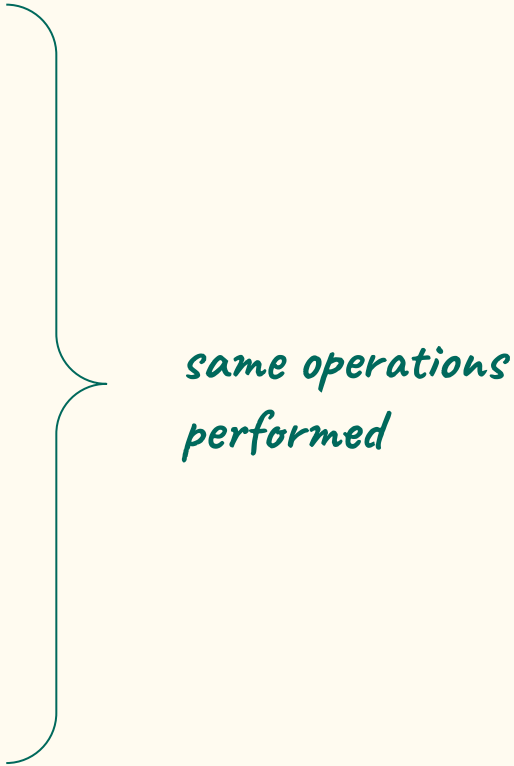
# Swapping values

```
void swap_int(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void swap_double(double& a, double& b) {
    double temp = a;
    a = b;
    b = temp;
}
```

*What about strings?*

# Swapping values

```
void swap_int(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void swap_double(double& a, double& b) {
    double temp = a;
    a = b;
    b = temp;
}

void swap_string(string& a, string& b) {
    string temp = a;
    a = b;
    b = temp;
}
```



# Swapping values

```
...
#include <utility>
using namespace std;

int main() {
    int a_val = 17, b_val = 42;
    cout << a_val << ' ' << b_val << endl;

    swap_int(a_val, b_val);
    cout << a_val << ' ' << b_val << endl;
}
```

# Swapping values

```
...
#include <utility>
using namespace std;

int main() {
    int a_val = 17, b_val = 42;
    cout << a_val << ' ' << b_val << endl;

    swap(a_val, b_val);
    cout << a_val << ' ' << b_val << endl;
}
```

*std::swap() defined here*

```
17 42
42 17
```

# Swapping values

```
...
#include <utility>
#include<string>
using namespace std;

int main() {
    string a_str = "string a", b_str = "string b";
    cout << a_str << ', ' << b_str << endl;

    swap(a_str, b_str);
    cout << a_str << ', ' << b_str << endl;
}
```

How??

```
string a, string b
string b, string a
```

# A templated swap function

```
void swap_int(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void swap_double(double& a, double& b) {
    double temp = a;
    a = b;
    b = temp;
}

void swap_string(string& a, string& b) {
    string temp = a;
    a = b;
    b = temp;
}
```

```
void swap_general() { }
```

# A templated swap function

```
void swap_int(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void swap_double(double& a, double& b) {
    double temp = a;
    a = b;
    b = temp;
}

void swap_string(string& a, string& b) {
    string temp = a;
    a = b;
    b = temp;
}
```

```
void swap_general(T& a, T& b) { }
```

# A templated swap function

```
void swap_int(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void swap_double(double& a, double& b) {
    double temp = a;
    a = b;
    b = temp;
}

void swap_string(string& a, string& b) {
    string temp = a;
    a = b;
    b = temp;
}
```

```
template <typename T>
void swap_general(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

# Swapping values

```
template <typename T>
void swap_general(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int a_str = "string a", b_str = "string b";
    cout << a_str << ', ' << b_str << endl;

    swap(a_str, b_str);
    cout << a_str << ', ' << b_str << endl;
}
```

# Swapping values

```
template <typename T>
void swap_general(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int a_str = "string a", b_str = "string b";
    cout << a_str << ', ' << b_str << endl;

    swap_general(a_str, b_str);
    cout << a_str << ', ' << b_str << endl;
}
```

```
string a, string b
string b, string a
```

# Generic linear search

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


int main() {
    char array[] = "Alan Turing";
    const int len = 11;

    vector<char> vc(array, array + len);
}
```

range constructor

- requires specifying *half-open* range
  - first argument included
  - values **preceding** second argument included
- enables initialization from container of different type

type container contains

beginning of half-open range

```
STLCont<type> cont(start, end);
```

Note: conceptual code (will not compile)

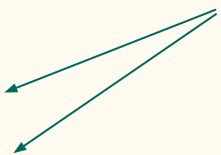
container type

variable name

end of half-open range

# Generic linear search

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

int main() {
   char array[] = "Alan Turing";
   const int len = 11;

   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());
}
```

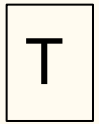


**range constructor**

- requires specifying *half-open* range
- enables initialization from container of different type

# Generic linear search

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




?

"Alan Turing"

```
int main() {
   char array[] = "Alan Turing";
   const int len = 11;

   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());

}
```

# Generic linear search

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

char* my_find() { }
```

```
int main() {
   char array[] = "Alan Turing";
   const int len = 11;

   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());

}
```

# Generic linear search

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

char* my_find(char* start, char* stop, char target) { }
```

```
int main() {
   char array[] = "Alan Turing";
   const int len = 11;

   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());

}
```

# Generic linear search

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

char* my_find(char* start, char* stop, char target) {
    for (char* p = start; p < stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}

int main() {
   char array[] = "Alan Turing";
   const int len = 11;

   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());
   my_find(array, array + len, 'q');
}
```

*returns* **array + len** *(an address)*

# Generic linear search

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

char* my_find(char* start, char* stop, char target) {
    for (char* p = start; p < stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}

int main() {
   char array[] = "Alan Turing";
   const int len = 11;

   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());
   cout << *my_find(array, array + len, 'T') << endl;
}
```

```
% g++ --std=c++17 stl.cpp -o stl.o
% ./stl.o
T
```

# Generic linear search

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

list<char>::iterator my_find() { }




int main() {
   char array[] = "Alan Turing";
   const int len = 11;

   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());

}
```

# Generic linear search

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

list<char>::iterator my_find(list<char>::iterator start,
                             list<char>::iterator stop, char target) {









}
int main() {
   char array[] = "Alan Turing";
   const int len = 11;

   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());

}
```

# Generic linear search

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

list<char>::iterator my_find(list<char>::iterator start,
                             list<char>::iterator stop, char target) {
    for (list<char>::iterator p = start; p != stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
int main() {
   char array[] = "Alan Turing";
   const int len = 11;

   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());

}
```

# Generic linear search

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

list<char>::iterator my_find(list<char>::iterator start,
                             list<char>::iterator stop, char target) {
    for (list<char>::iterator p = start; p != stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
int main() {
   char array[] = "Alan Turing";
   const int len = 11;

   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());
   cout << *my_find(lc.begin(), lc.end(), 'T') << endl;
}
```

```
% g++ --std=c++17 stl.cpp -o stl.o
% ./stl.o
T
```

# Generic linear search

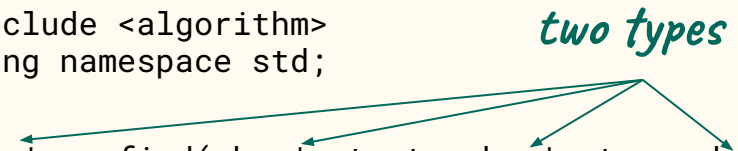
```
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
```

```
int main() {
   char array[] = "Alan Turing";
   const int len = 11;
   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());

   find(vc.begin(), vc.end(), 'T');
}
```

```
sort(first, last)

count(first, last, val)

copy (first, last, result)

much more...
```

# Generic linear search

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


char* my_find(char* start, char* stop, char target) {
    for (char* p = start; p < stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
int main() {
   char array[] = "Alan Turing";
   const int len = 11;
   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());
}
```




Let's turn this into a generic function...

# Generic linear search

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

char* my_find(char* start, char* stop, char target) {
    for (char* p = start; p < stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
int main() {
   char array[] = "Alan Turing";
   const int len = 11;
   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());
}
```

*Let's turn this into a generic function...*

# Generic linear search

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


T my_find(T start, T stop, ~~char~~ target) {
    for (T p = start; p < stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
int main() {
   char array[] = "Alan Turing";
   const int len = 11;
   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());
}
```

*Let's turn this into a generic function...*

# Generic linear search

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


T my_find(T start, T stop, U target) {
    for (T p = start; p < stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
int main() {
   char array[] = "Alan Turing";
   const int len = 11;
   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());
}
```

!= works for pointers and iterators

Let's turn this into a generic function...

# Generic linear search

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

// need to indicate types are template parameters
T my_find(T start, T stop, U target) {
    for (T p = start; p != stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
int main() {
   char array[] = "Alan Turing";
   const int len = 11;
   vector<char> vc(array, array + len);
   list<char> lc(vc.begin(), vc.end());
}
```

*Let's turn this into a generic function...*

# Generic linear search

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

template <typename T, typename U>
T my_find(T start, T stop, U target) {
    for (T p = start; p != stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
int main() {
    char array[] = "Alan Turing";
    const int len = 11;
    vector<char> vc(array, array + len);
    list<char> lc(vc.begin(), vc.end());
    if (my_find(lc.begin(), lc.end(), 'q') == lc.end())
        cout << "not found" << endl;
}
```

*Let's turn this into a generic function...*

```
my_find(array, array + len, 'T');
my_find(vc.begin(), vc.end(), 'T');
my_find(lc.begin(), lc.end(), 'T');
```

*All available*

# Functions as arguments

# Function arguments

```
int main() {

    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    for (int* p = arr; p != arr + len; ++p) {
        if (*p % 2 != 0) {
            cout << *p << endl;
            break;
        }
    }
}
```

# What do we know about the value output in the if statement?

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    for (int* p = arr; p != arr + len; ++p) {
        if (*p % 2 != 0) {
            cout << *p << endl;
            break;
        }
    }

}
```

# Function arguments

```
bool is_odd(int num) { return num % 2 != 0; }


int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    for (int* p = arr; p != arr + len; ++p) {
        if (*p % 2 != 0) {
            cout << *p << endl;
            break;
        }
    }

}
```

```
% g++ --std=c++17 stl.cpp -o stl.o
% ./stl.o
23
```

# Function arguments

```
bool is_odd(int num) { return num % 2 != 0; }


int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    for (int* p = arr; p != arr + len; ++p) {
        if (is_odd(*p)) {
            cout << *p << endl;
            break;
        }
    }

}
```

STL provides find function helpful for this task

```
% g++ --std=c++17 stl.cpp -o stl.o
% ./stl.o
23
```

# Function arguments

```
bool is_odd(int num) { return num % 2 != 0; }


int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;



    int* f_odd_ptr = find_if(arr, arr + len, is_odd);

}
```

STL provides find function helpful for this task

function passed as argument

can accept any predicate (function returning bool)

# Function arguments

```
template <typename T, typename U>
T my_find(T start, T stop, U target) {
    for (T p = start; p != stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
```

# Function arguments

```
template <typename T, typename U>
T my_find_if(T start, T stop, U target) {
    for (T p = start; p != stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
```

# Function arguments

```
template <typename T, typename U>
T my_find_if(T start, T stop, U pred) {
    for (T p = start; p != stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}
```

# Function arguments

```
template <typename T, typename U>
T my_find_if(T start, T stop, U pred) {
    for (T p = start; p != stop; ++p) {
        if (pred(*p)) {
            return p;
        }
    }
    return stop;
}
```

# Function arguments

```
bool is_odd(int num) { return num % 2 != 0; }
```

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;


    int* f_odd_ptr = my_find_if(arr, arr + len, is_odd);

}
```

not limited to finding first odd integer

# Function arguments

```
bool is_odd(int num) { return num % 2 != 0; }
```

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;


    int* f_odd_ptr = my_find_if(arr, arr + len, ___);

}
```

not limited to finding first odd integer

# Function arguments

```
bool is_odd(int num) { return num % 2 != 0; }

struct IsEven {
    bool operator() (int num) const { return num % 2 == 0; }
};
```

functor -- functions as a type

defines behavior when calling the functor

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    IsEven is_even;
    int* ptr = my_find_if(arr, arr + len, ___);

}
```

not limited to finding first odd integer

# Function arguments

```
bool is_odd(int num) { return num % 2 != 0; }

struct IsEven {
    bool operator() (int num) const { return num % 2 == 0; }
};
```

functor -- functions as a type

defines behavior when calling the functor

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    IsEven is_even;
    int* ptr = my_find_if(arr, arr + len, is_even);

}
```

not limited to finding first odd integer

# Function arguments

```
struct IsEven {
    bool operator() (int num) const { return num % 2 == 0; }
};

template <typename T, typename U>
T my_find_if(T start, T stop, U pred) {
    for (T p = start; p != stop; ++p) {
        if (pred(*p)) {
            return p;
        }
    }
    return stop;
}

int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    IsEven is_even;
    int* ptr = my_find_if(arr, arr + len, is_even);

}
```

invokes:

pred.operator()(*p)

# Function arguments

```
struct IsEven {
    bool operator() (int num) const { return num % 2 == 0; }
};

template <typename T, typename U>
T my_find_if(T start, T stop, U pred) {
    for (T p = start; p != stop; ++p) {
        if (pred(*p)) {
            return p;
        }
    }
    return stop;
}

int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    IsEven is_even;
    int* ptr = my_find_if(arr, arr + len, is_even);

}
```

*What is the benefit??*

# Function arguments

```
struct IsMultiple {
    int divisor;
    IsMultiple(int num) : divisor(num) {}
    bool operator() (int num) { return num % divisor == 0; }
};
```

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    IsMultiple mult_of_7(7);
    int* ptr = my_find_if(arr, arr + len, ___);

}
```

# Function arguments

*divisor member enables multiple functions with different divisors to be defined*

```
struct IsMultiple {
    int divisor;
    IsMultiple(int num) : divisor(num) {}
    bool operator() (int num) { return num % divisor == 0; }
};
```

*IsMultiple mult_of_3(3);*
*IsMultiple mult_of_8(8);*
*IsMultiple mult_of_5(5);*
*etc...*

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    IsMultiple mult_of_7(7);
    int* ptr = my_find_if(arr, arr + len, mult_of_7);

}
```

*finds first array element that is a multiple of 7*

# Function arguments

```
struct IsMultiple {
    int divisor;
    IsMultiple(int num) : divisor(num) {}
    bool operator() (int num) { return num % divisor == 0; }
};
```

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    IsMultiple mult_of_7(7);
    int* ptr = my_find_if(arr, arr + len, mult_of_7);

}
```

functor can be used without declaring struct instance

# Function arguments

```
struct IsMultiple {
    int divisor;
    IsMultiple(int num) : divisor(num) {}
    bool operator() (int num) { return num % divisor == 0; }
};
```

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    IsMultiple mult_of_7(7);
    int* ptr = my_find_if(arr, arr + len, ___);

}
```

functor can be used without declaring struct instance

# Function arguments

```
struct IsMultiple {
    int divisor;
    IsMultiple(int num) : divisor(num) {}
    bool operator() (int num) { return num % divisor == 0; }
};
```

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    IsMultiple mult_of_7(7);
    int* ptr = my_find_if(arr, arr + len, IsMultiple(17));

}
```

lambda expressions provide 3rd approach

# Function arguments

*parameter list*

```
[] (int num)  { return num % 2 == 0; }
```

*lambda-introducer*

*body of function*

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;

    int* ptr = my_find_if(arr, arr + len, [] (int num)  { return num % 2 == 0; });
}
```

*lambda expressions provide 3rd approach*

*finds first even integer*

# Function arguments

```
[] (int num)  { return num % 2 == 0; }
```

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;


    int* ptr = my_find_if(arr, arr + len, [] (int num) -> bool { return num % 2 == 0; });

}
```

finds first even integer

# Function arguments

```
[] (int num) -> bool { return num % 2 == 0; }
```

*trailing return type*

```
int main() {
    int arr[] = {90, 30, 23, 68, 4};
    const int len = 5;


    int* ptr = my_find_if(arr, arr + len, [] (int num) -> bool { return num % 2 == 0; });

}
```

*finds first even integer*

# Function arguments

```
int main() {
    [] { cout << "lambda\n"; }();
}
```

*lambda not required to return a value*

```
lambda
```

# Function arguments

```
int main() {

    auto func = [] { cout << "lambda\n"; };
    func();
}
```

*lambda can be used to create functor*

# In-class problem

# A templated Vector class

```
class Vector {
public:
    explicit Vector(size_t size, 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;
    size_t the_capacity;
};
```

# What changes need to be made to this class so that the `Vector` class can support storage of any single type when instantiated?

```
class Vector {
public:
    explicit Vector(size_t size, 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;
    size_t the_capacity;
};
```

# A templated Vector class

```
---
class Vector {
public:
    explicit Vector(size_t size, 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;
    size_t the_capacity;
};
```

# A templated Vector class

```
_1_
class Vector {
public:
    explicit Vector(size_t size, 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;
    size_t the_capacity;
};
```

# What declaration replaces blank #1 so that the `Vector` class can support storage of a generic type `T`?

```
_1_
class Vector {
public:
    explicit Vector(size_t size, 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;
    size_t the_capacity;
};
```

# A templated Vector class

```
template <typename T>
class Vector {
public:
    explicit Vector(size_t size, 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;
    size_t the_capacity;
};
```

# A templated Vector class

```
template <typename T>
class Vector {
public:
    explicit Vector(size_t size, ___ value = 0) {
        the_size = size;
        the_capacity = size;
        data = new ___[size];
        for (size_t i = 0; i < the_size; ++i) {
            data[i] = value;
        }
    }
 ...

private:
    ___* data;
    size_t the_size;
    size_t the_capacity;
};
```

# A templated Vector class

```
template <typename T>
class Vector {
public:
    explicit Vector(size_t size, _2_ value = 0) {
        the_size = size;
        the_capacity = size;
        data = new _2_[size];
        for (size_t i = 0; i < the_size; ++i) {
            data[i] = value;
        }
    }
 ...

private:
    _2_* data;
    size_t the_size;
    size_t the_capacity;
};
```

# Which name replaces blank #2 to support a templated version of the `Vector` class?

```
template <typename T>
class Vector {
public:
    explicit Vector(size_t size, _2_ value = 0) {
        the_size = size;
        the_capacity = size;
        data = new _2_[size];
        for (size_t i = 0; i < the_size; ++i) {
            data[i] = value;
        }
    }
 ...

private:
    _2_* data;
    size_t the_size;
    size_t the_capacity;
};
```

# A templated Vector class

```
template <typename T>
class Vector {
public:
    explicit Vector(size_t size, T value = 0) {
        the_size = size;
        the_capacity = size;
        data = new T[size];
        for (size_t i = 0; i < the_size; ++i) {
            data[i] = value;
        }
    }
 ...

private:
    T* data;
    size_t the_size;
    size_t the_capacity;
};
```

*Full code for templated class available on Brightspace*

# Solving problems with recursion

# A recursive function

```
void recursive_function() {
    recursive_function();
}
```

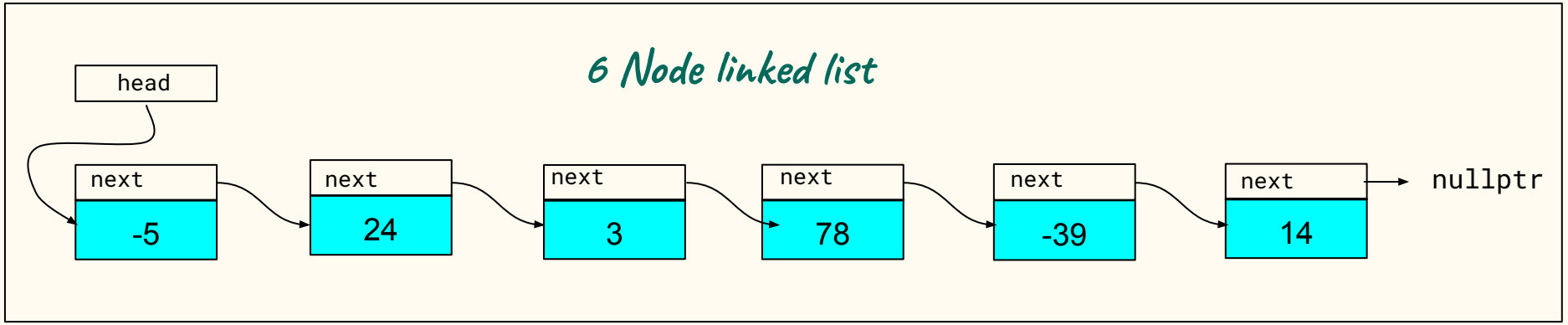
```
int main() {
   recursive_function();   // infinite loop
}
```

# Solving recursive problems

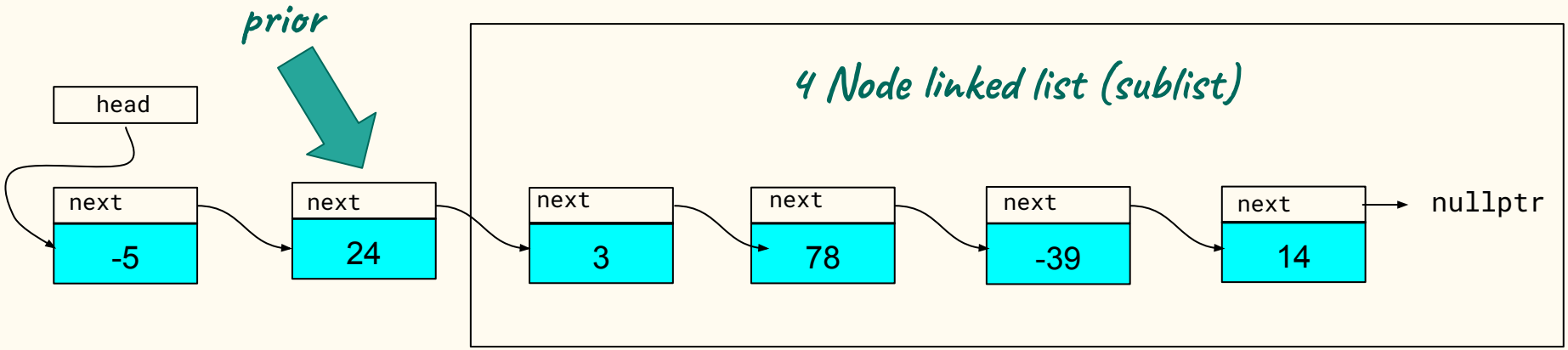
1) define version of problem with direct solution — *base case*
2) identify problem structure
   a) break problem into smaller subproblems of same form
   b) call recursive function on subproblem — *recursive case*
   c) assemble solutions to subproblems to solve full problem

# Linked lists and recursion

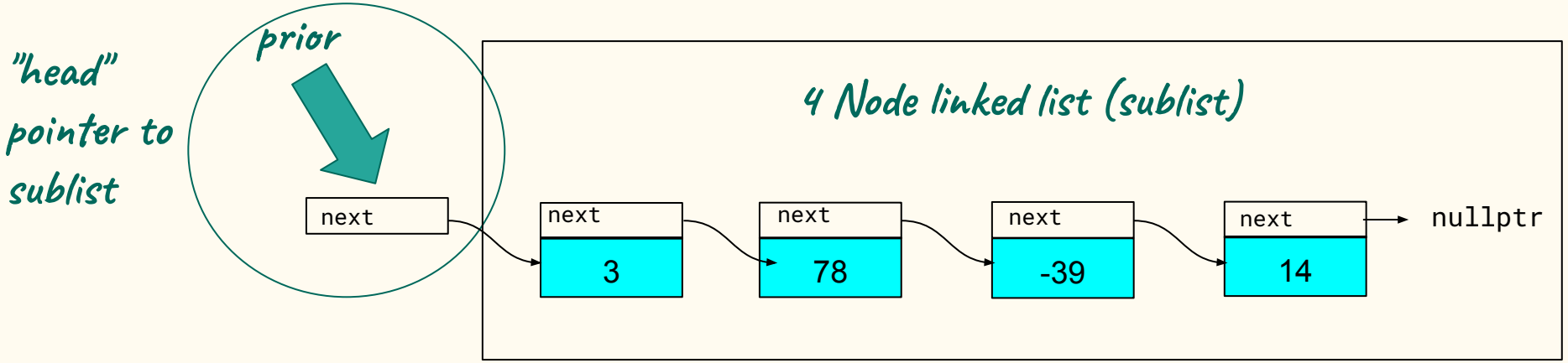
# Linked lists as recursive data structures



# Linked lists as recursive data structures



# Linked lists as recursive data structures




functions can be applied to sublists until no sublists remain

# Linked lists as recursive data structures

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

Node* build_list(const vector<int>& vals) {
    Node* head = nullptr;
    for (size_t i = vals.size(); i > 0; --i) {
        head = new Node(vals[i-1], head);
    }
    return head;
}

int main() {
    Node* my_list = build_list({1, 1, 2, 3, 5, 8, 13, 21, 34, 55});
}
```

# Linked lists as recursive data structures

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

Node* build_list(const vector<int>& vals);
```

```
int main() {
    Node* my_list = build_list({1, 1, 2, 3, 5, 8, 13, 21, 34, 55});
}
```

# Linked lists as recursive data structures

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

Node* build_list(const vector<int>& vals);

int main() {
    Node* my_list = build_list({1, 1, 2, 3, 5, 8, 13, 21, 34, 55});
    print_list(my_list);



}
```

```
1 1 2 3 5 8 13 21 34 55
```

# Linked lists as recursive data structures

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

void print_list() {}
```

# Linked lists as recursive data structures

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


void print_list(_1_ ptr) {}
```

# What type replaces blank #1 when declaring the `ptr` parameter?

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

void print_list(_1_ ptr) {}
```

# Linked lists as recursive data structures

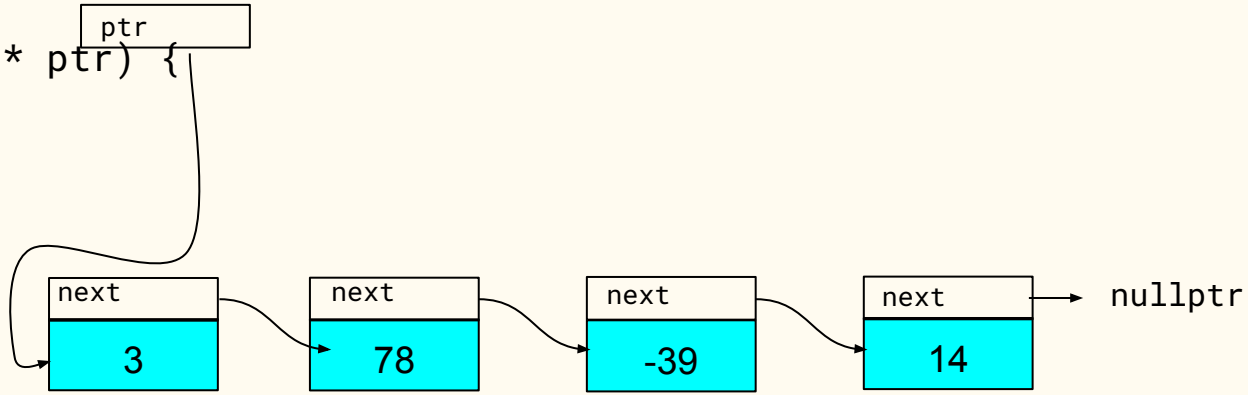
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


void print_list(const Node* ptr)
{}
```

# Linked lists as recursive data structures

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

void print_list(const Node* ptr) {}
```

# Linked lists as recursive data structures

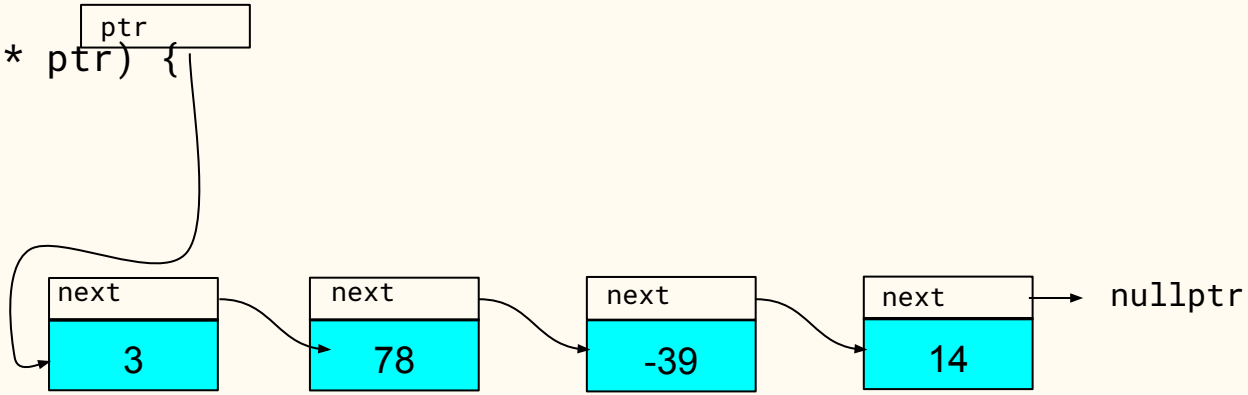
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};

void print_list(const Node* ptr) {
    // base case

    // recursive case

}
```




```
3 78 -39 14
```

# What operation needs to happen at each `Node` in the list for the `print_list()` function definition?

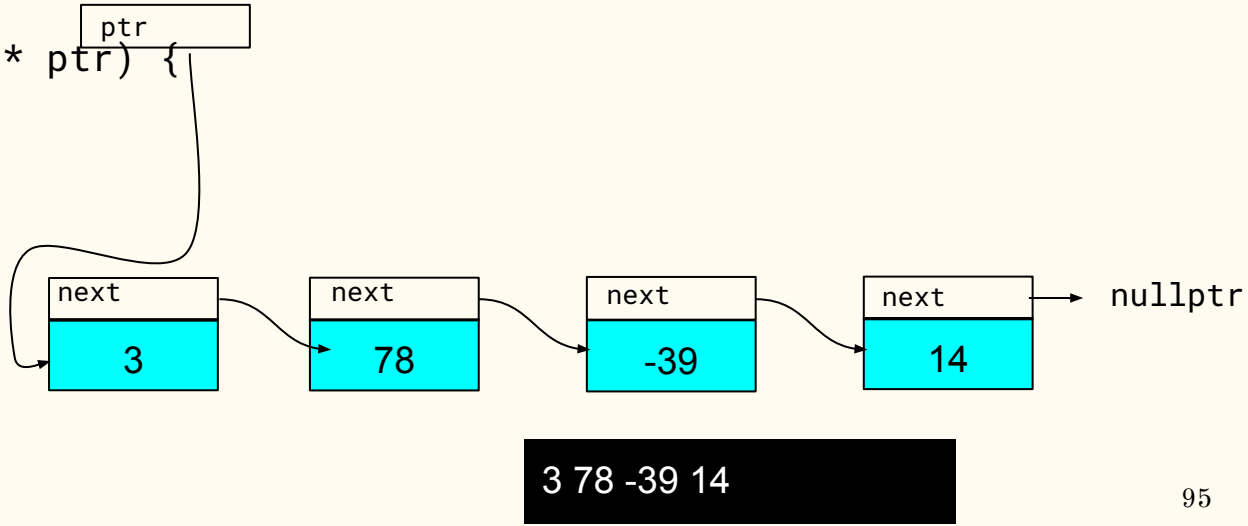
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};

void print_list(const Node* ptr) {
    // base case

    // recursive case

}
```




```
3 78 -39 14
```

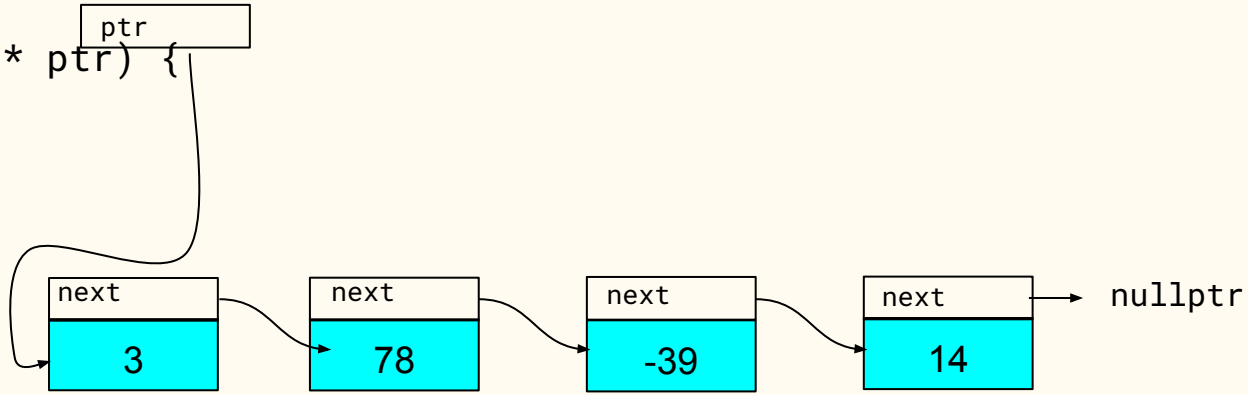
# Linked lists as recursive data structures

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

void print_list(const Node* ptr) {
    // base case

    // recursive case
    // output data
}
```



# When are we done? What will be true of the state of the problem after all `Node` data has been printed?

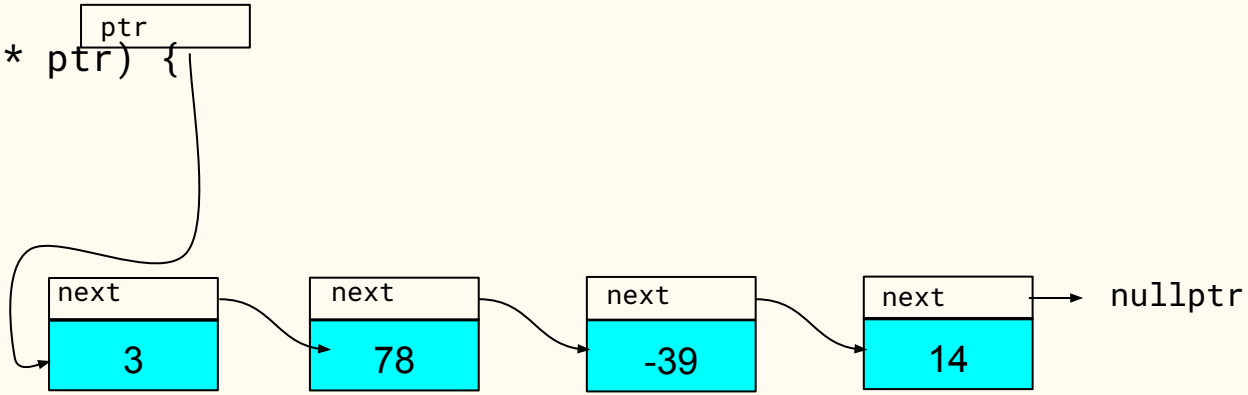
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};

void print_list(const Node* ptr) {
    // base case

    // recursive case
    // output data
}
```




```
3 78 -39 14
```

# Linked lists as recursive data structures

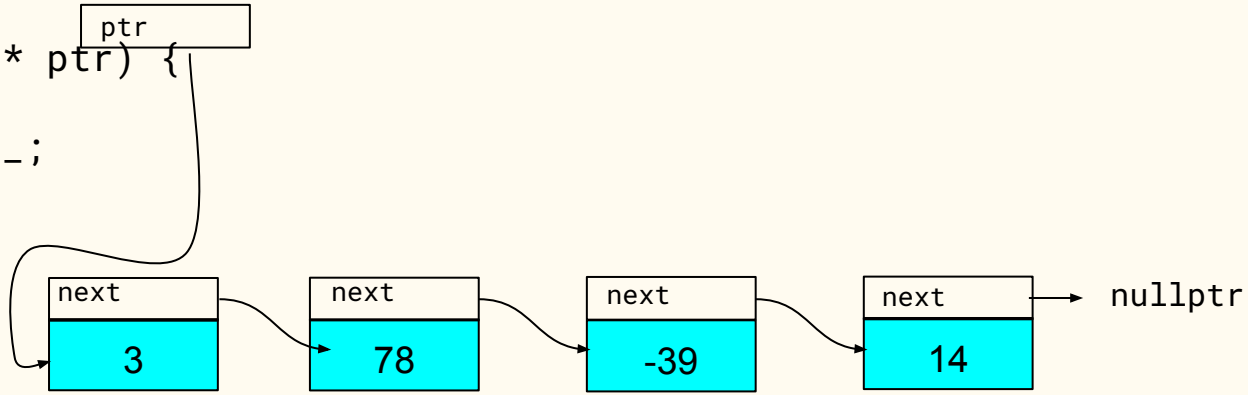
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};

void print_list(const Node* ptr) {
    // base case

    // recursive case
    // output data
}
```




```
3 78 -39 14
```

# Linked lists as recursive data structures

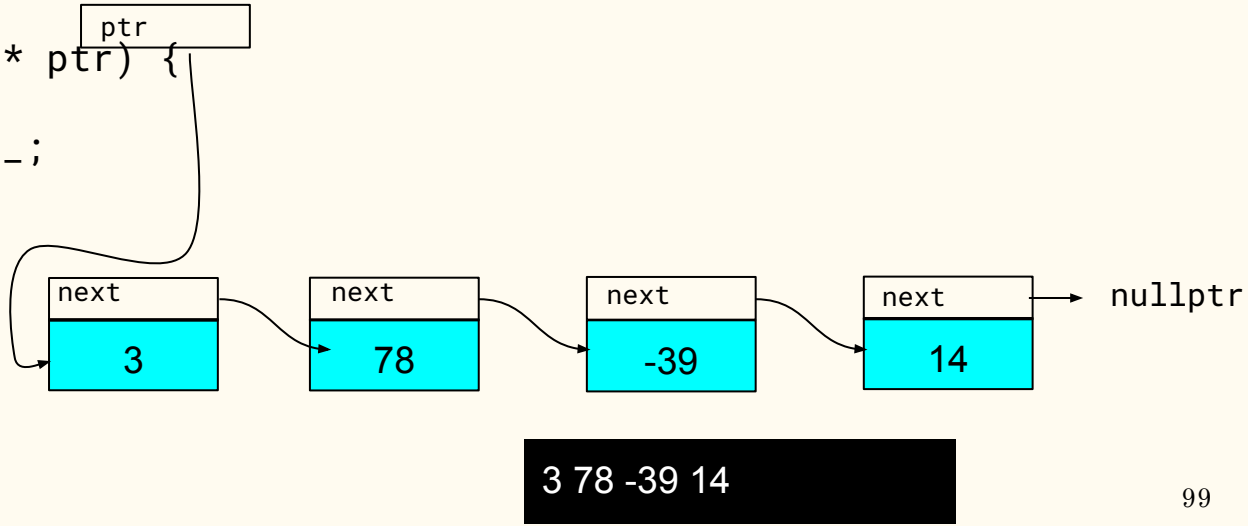
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};

void print_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) ___;

    // recursive case
    // output data
}
```




```
3 78 -39 14
```

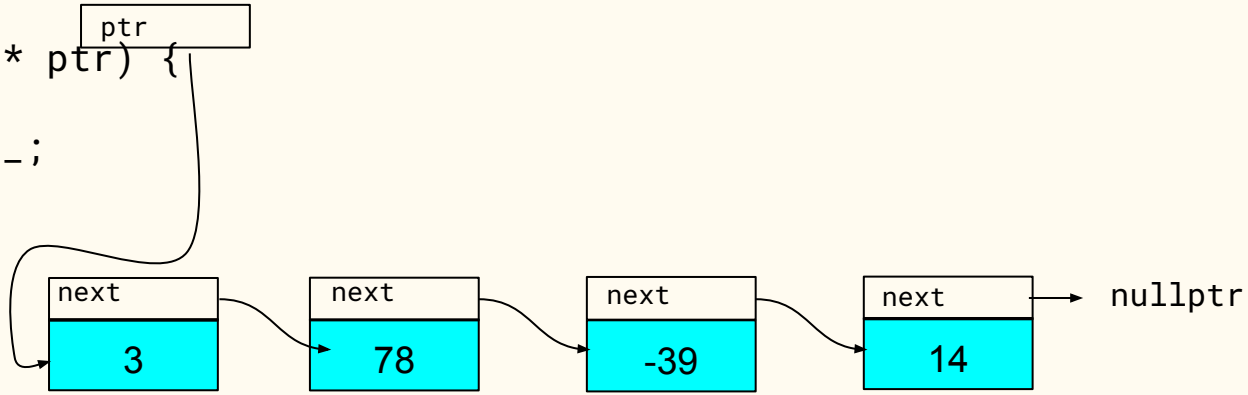
# Linked lists as recursive data structures

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

void print_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) _1_;

    // recursive case
    // output data
}
```



# Which statement replaces blank #1 so that the recursion ends when `ptr` is equal to the `nullptr`?

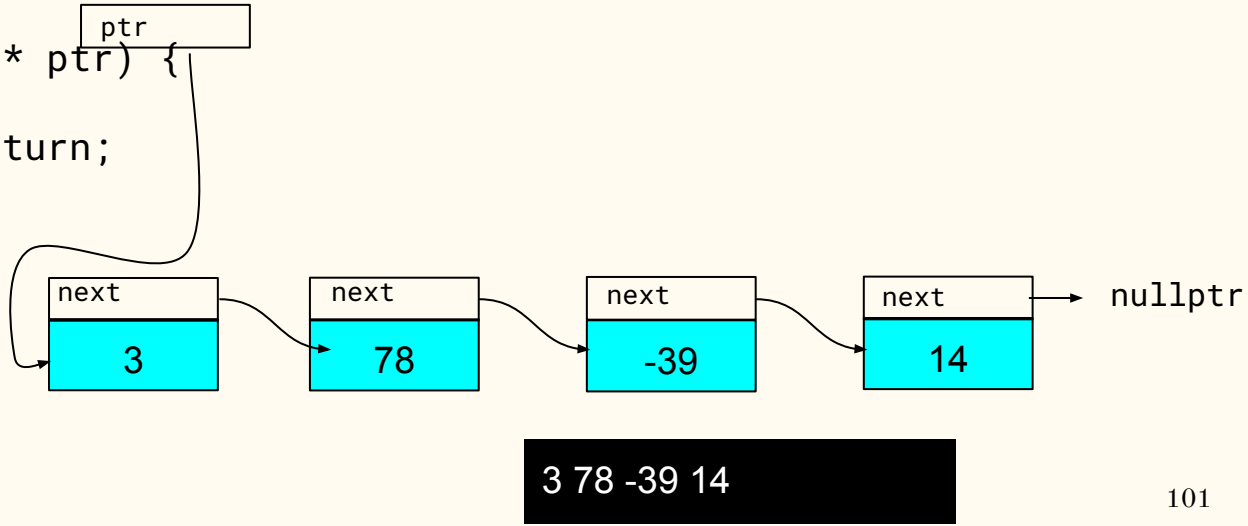
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};

void print_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) _1_;

    // recursive case
    // output data
}
```




3 78 -39 14

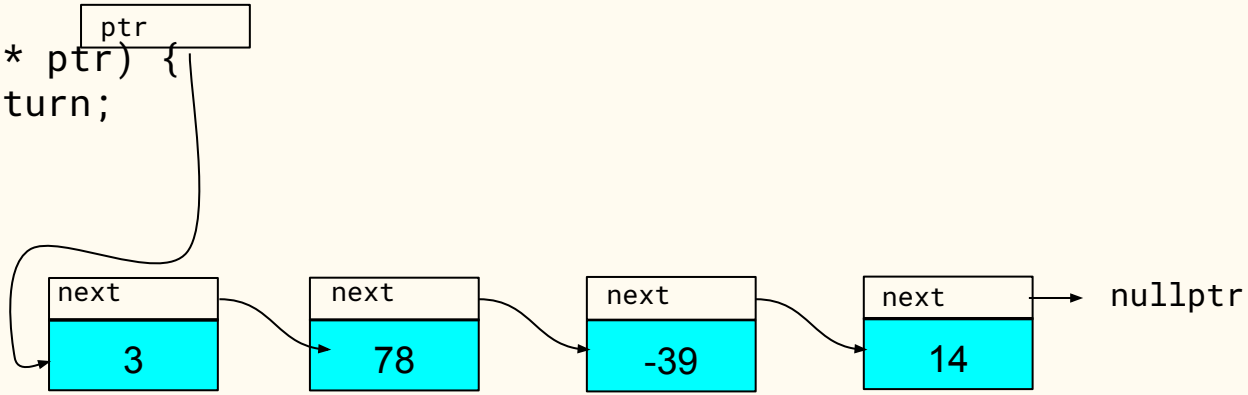
# Linked lists as recursive data structures

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

void print_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) return;

    // recursive case
    // output data
}
```



# Linked lists as recursive data structures

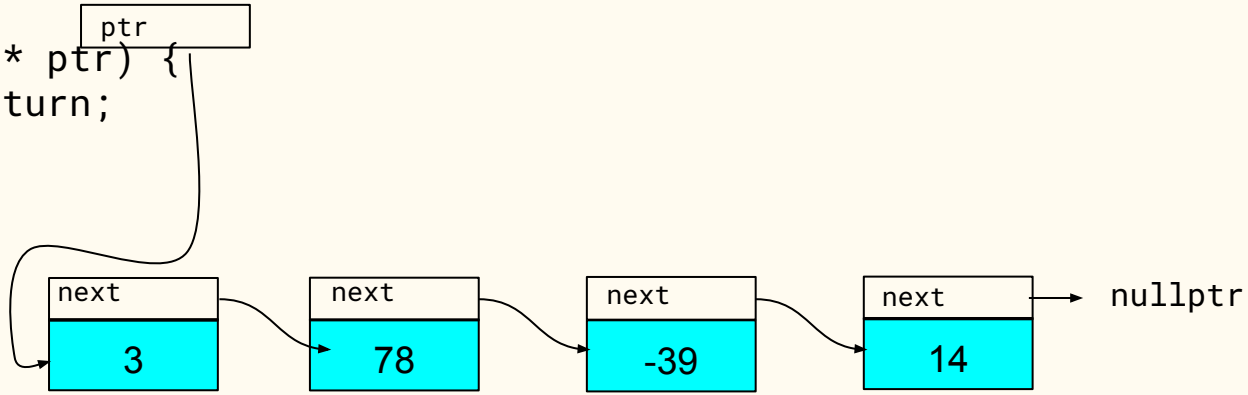
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};

void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    // output data
    cout << ___ << ' ';
}
```




```
3 78 -39 14
```

# Linked lists as recursive data structures

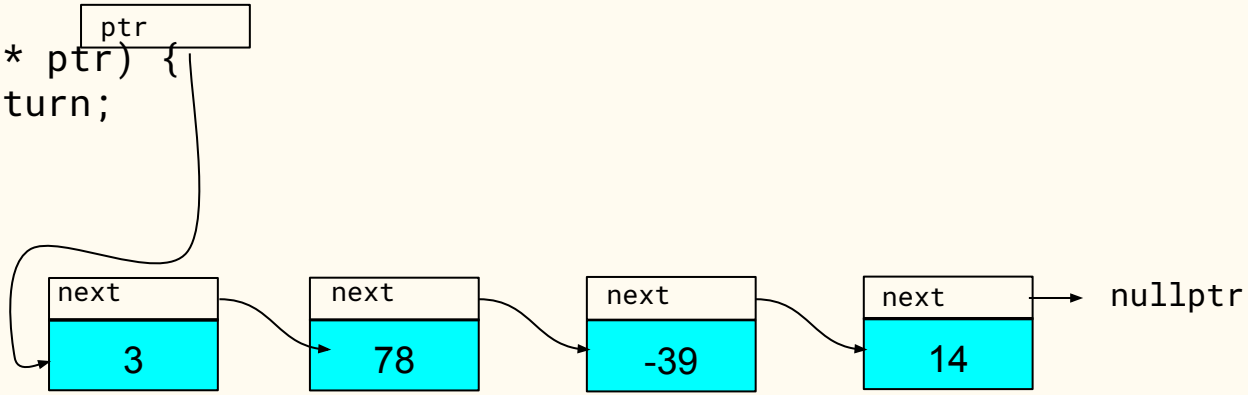
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};

void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    // output data
    cout << _2_ << ' ';
}
```




```
3 78 -39 14
```

# Which expression replaces blank #2 to output the `data` at the current `Node`?

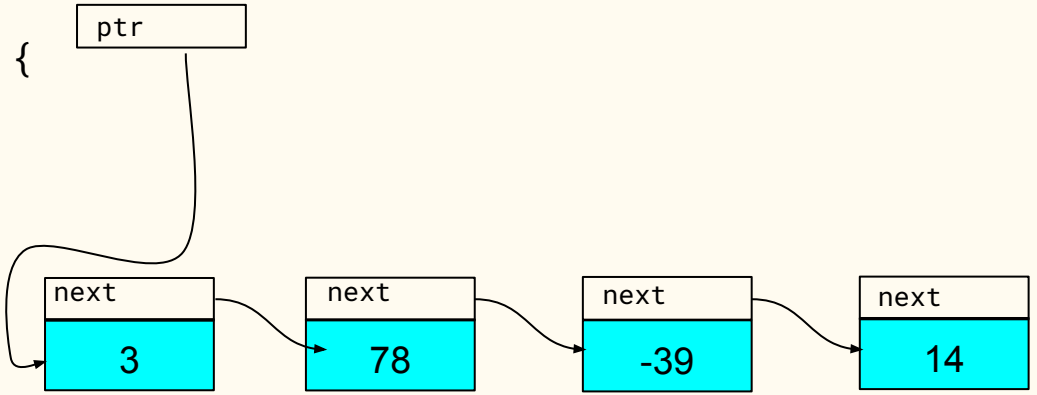
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};

void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    // output data
    cout << _2_ << ' ';
}
```




3 78 -39 14

# Linked lists as recursive data structures

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

void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    // output data
    cout << ptr->data << ' ';
}
```

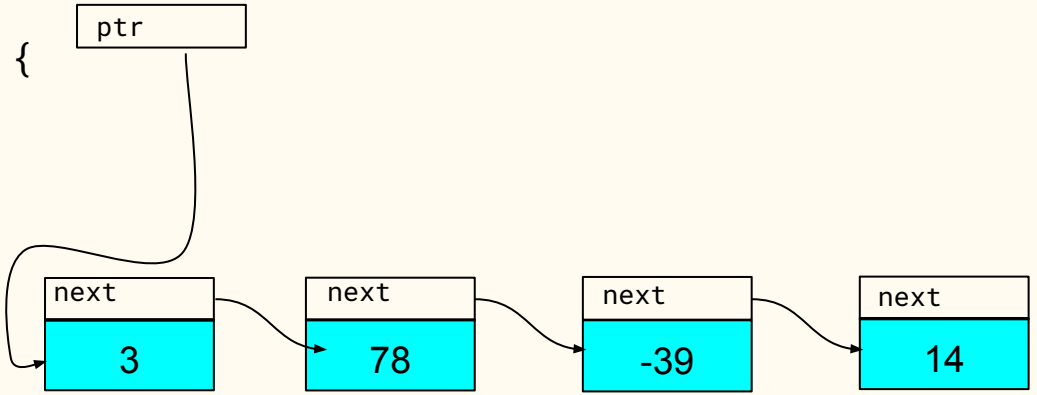



```
3 78 -39 14
```

# Linked lists as recursive data structures

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

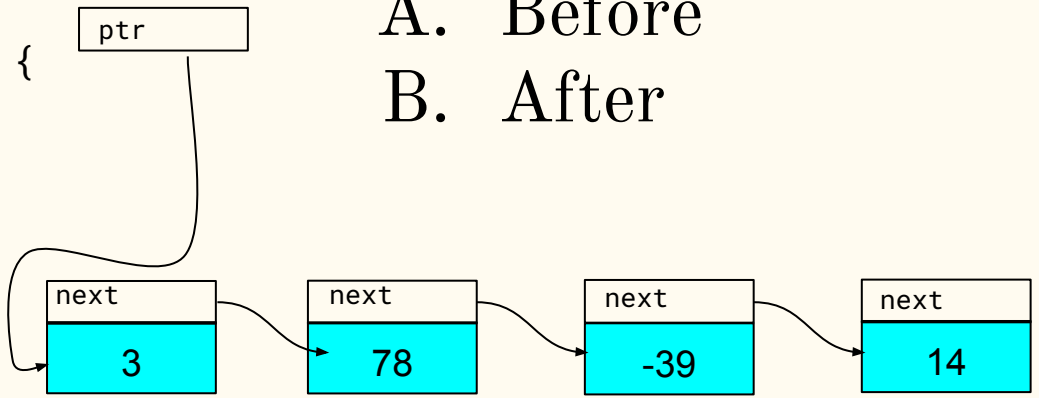
```
void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    cout << ptr->data << ' ';
}
```




```
3 78 -39 14
```

In order to print the remainder of the list, does the function call need to occur before or after outputting the `data` of the current `Node`?

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

void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    cout << ptr->data << ' ';
}
```



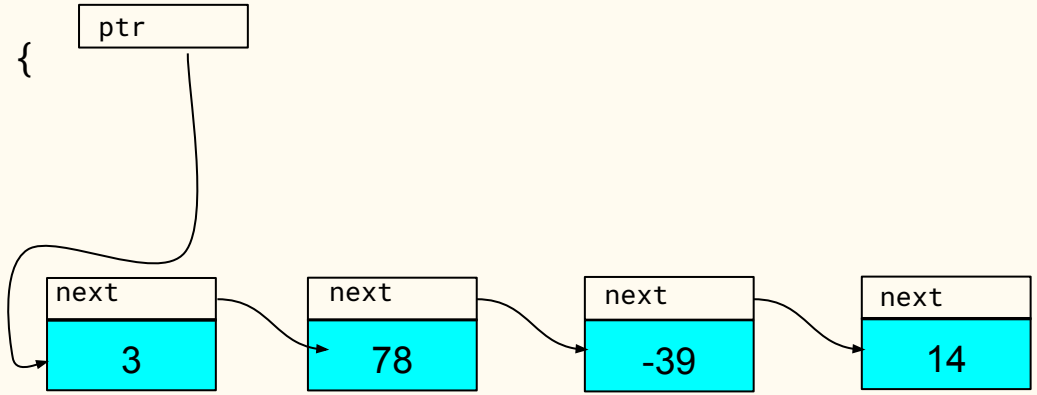

A. Before
B. After

3 78 -39 14

# Linked lists as recursive data structures

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

```
void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    cout << ptr->data << ' ';
    // print rest of list
}
```

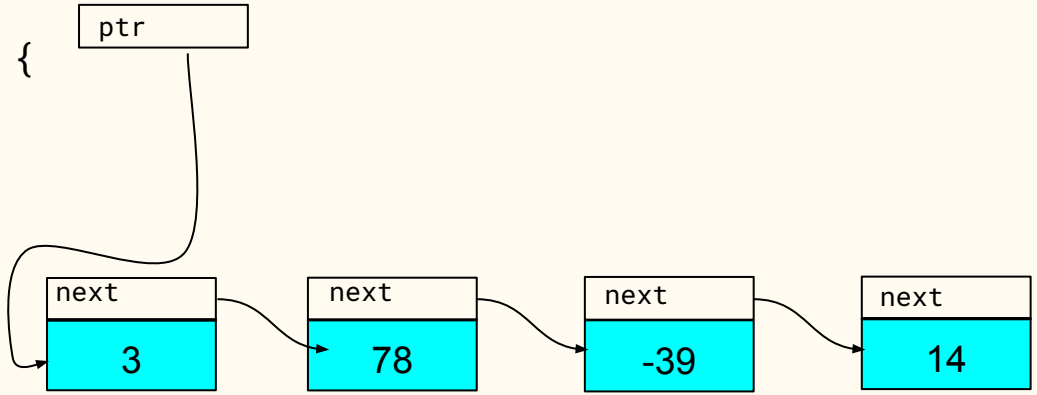



```
3 78 -39 14
```

# Linked lists as recursive data structures

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

```
void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    cout << ptr->data << ' ';
    // print rest of list
    ---
}
```

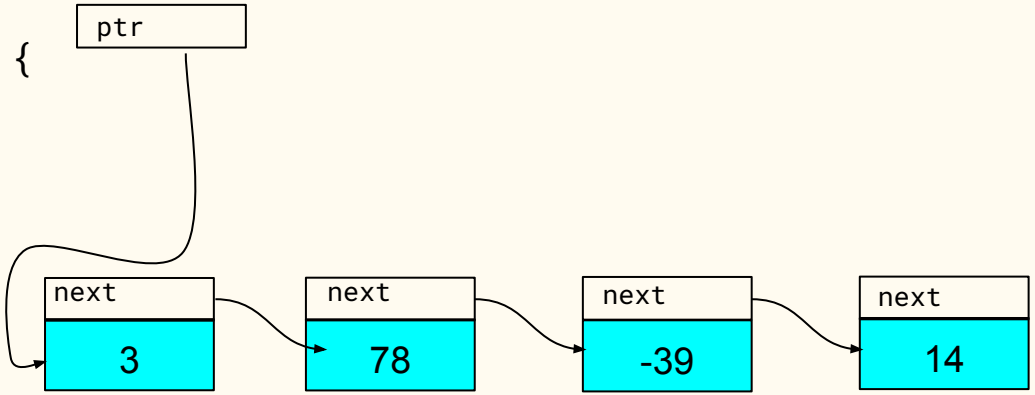



```
3 78 -39 14
```

# Linked lists as recursive data structures

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

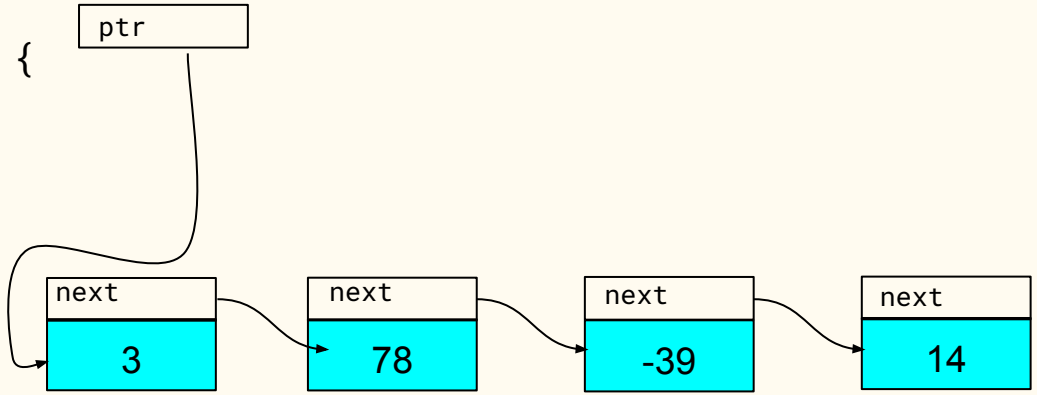
```
void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    cout << ptr->data << ' ';
    // print rest of list
    _3_
}
```




```
3 78 -39 14
```

# Which expression replaces blank #3 to output the remainder of the list?

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

void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    cout << ptr->data << ' ';
    // print rest of list
    _3_
}
```

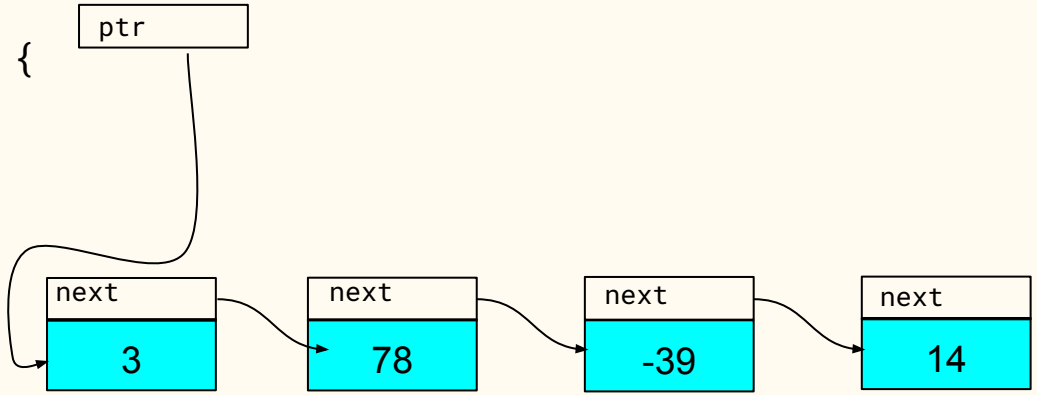



3 78 -39 14

# Linked lists as recursive data structures

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

```
void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    cout << ptr->data << ' ';
    // print rest of list
    print_list(ptr->next);
}
```

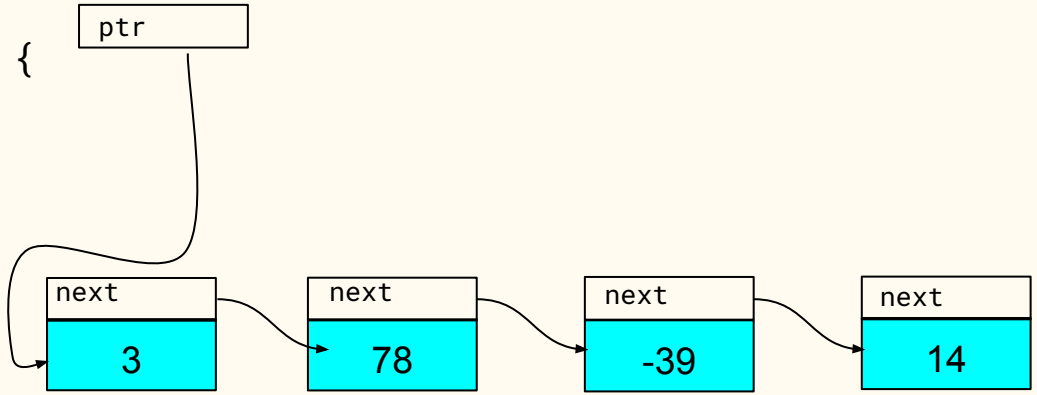



```
3 78 -39 14
```

# Linked lists as recursive data structures

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

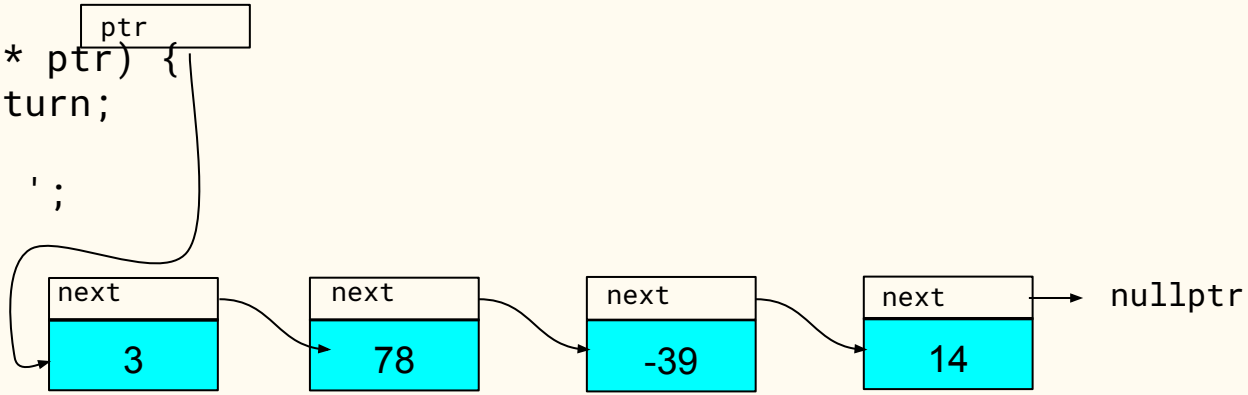
```
void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    // recursive case
    cout << ptr->data << ' ';
    print_list(ptr->next);
}
```




```
3 78 -39 14
```

# Linked lists as recursive data structures

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

void print_list(const Node* ptr) {
    if (ptr == nullptr) return;

    cout << ptr->data << ' ';
    print_list(ptr->next);
}
```




```
3 78 -39 14
```

# Linked lists as recursive data structures

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

void print_list(const Node* ptr) {
    if (ptr == nullptr) return;   base case

    cout << ptr->data << ' ';
    print_list(ptr->next);        recursive case
}
```

# Linked lists as recursive data structures

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

Node* build_list(const vector<int>& vals);
void print_list(const Node* ptr);

int main() {
    Node* my_list = build_list({1, 1, 2, 3, 5, 8, 13, 21, 34, 55});
    print_list(my_list);

    cout << endl;
    Node* other_list = dup_list(my_list);



}
```

# Linked lists as recursive data structures

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


___ dup_list() {}
```

# Linked lists as recursive data structures

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


_4_ dup_list() {}
```

# Which return type replaces blank #4 when duplicating a list?

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

_4_ dup_list() {}
```

# Linked lists as recursive data structures

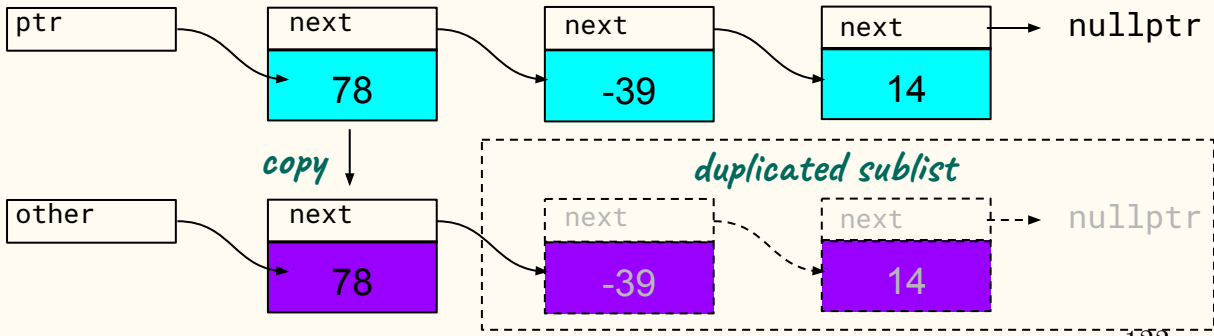
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


Node* dup_list() {}
```

# Linked lists as recursive data structures

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


Node* dup_list(const Node* ptr) {}
```

# Linked lists as recursive data structures

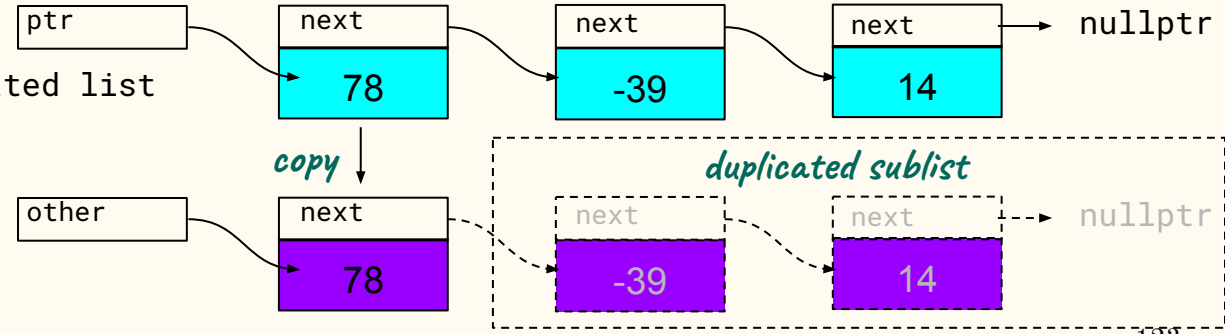
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


Node* dup_list(const Node* ptr) {
    // base case

    // recursive case
}
```



# Linked lists as recursive data structures

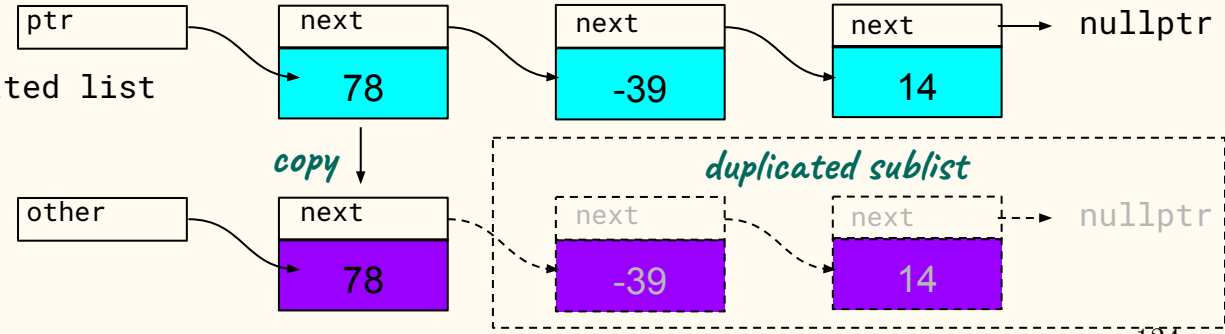
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {
    // base case

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



# When are we done? What will be true of the state of the problem after all `Node`s have been duplicated?

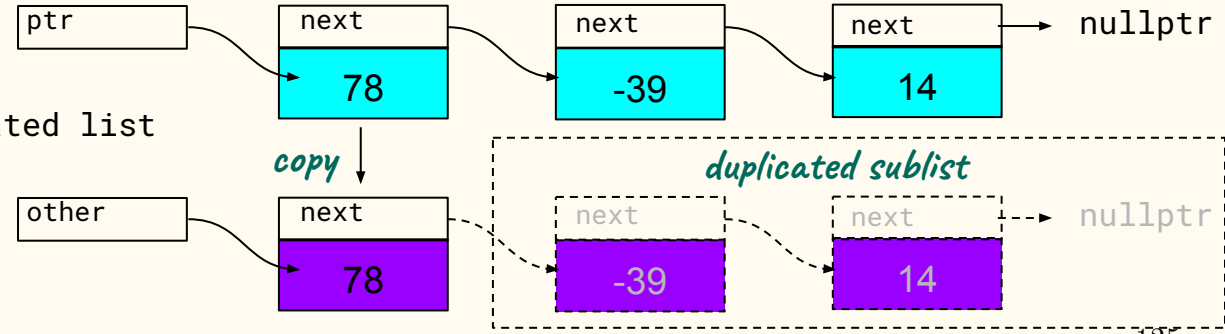
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


Node* dup_list(const Node* ptr) {
    // base case

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



# Linked lists as recursive data structures

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

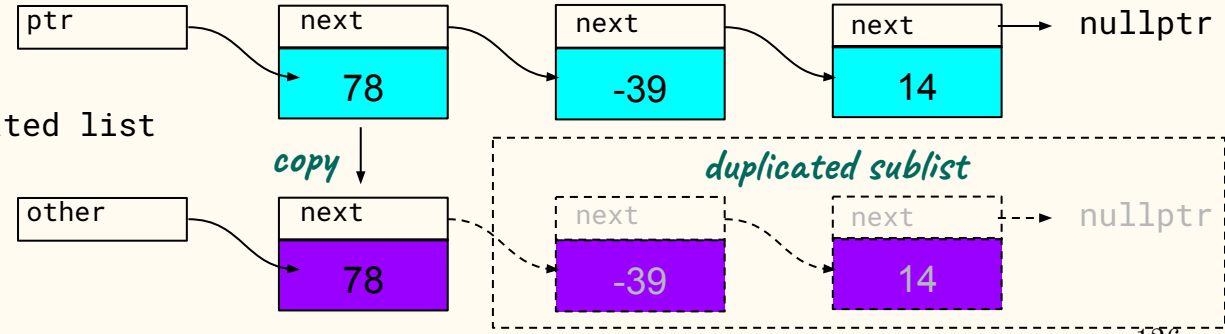
```
Node* dup_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) return ___;

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



# Linked lists as recursive data structures

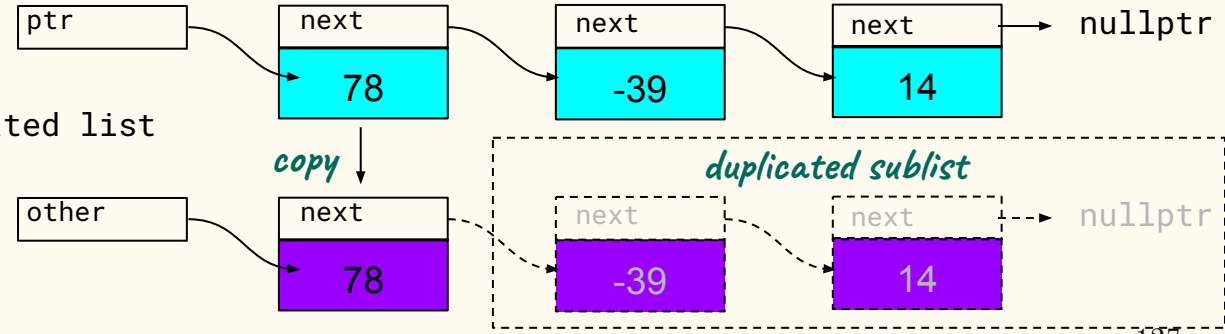
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) return _5_;

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



# Which value do we return (replacing blank #5) when we want to duplicate an *empty* list?

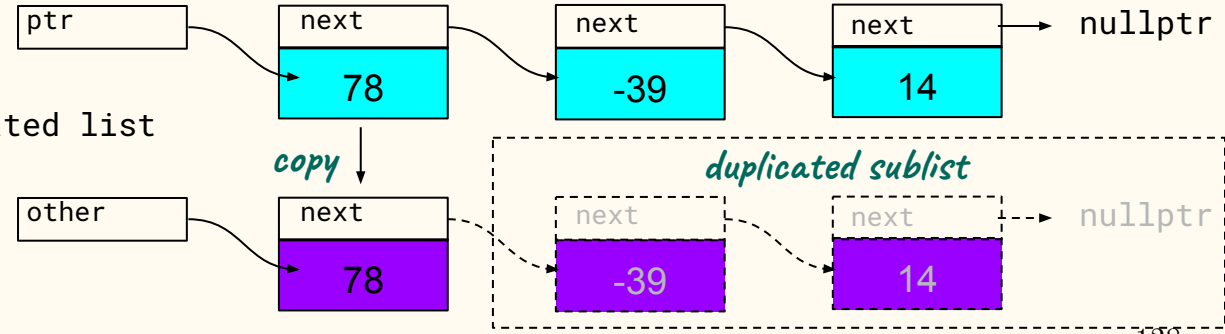
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


Node* dup_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) return _5_;

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



# Linked lists as recursive data structures

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

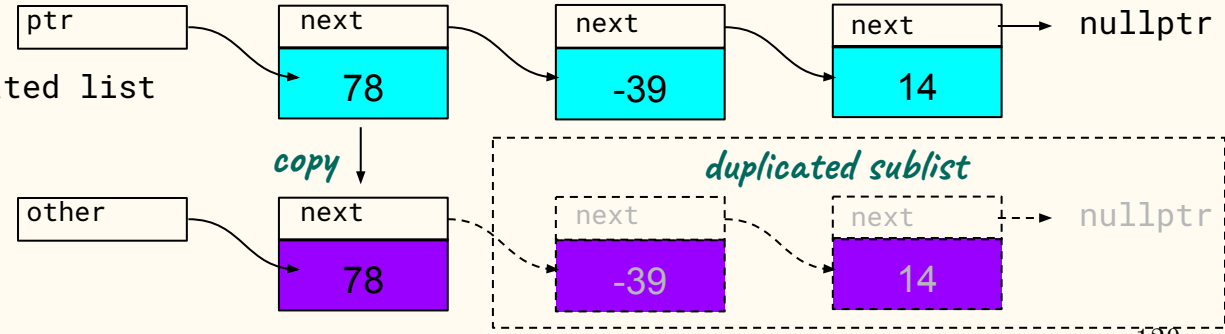
```
Node* dup_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



# Linked lists as recursive data structures

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

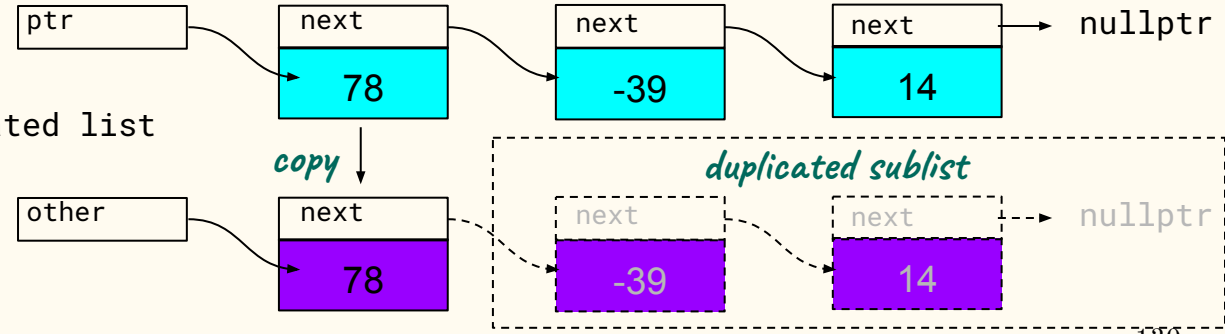
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



# Linked lists as recursive data structures

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

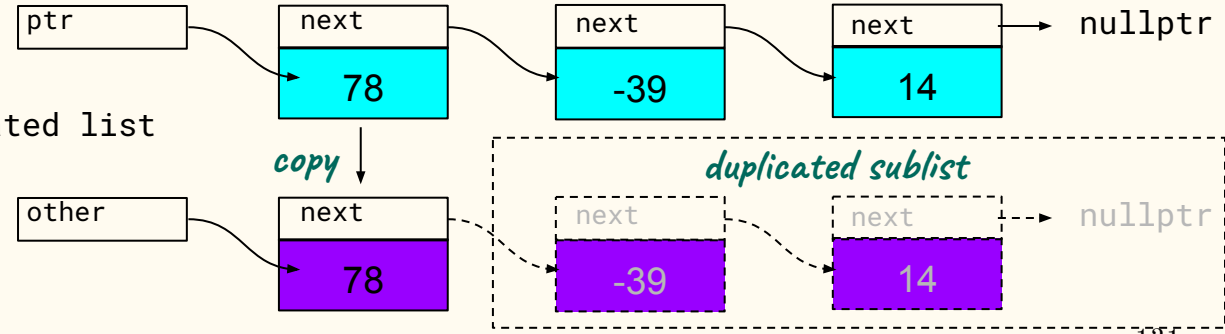
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    ___
    // make Node head of duplicated list
}
```



# Linked lists as recursive data structures

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

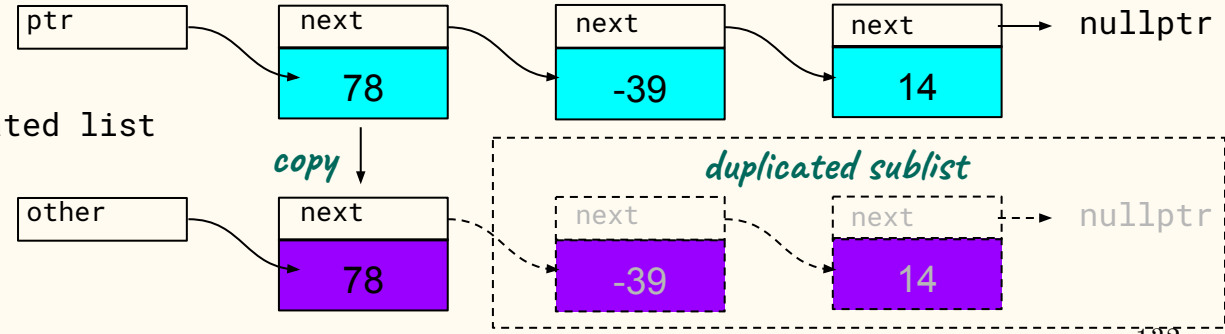
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = ___;
    // make Node head of duplicated list
}
```



# Linked lists as recursive data structures

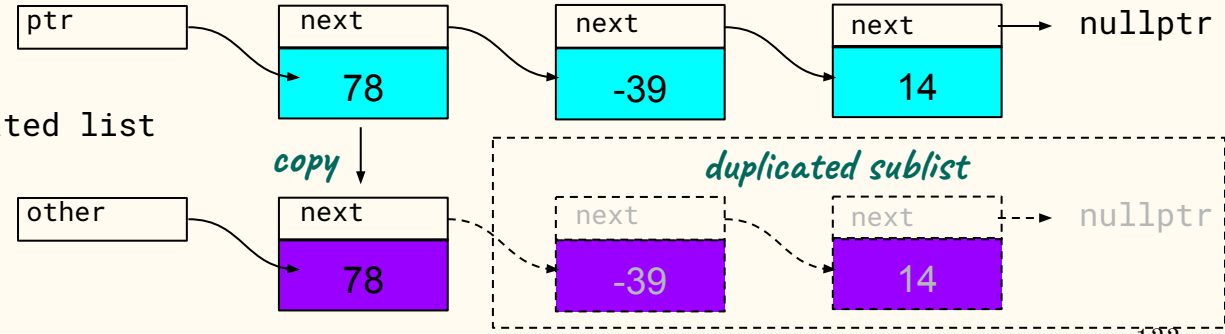
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = _6_;
    // make Node head of duplicated list
}
```



# Which expression replaces blank #6 to instantiate a Node on the heap with the same `data` value as the `Node` pointed to by `ptr`?

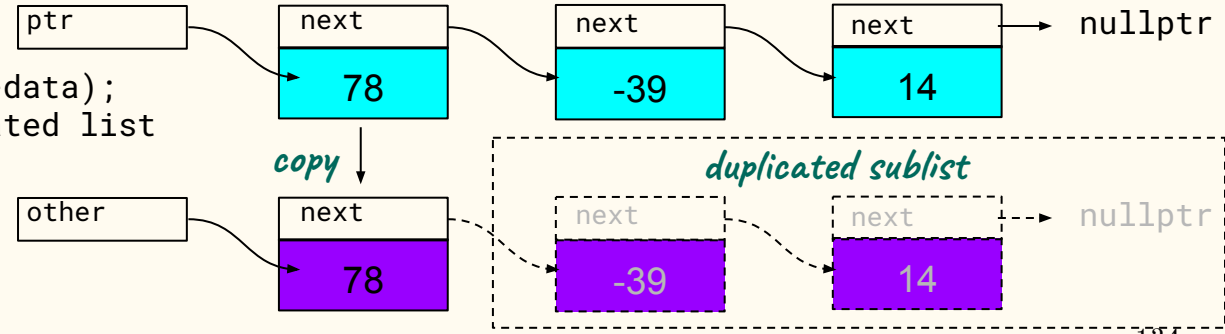
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = _6_;
    // make Node head of duplicated list
}
```



# Linked lists as recursive data structures

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

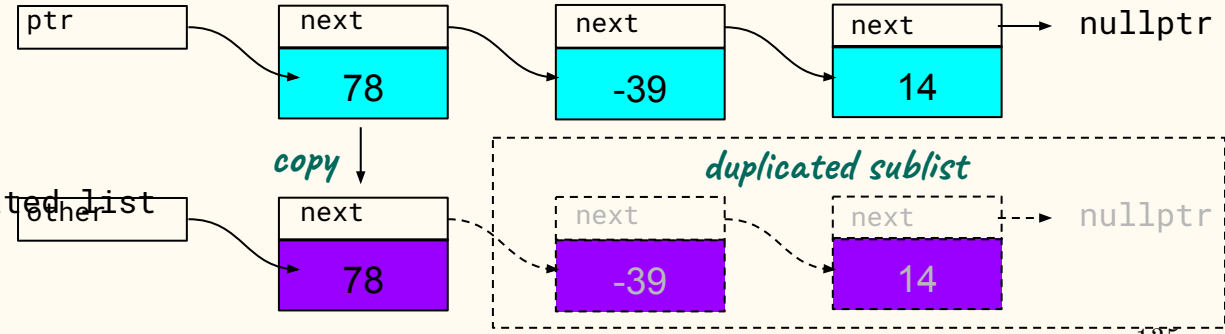
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(ptr->data);
    // make Node head of duplicated list
}
```



# Linked lists as recursive data structures

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

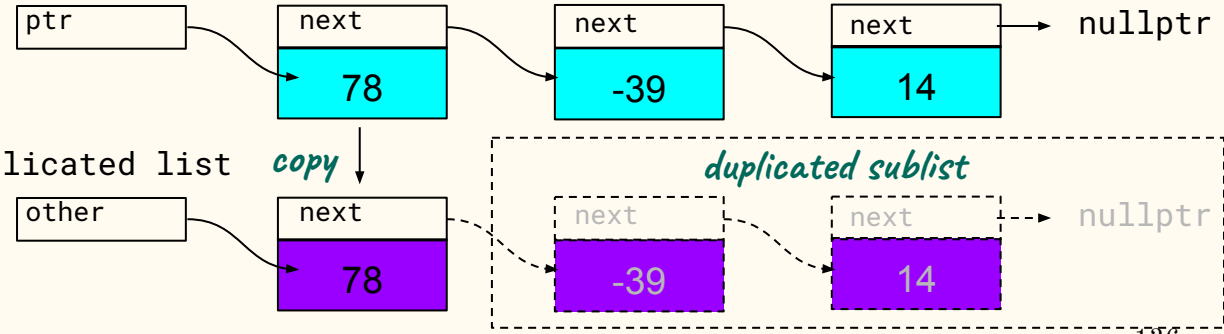
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data
    );
    // make Node head of duplicated list
}
```



# Linked lists as recursive data structures

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

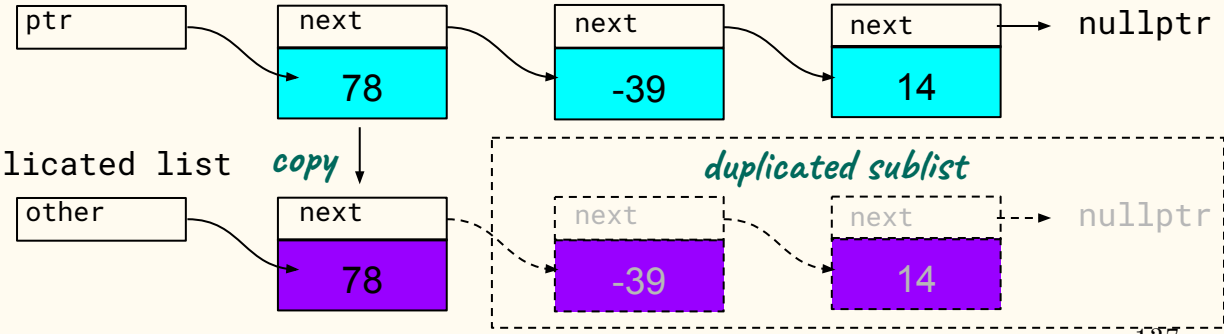
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        // make Node head of duplicated list
    );
}
```



# Linked lists as recursive data structures

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

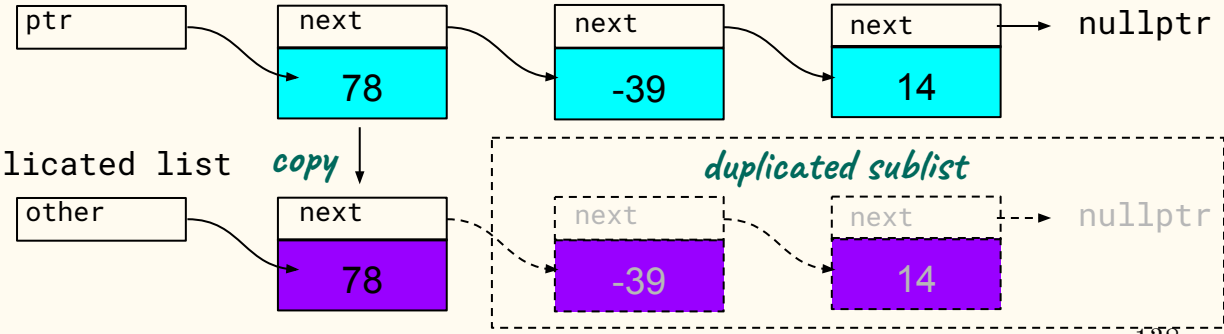
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        // make Node head of duplicated list
        ---
    );
}
```



# Linked lists as recursive data structures

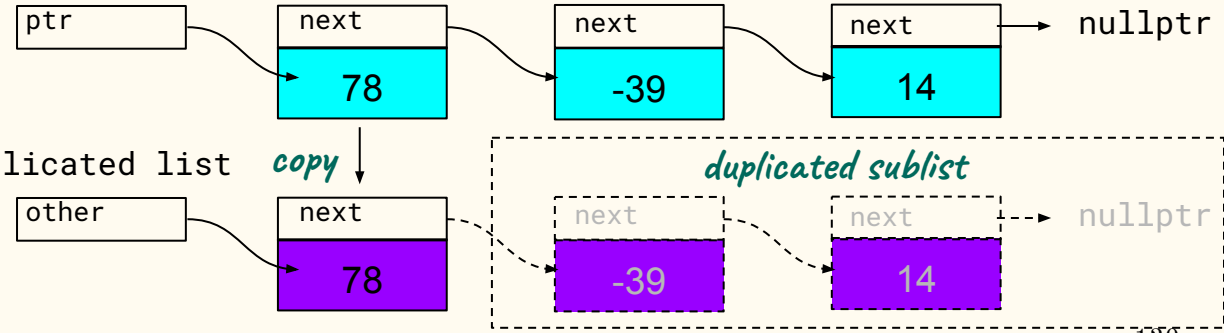
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        // make Node head of duplicated list
        _6_
    );
}
```



# Which expression replaces blank #6 to create a duplicate list from the sublist following the current `Node` "pointed to" by `ptr`?

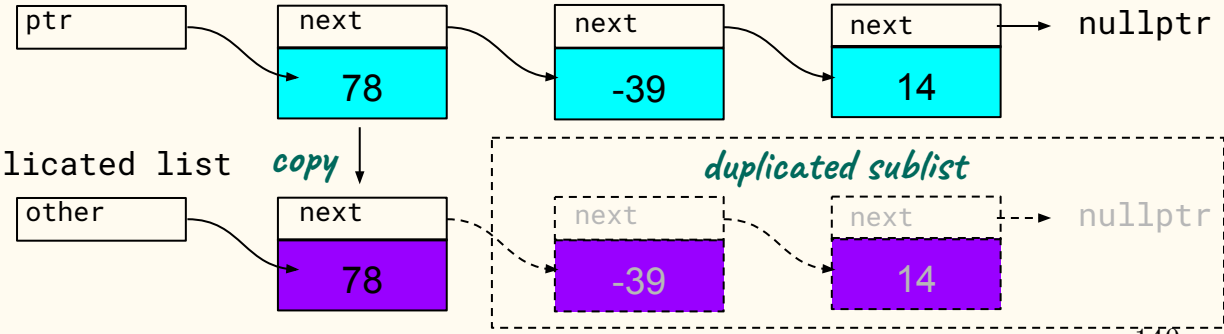
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        // make Node head of duplicated list
        _6_
    );
}
```



# Linked lists as recursive data structures

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

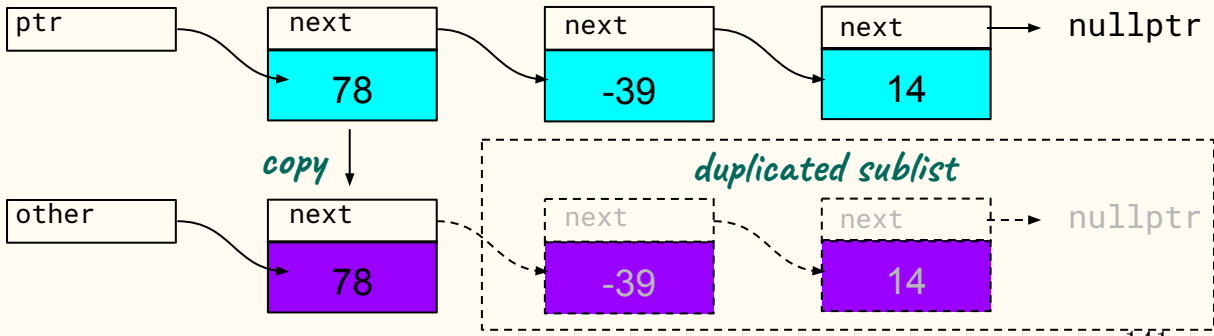
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        // make Node head of duplicated list
        dup_list(ptr->next)
    );
}
```



# Linked lists as recursive data structures

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

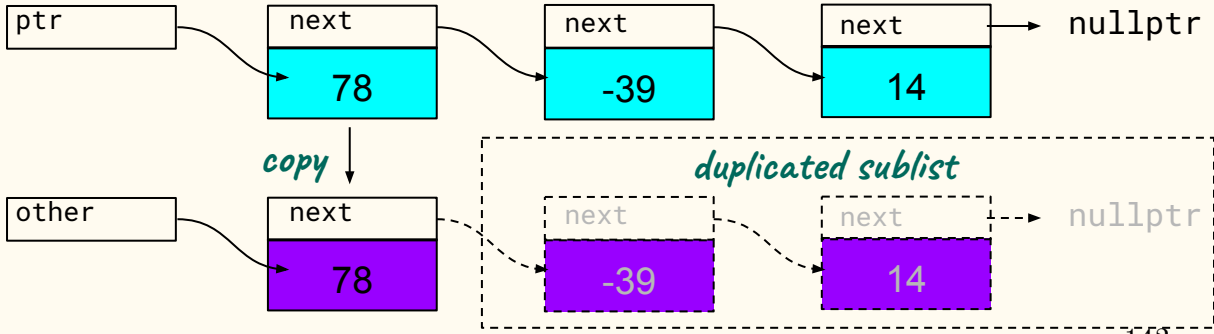
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
}
```



# Linked lists as recursive data structures

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

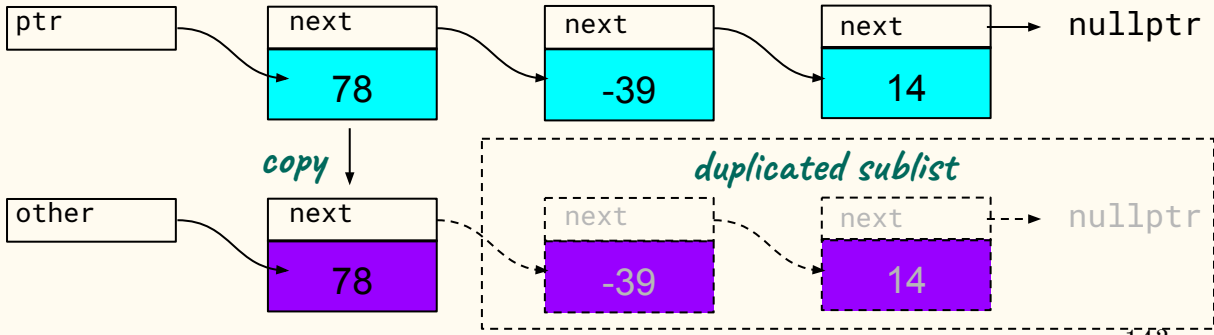
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
}
```



# Linked lists as recursive data structures

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

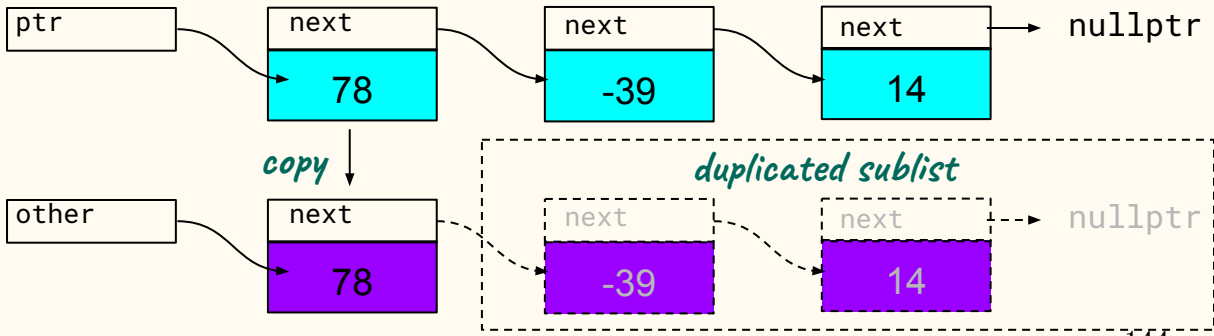
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    ---
}
```



# Linked lists as recursive data structures

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

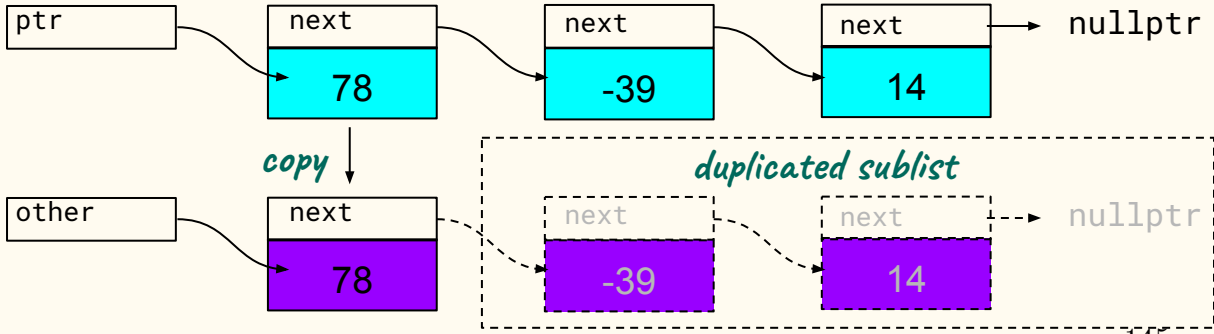
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return ___;
}
```



# Linked lists as recursive data structures

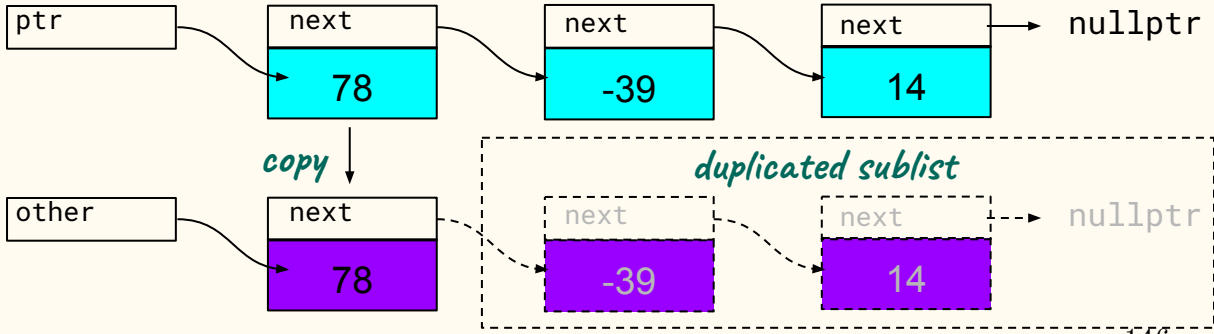
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return _7_;
}
```



What replaces blank #7 so that a `dup_list()` function call returns a pointer to the duplicated list?

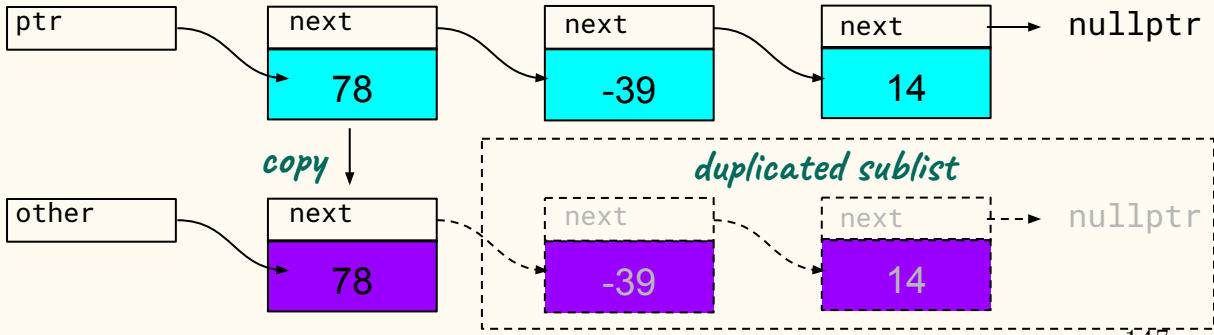
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return _7_;
}
```



# Linked lists as recursive data structures

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

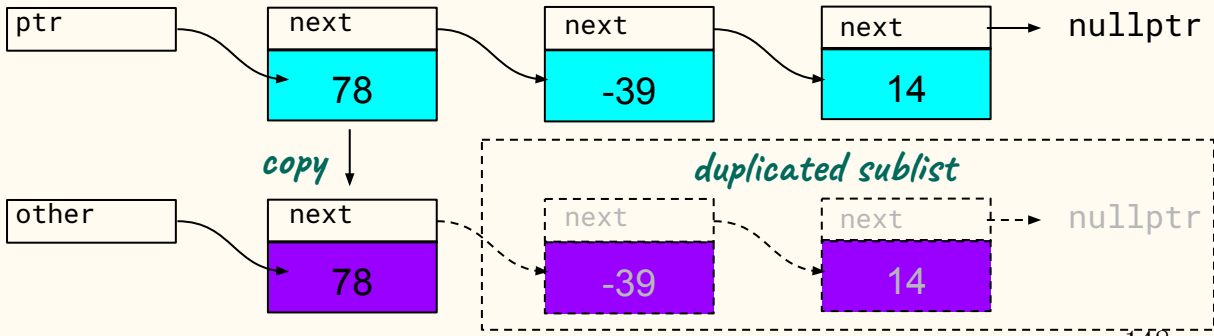
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return other;
}
```



# Linked lists as recursive data structures

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

```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return other;
}
```



# Linked lists as recursive data structures

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

```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return other;
}
```

*combine into single statement*

# Linked lists as recursive data structures

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

```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    return new Node(
         ptr->data,
         dup_list(ptr->next)
    );
}
```

base case

recursive case