Linked Lists : intro

The problems with vector

Vector has worked well for us so far

- Dynamically sized
- Can contain any type
- Convenient access functions

But there are some things it can't (easily) do

We have `push_back` but where is `push_front`?

A naive push_front

```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

A naive push_front

4	3	8	1	4	3	8	1	7

```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

A naive push_front

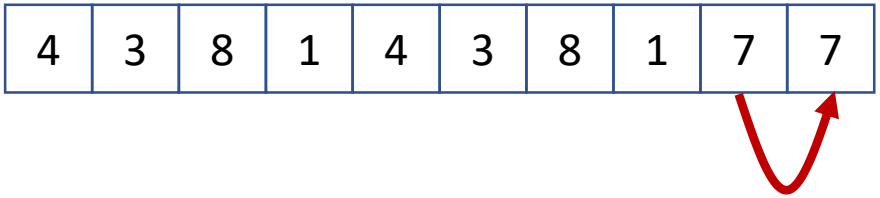
4	3	8	1	4	3	8	1	7	0

```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

We're ignoring the cost of resize here: see last weeks lab for discussion of efficiency

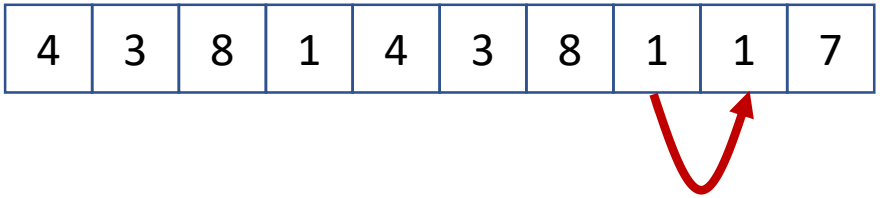
A naive push_front




```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

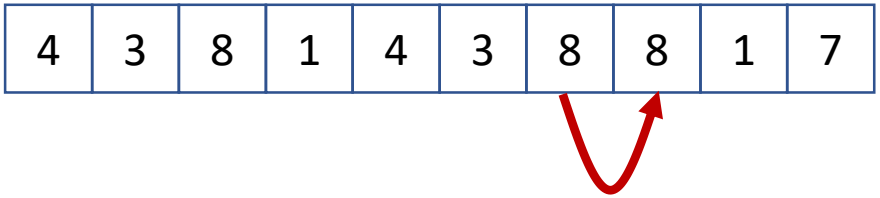
A naive push_front




```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

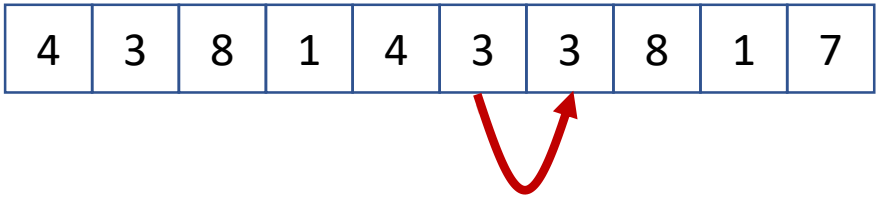
A naive push_front




```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

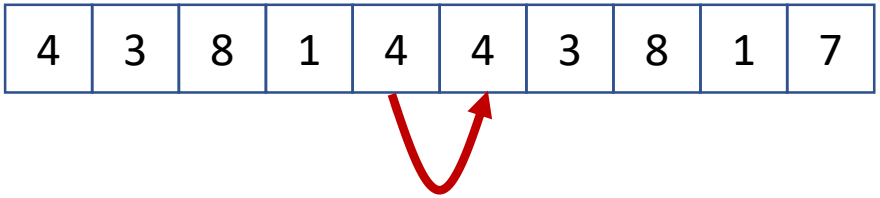
A naive push_front




```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

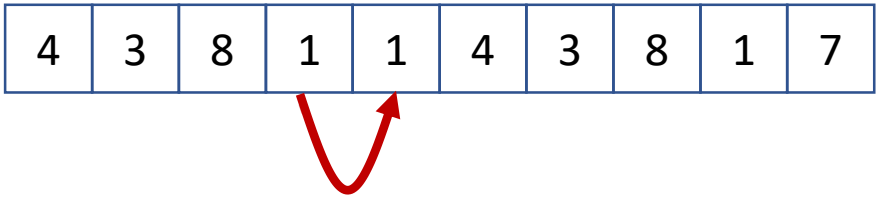
A naive push_front




```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

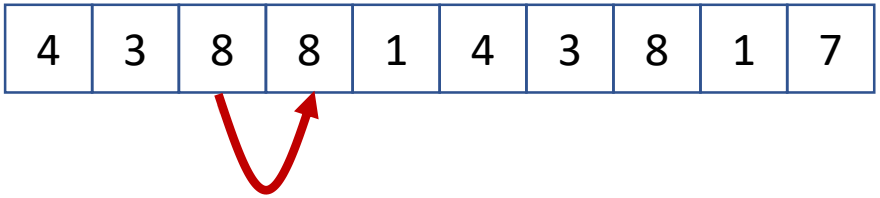
A naive push_front




```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

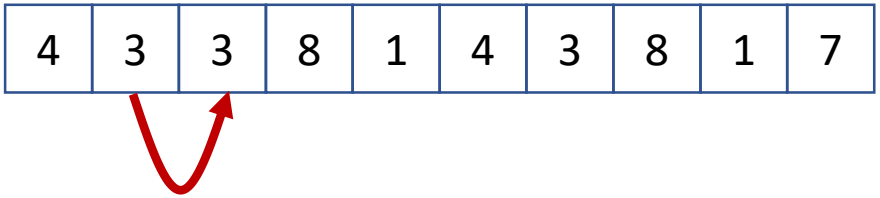
A naive push_front




```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

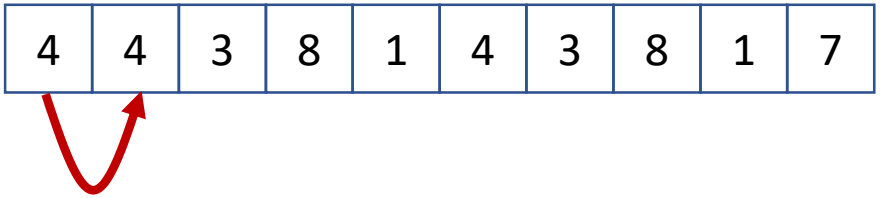
A naive push_front




```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

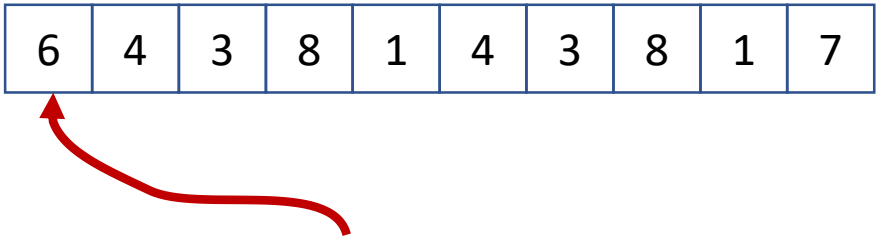
A naive push_front




```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

A naive push_front




```
void push_front(my_int_vec *v, int x)
{
    // Resize the vector v
    resize(v, size(v)+1);

    // Move all the existing values
    for(int i=size(v)-1; i>0; i--){
        write(v, i, read(v,i-1));
    }

    // Push the new value at front
    write(res, 0, x);
}
```

```
void push_front(vector<int> *v, int x)
{
    // Resize the vector v
    v->resize(v->size()+1);

    // Move all the existing values
    for(int i=v->size()-1; i>0; i--){
        (*v)[i] = (*v)[i-1];
    }

    // Push the new value at front
    (*v)[0] = x;
}
```

Functionality versus performance

- The ***functionality*** of push_front is fine
  - It does exactly what we want

- The ***performance*** of push_front is terrible
  - Takes *roughly* n operations to push_front 1 item
  - Takes *roughly* n^2 operations to push_front n items

- The API of `vector<T>` is carefully designed
  - Exposes everything the vector is good at
  - Tries to hide or make difficult the weak operations

An alternative : `list<T>`

- The C++ library contains multiple ***containers***
  - Each container provides different functionality
  - Each container provides different performance

- An example is `list<T>` : manages a sequence of `T`
  - Has an efficient implementation of `push_front` built in
  - ***But***: there is no array-like indexing through `[.]`

- Selecting the right container can be important
  - Use `vector<T>` -> program takes one year
  - Use `list<T>` -> program takes one second

  Or for another application it could be the opposite

Implementation of `list<T>`

Internally list<T> is implemented as a ***linked list***

A linked list consists of nodes, where each node has:

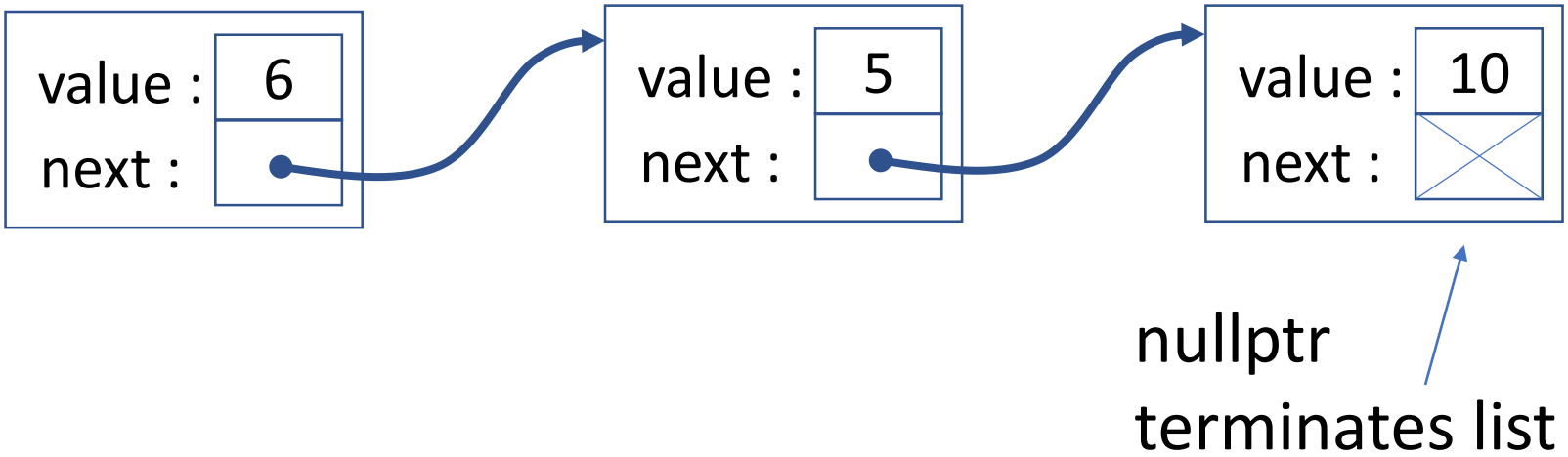
- A value
- A pointer to the next node

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

Technically list<T> is a doubly linked list, but that's not relevant yet.

Linked lists : chains of nodes

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```



Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```

Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```

p:

Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```

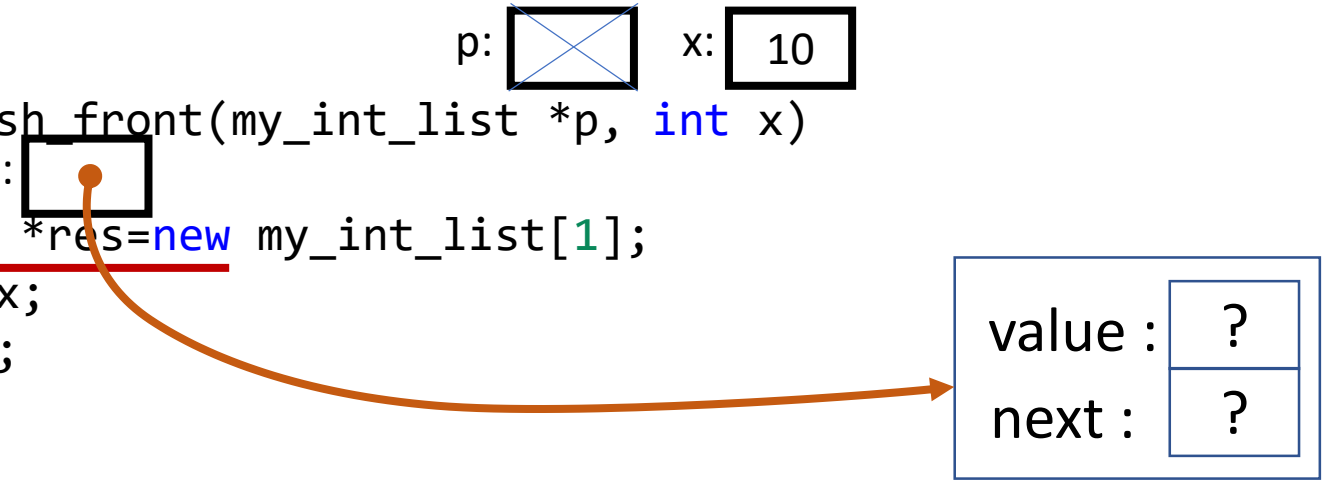
p: x: 10

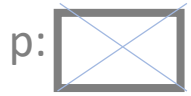
res: ?

p:

Building a linked list

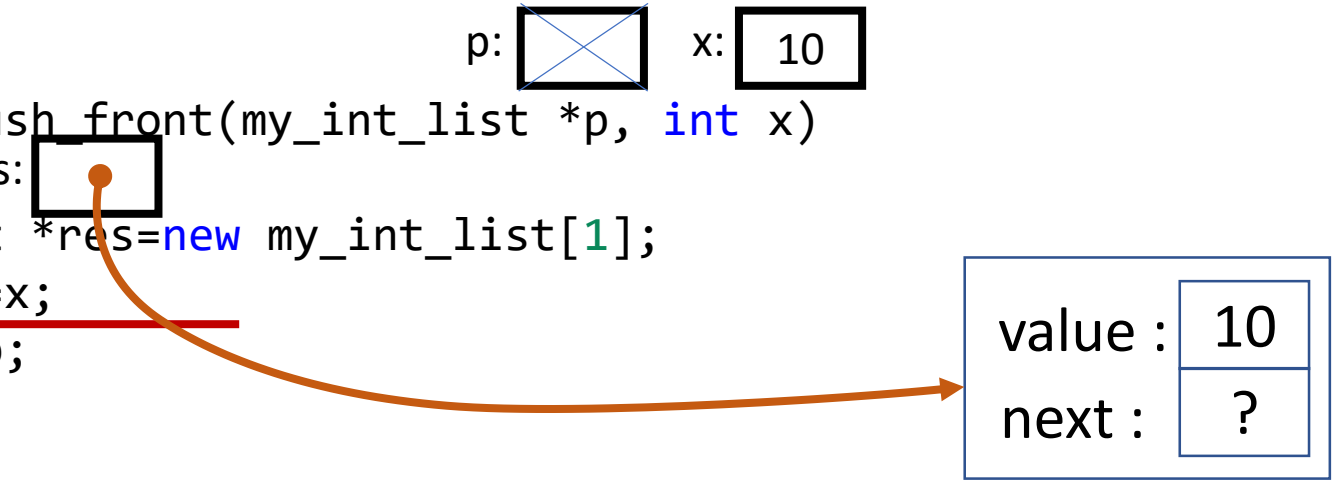
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```

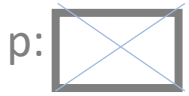




Building a linked list

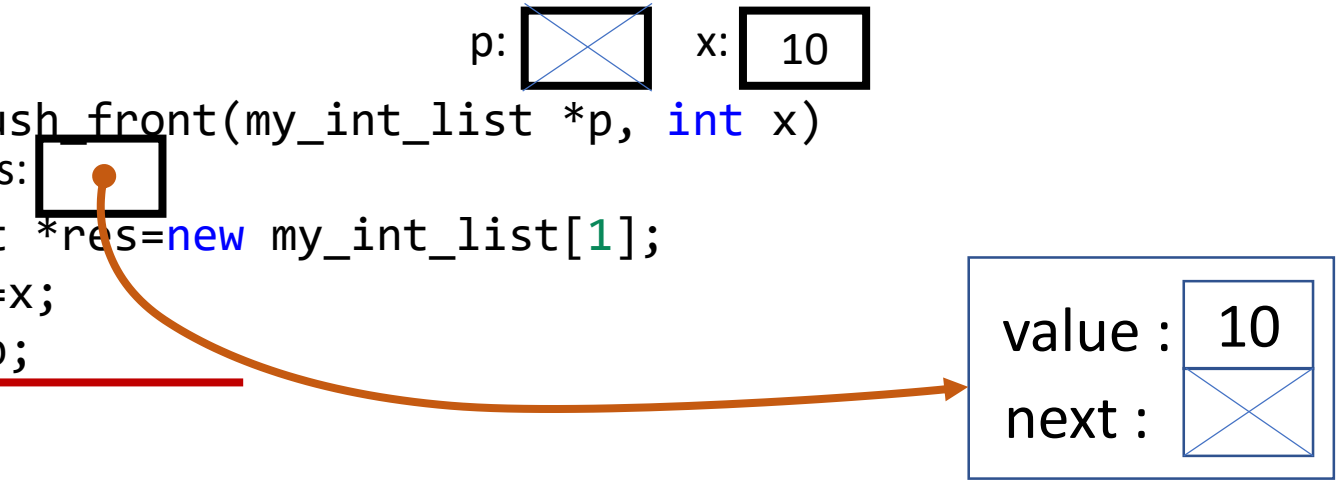
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```

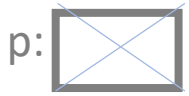




Building a linked list

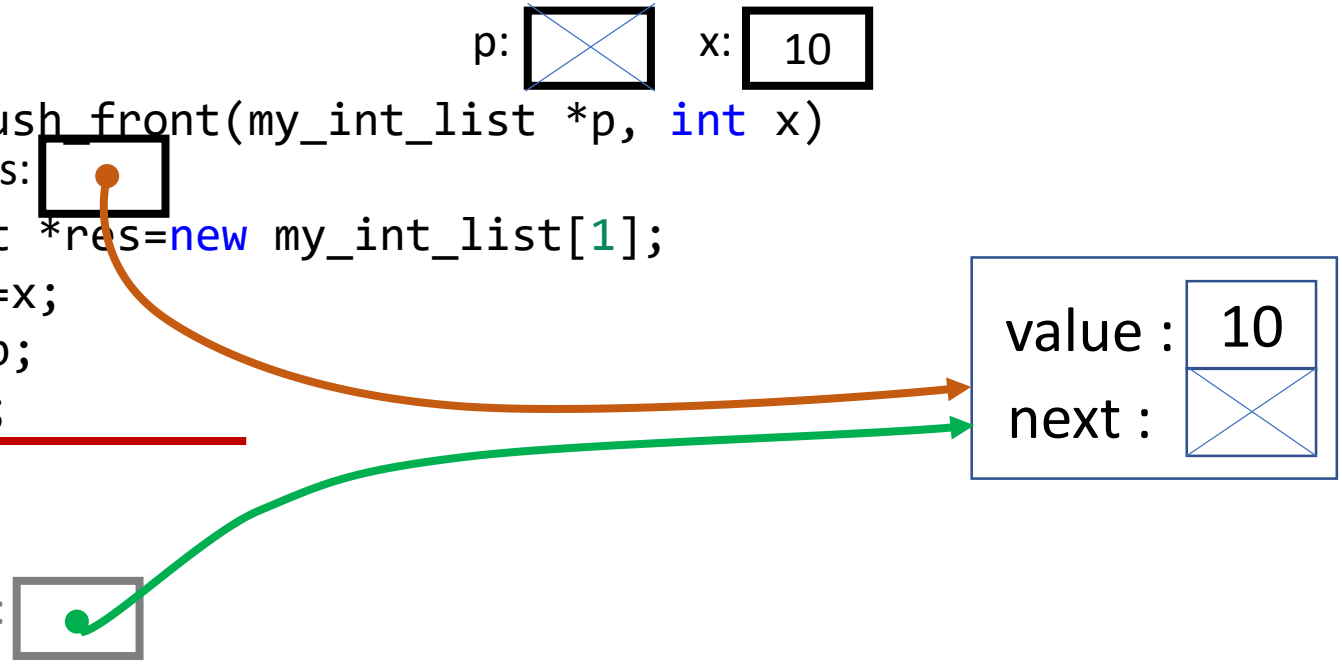
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```





Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```

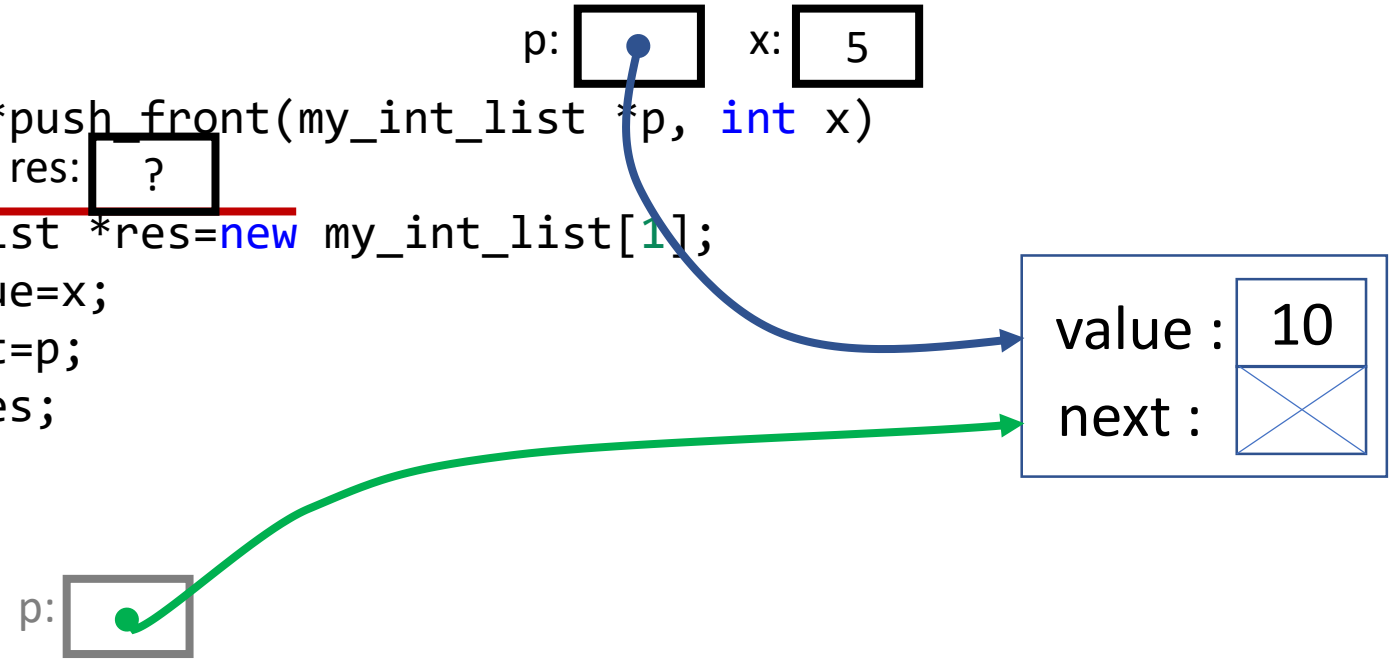


Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```

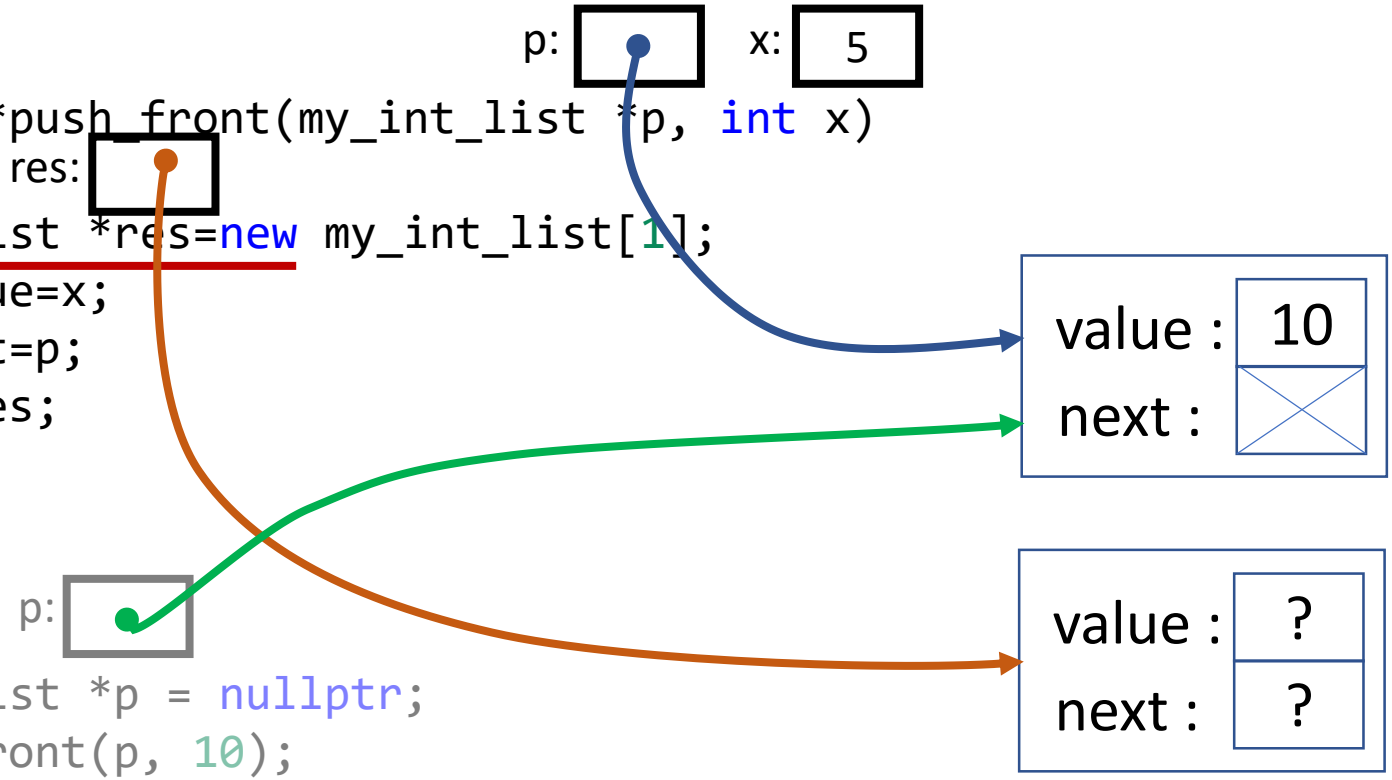
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



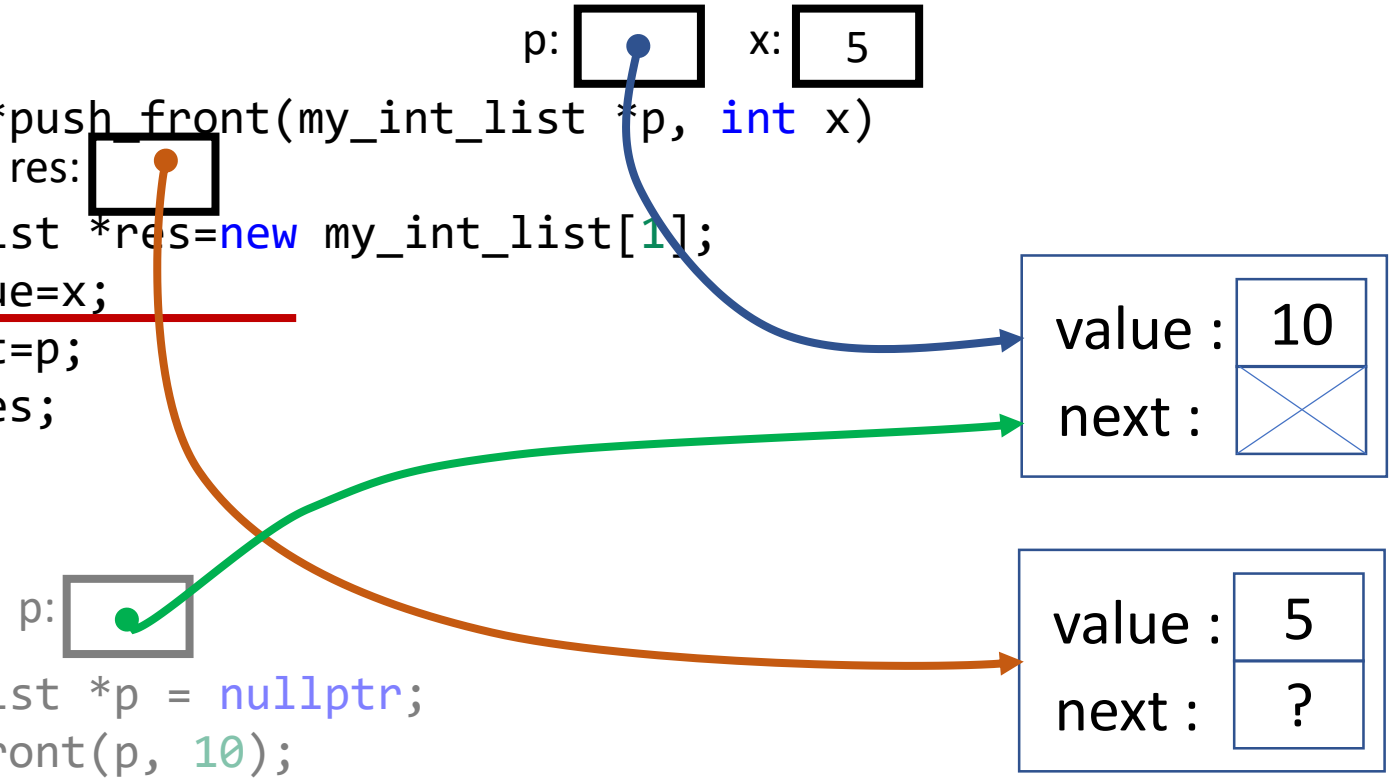
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



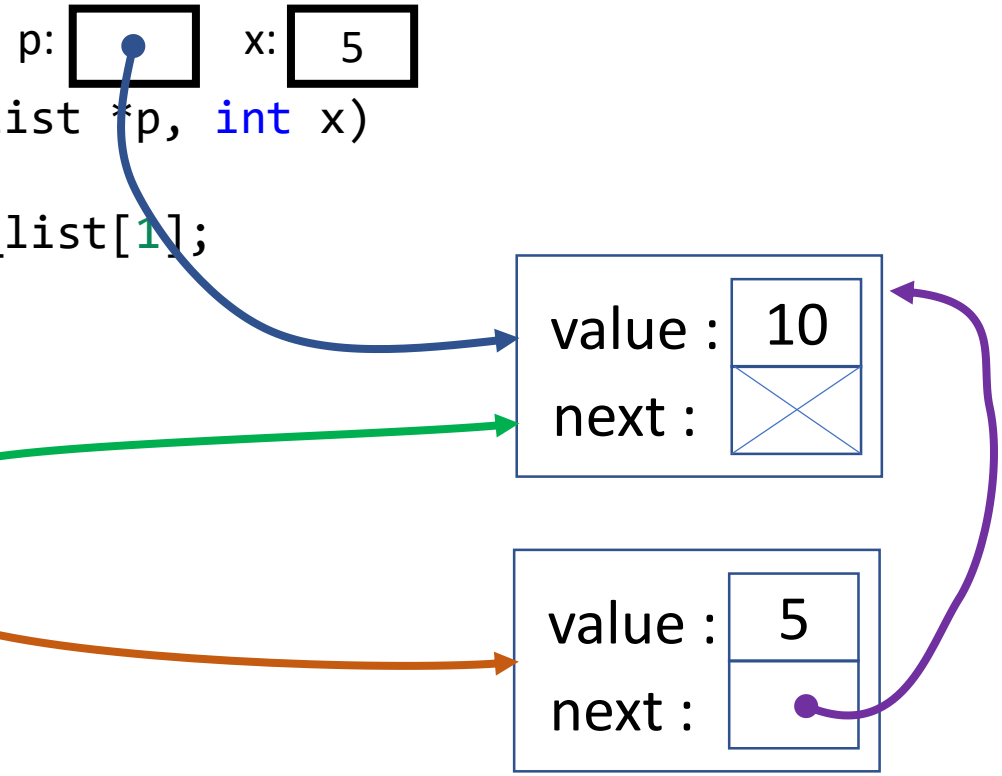
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



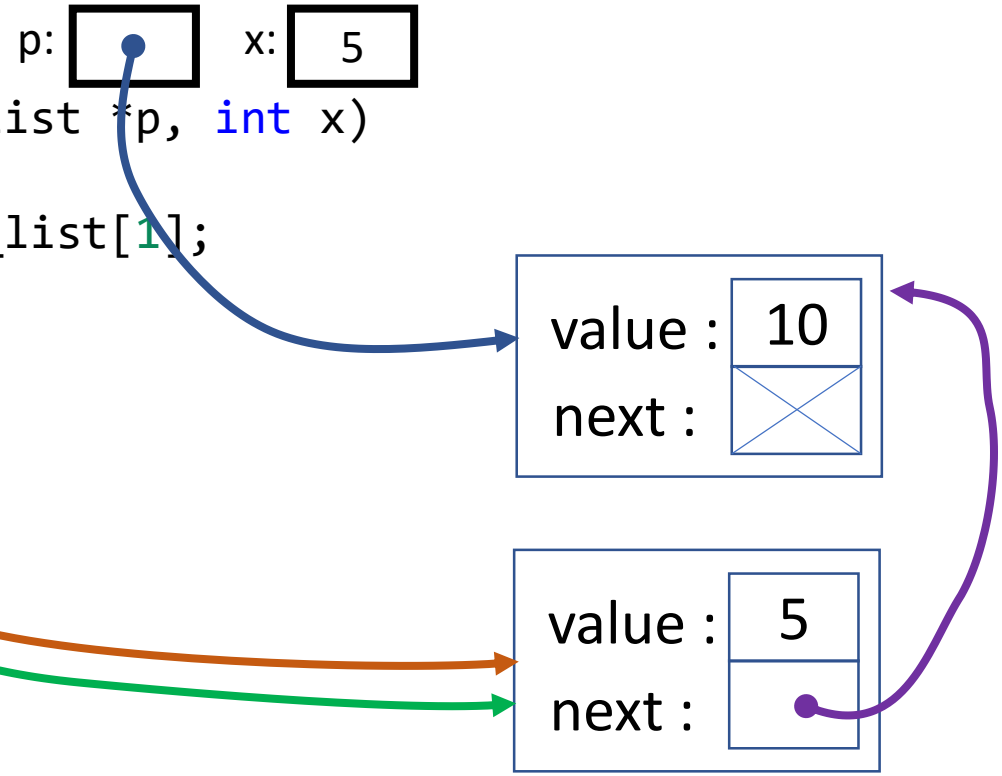
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



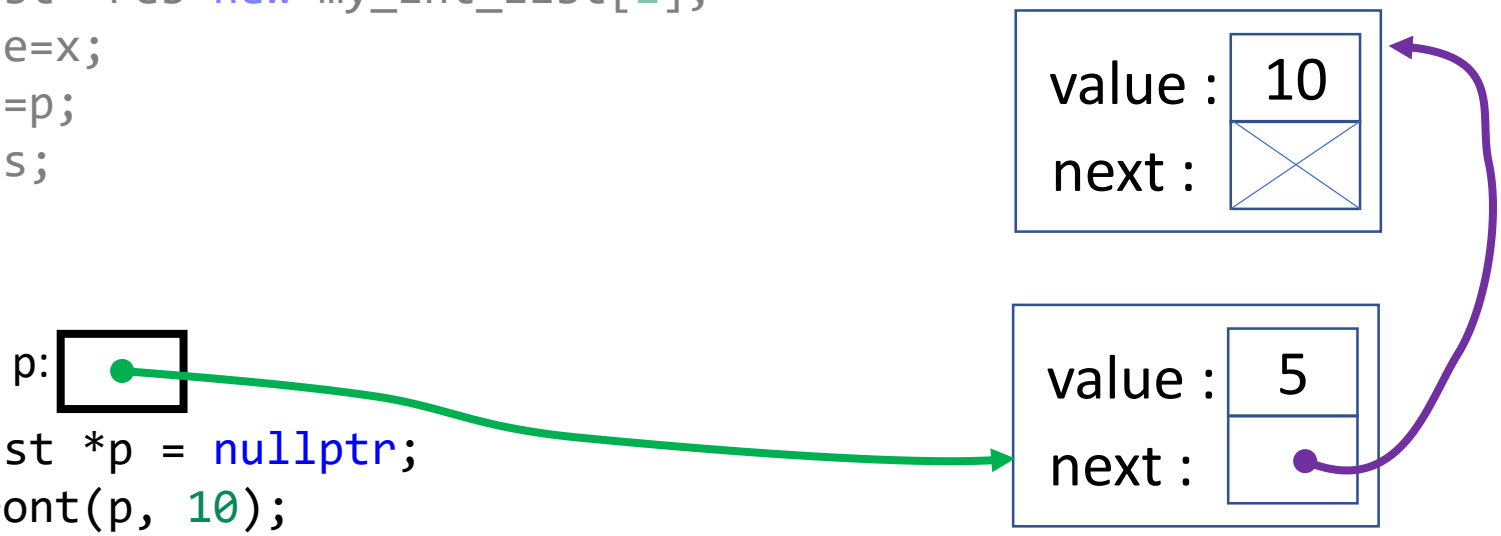
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



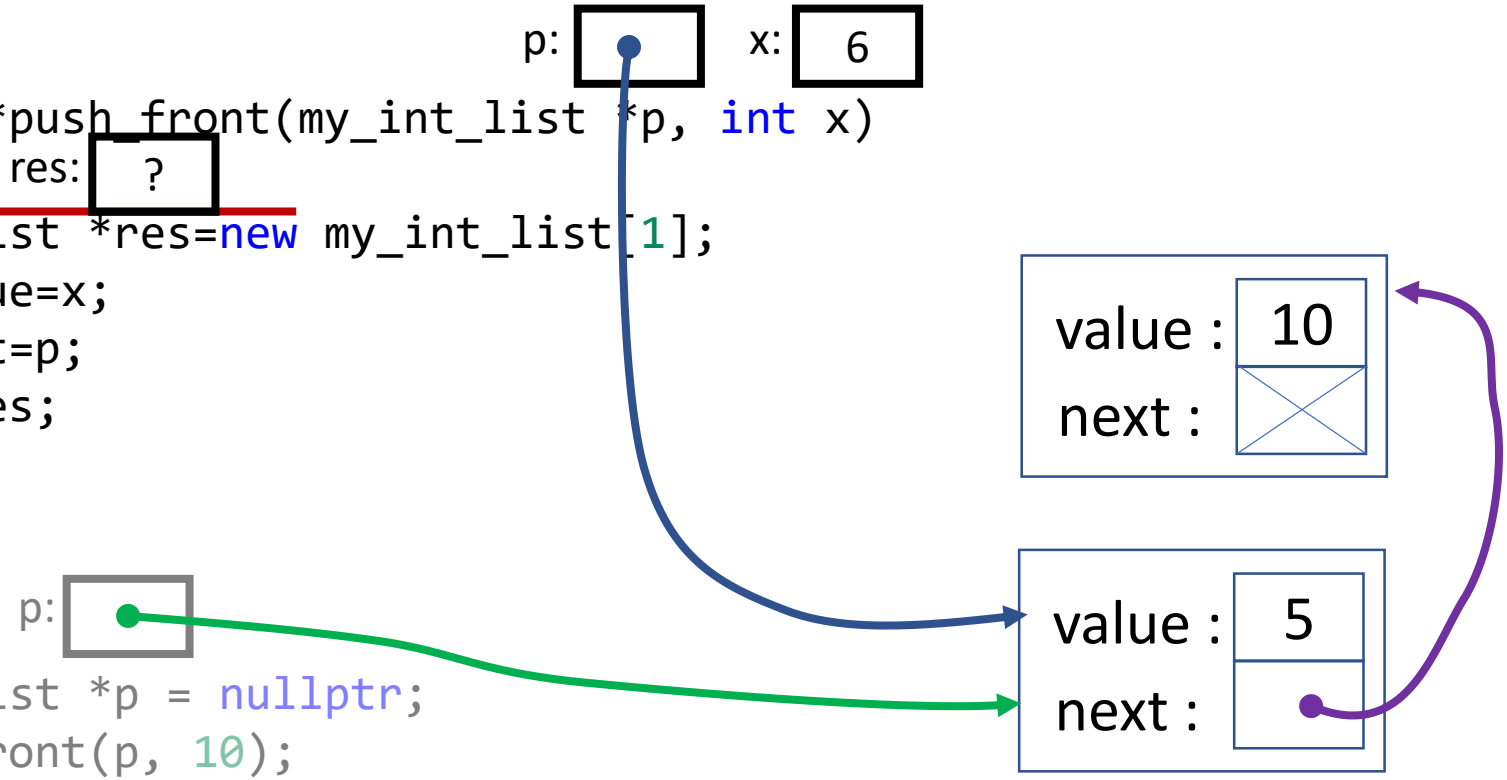
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



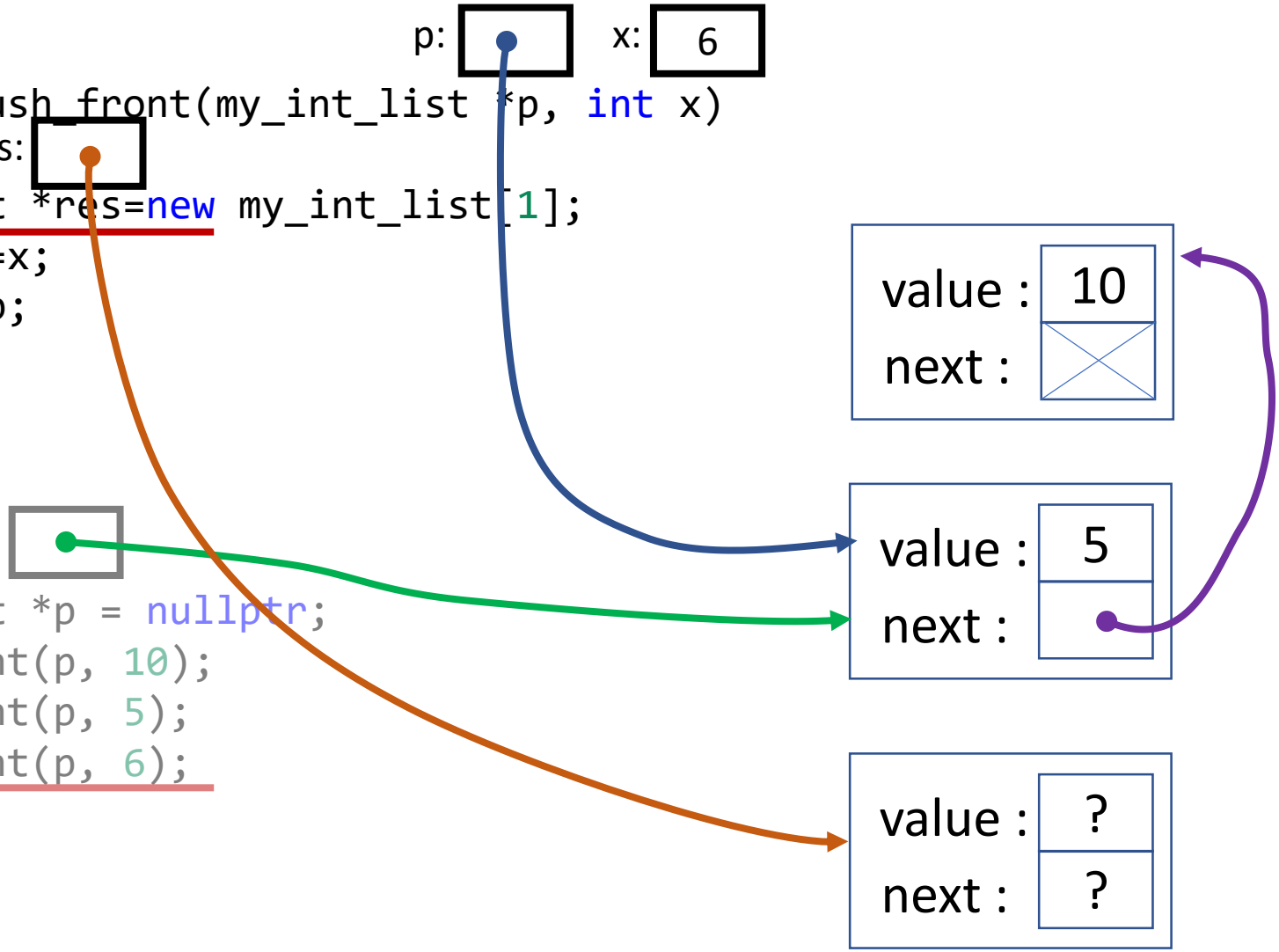
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



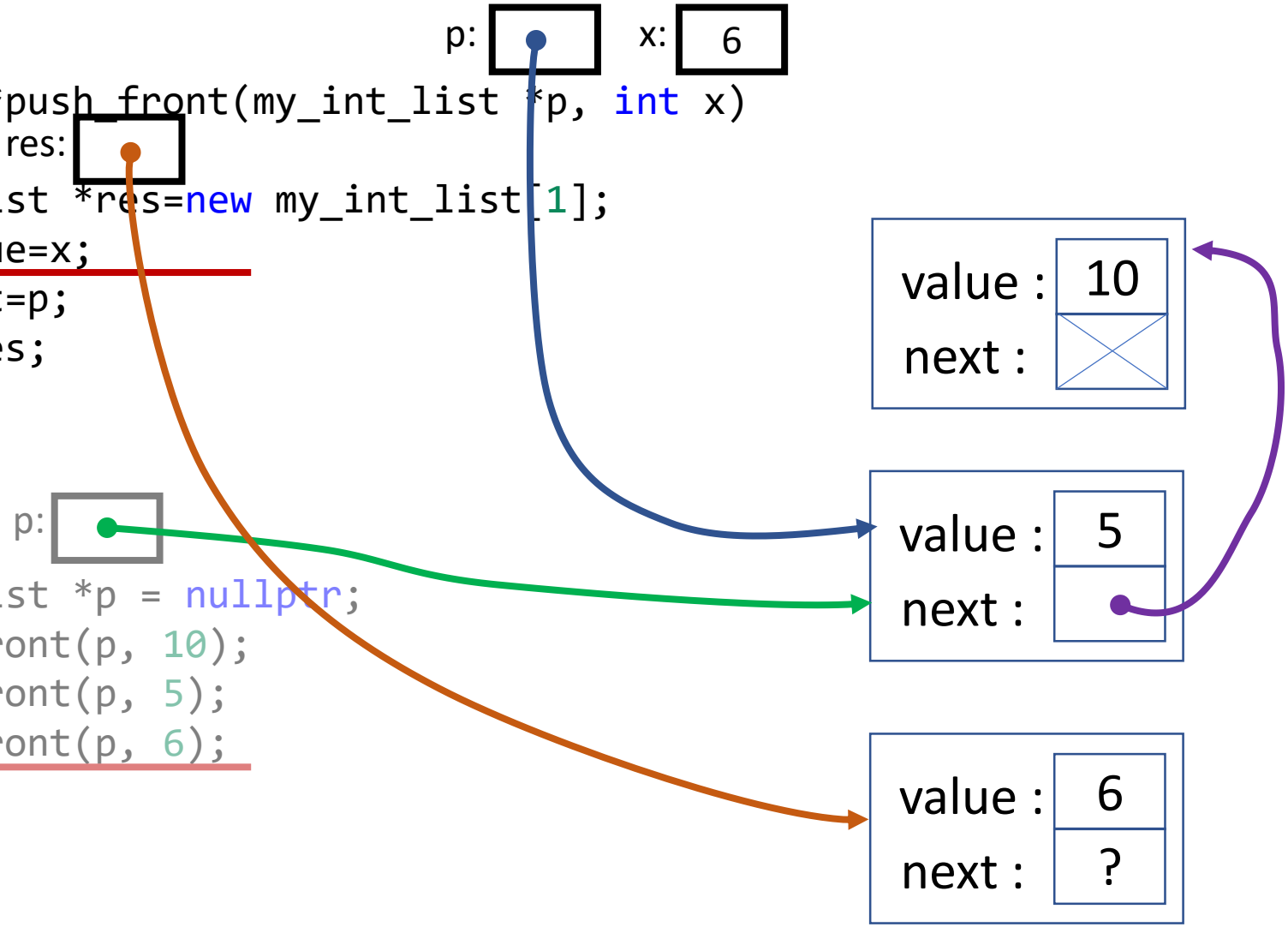
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



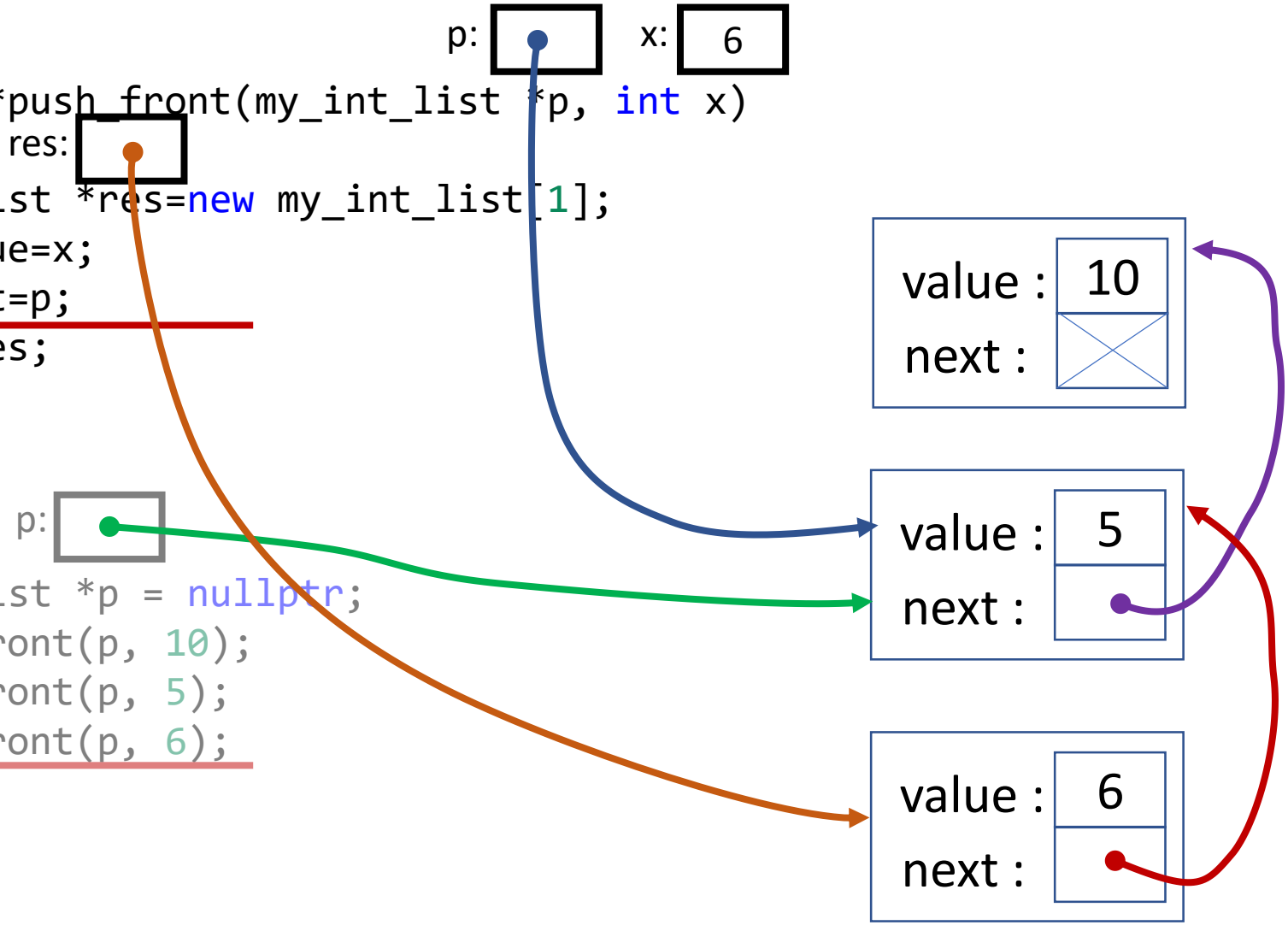
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



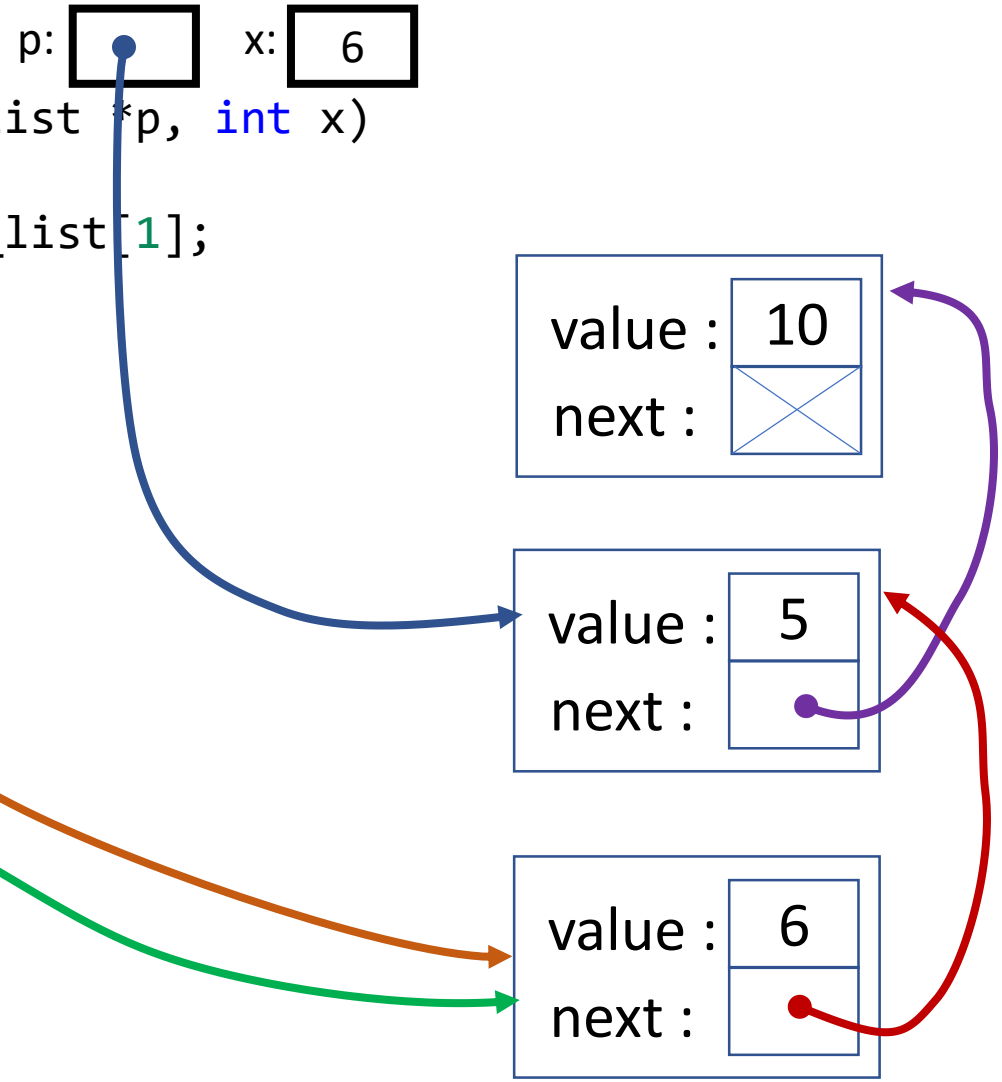
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



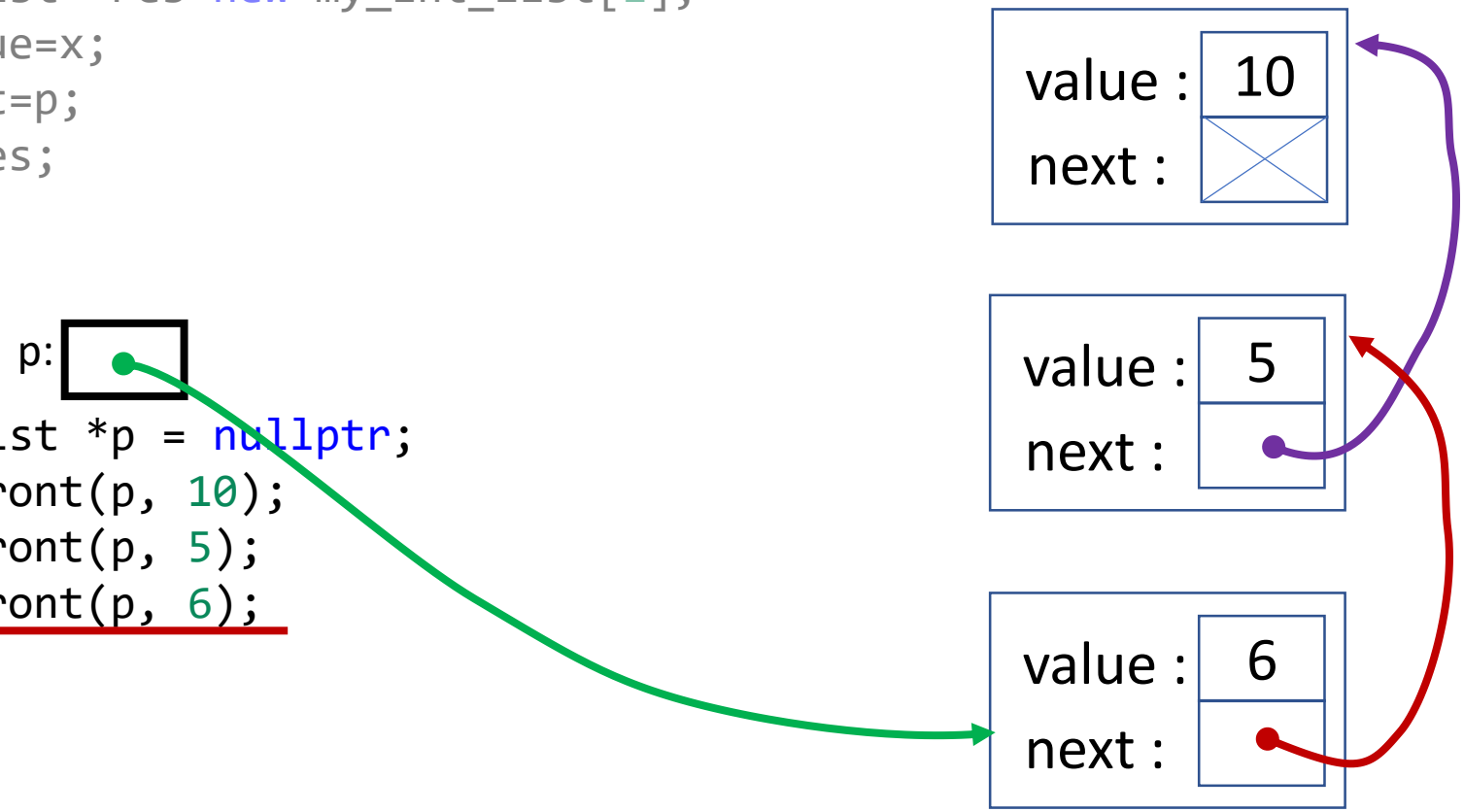
Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



Building a linked list

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

int main()
{
    my_int_list *p = nullptr;
    p=push_front(p, 10);
    p=push_front(p, 5);
    p=push_front(p, 6);
}
```



Lists as a recursive type

- A list can be one of two things:
  - An empty list: nullptr
  - A value plus a list : { value , list }

- A recursive function is one of two things:
  - A base case
  - Calculations plus a recursive call

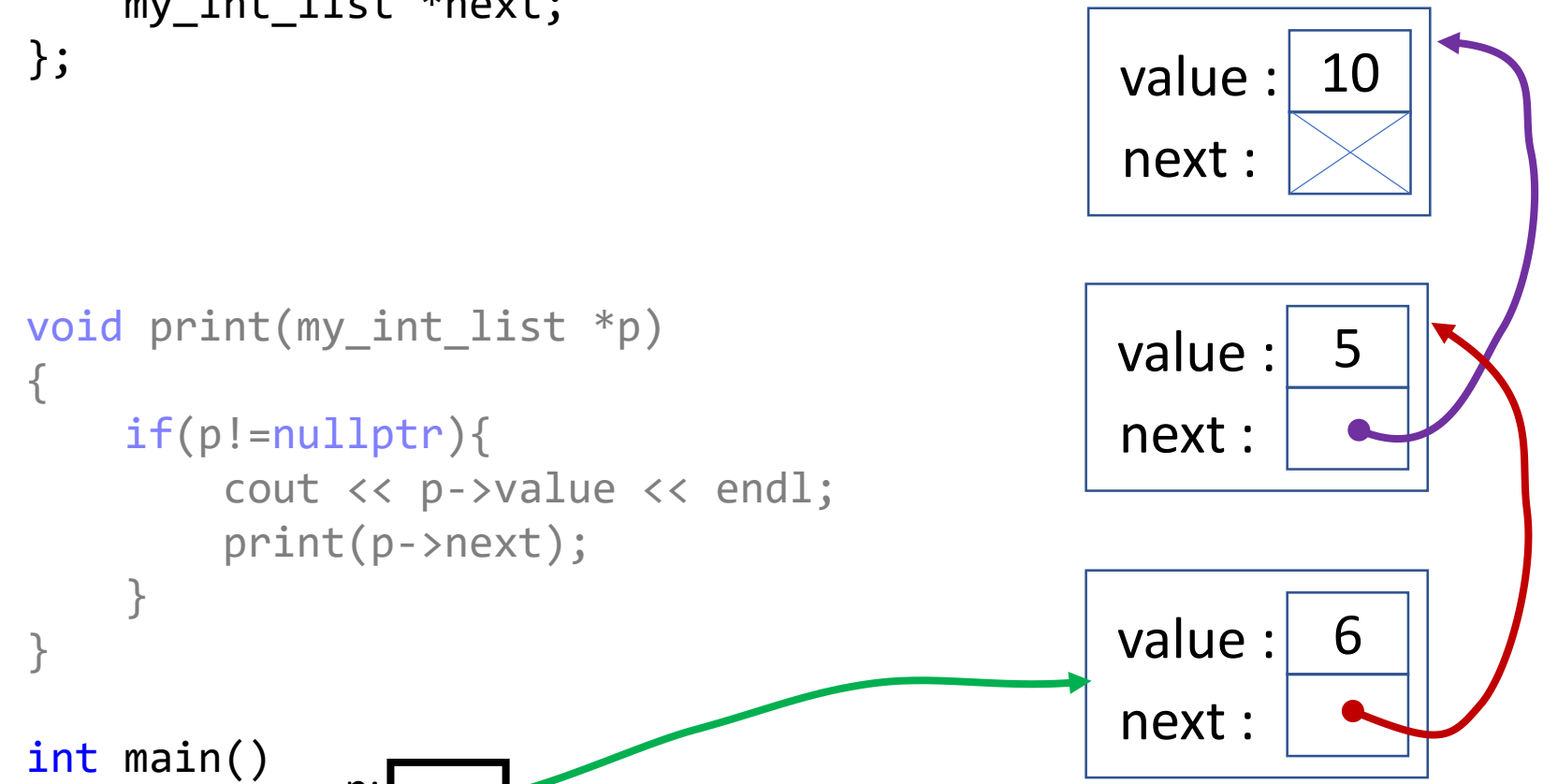
Printing: forwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};




void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

Printing: forwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

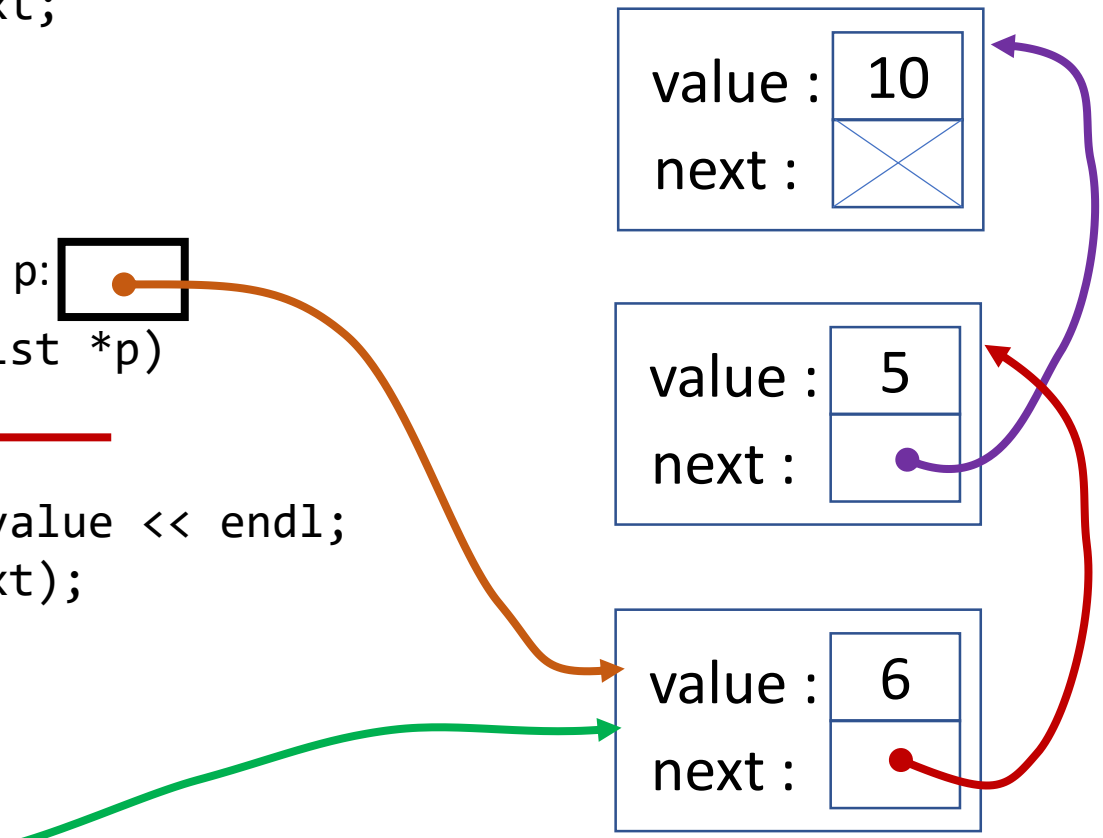



```
$ ./print-list
```

Printing: forwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

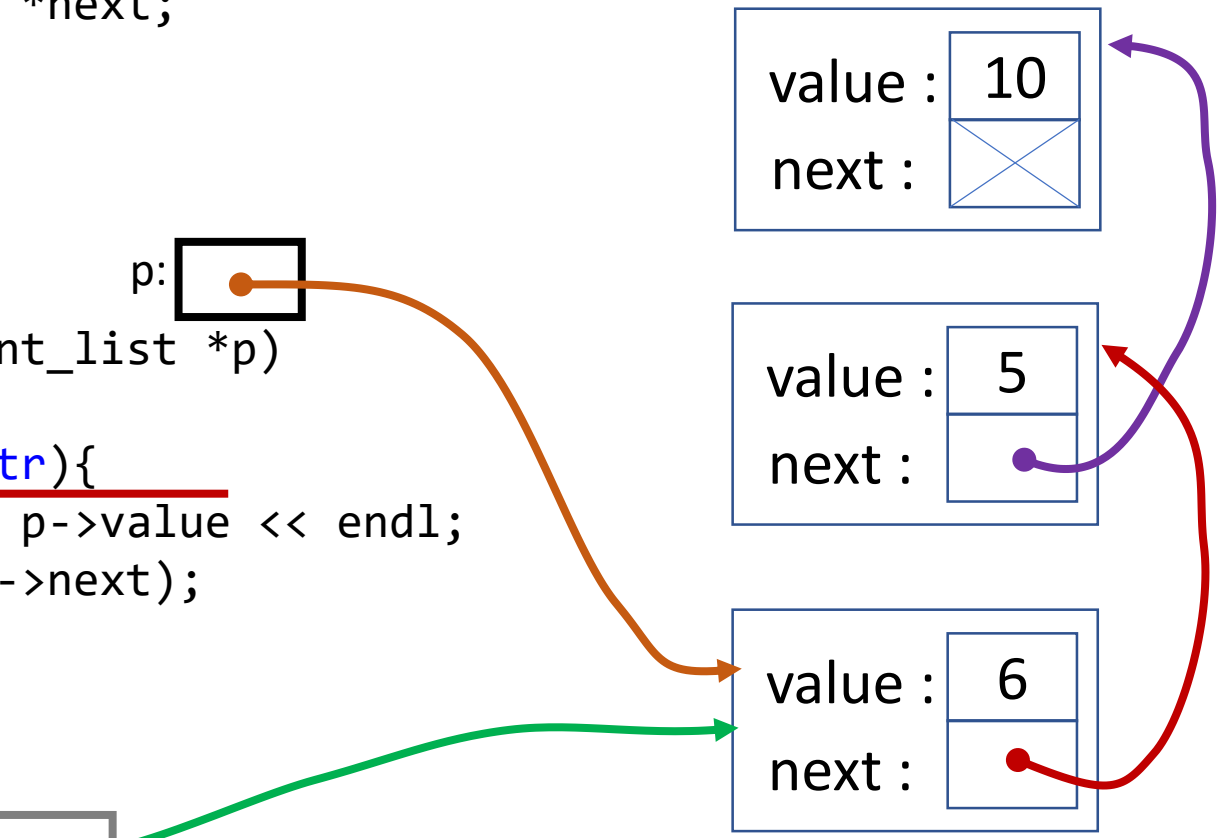



```
$ ./print-list
```

Printing: forwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

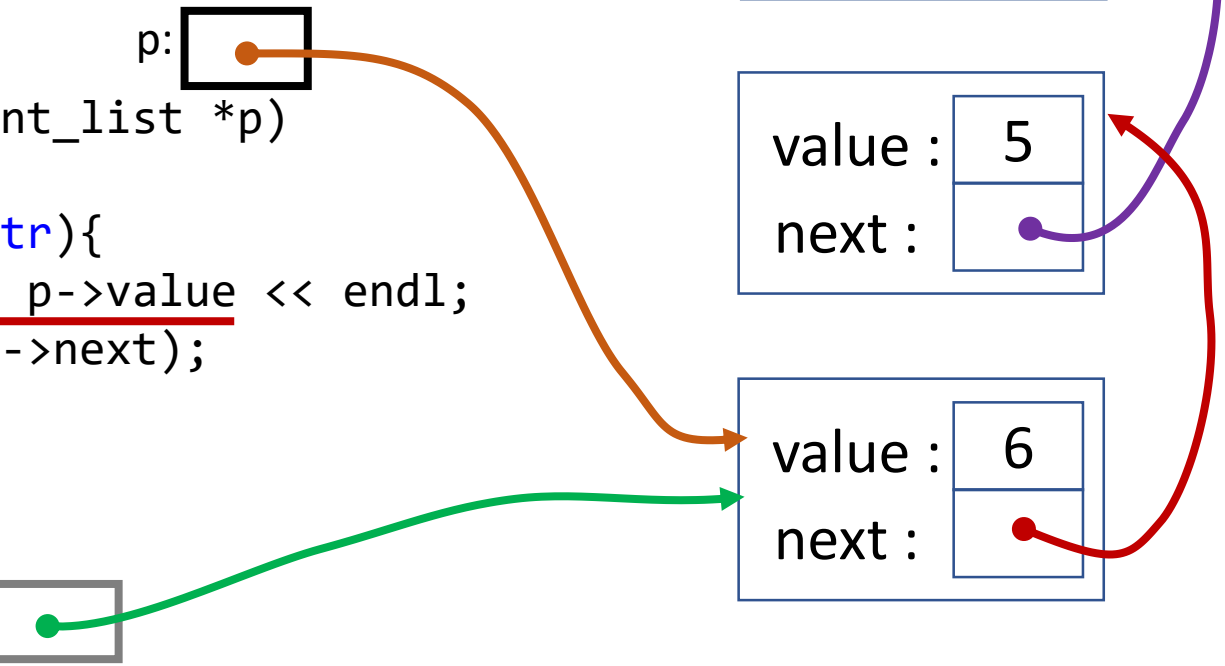



```
$ ./print-list
```

Printing: forwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

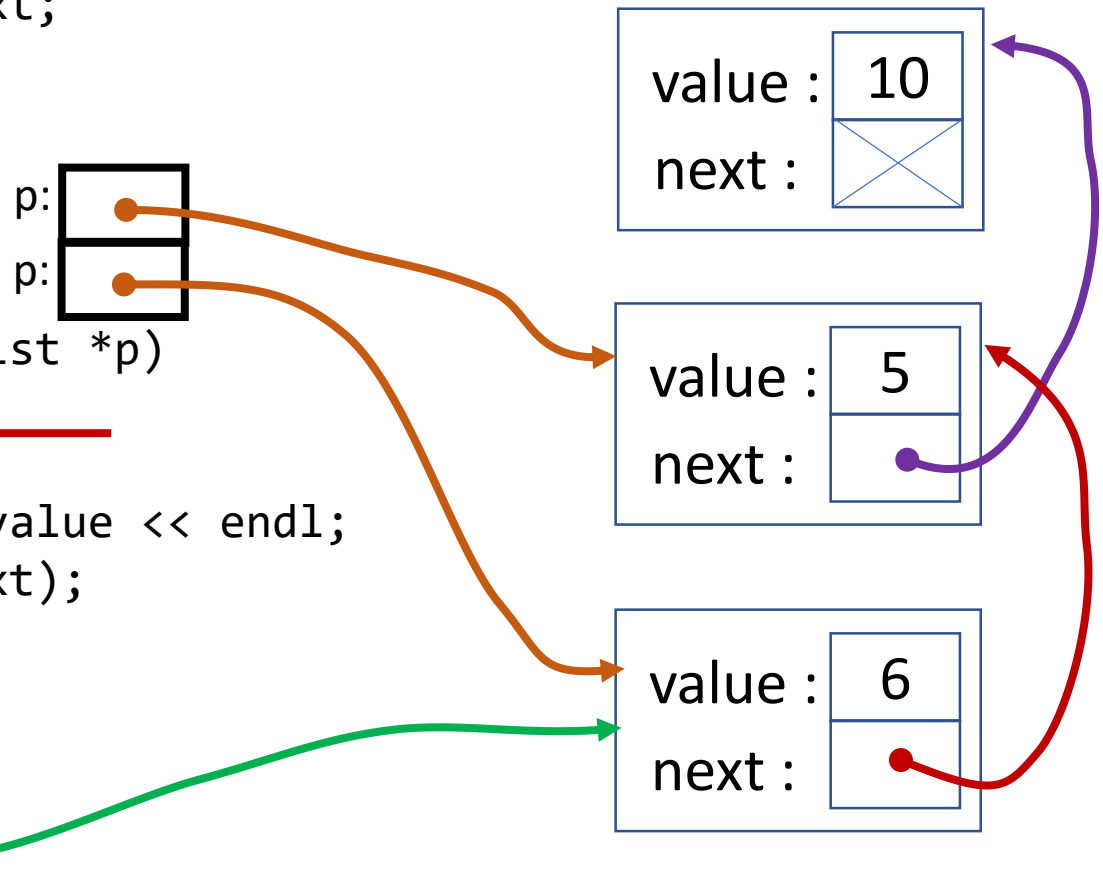
```
void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
```

Printing: forwards

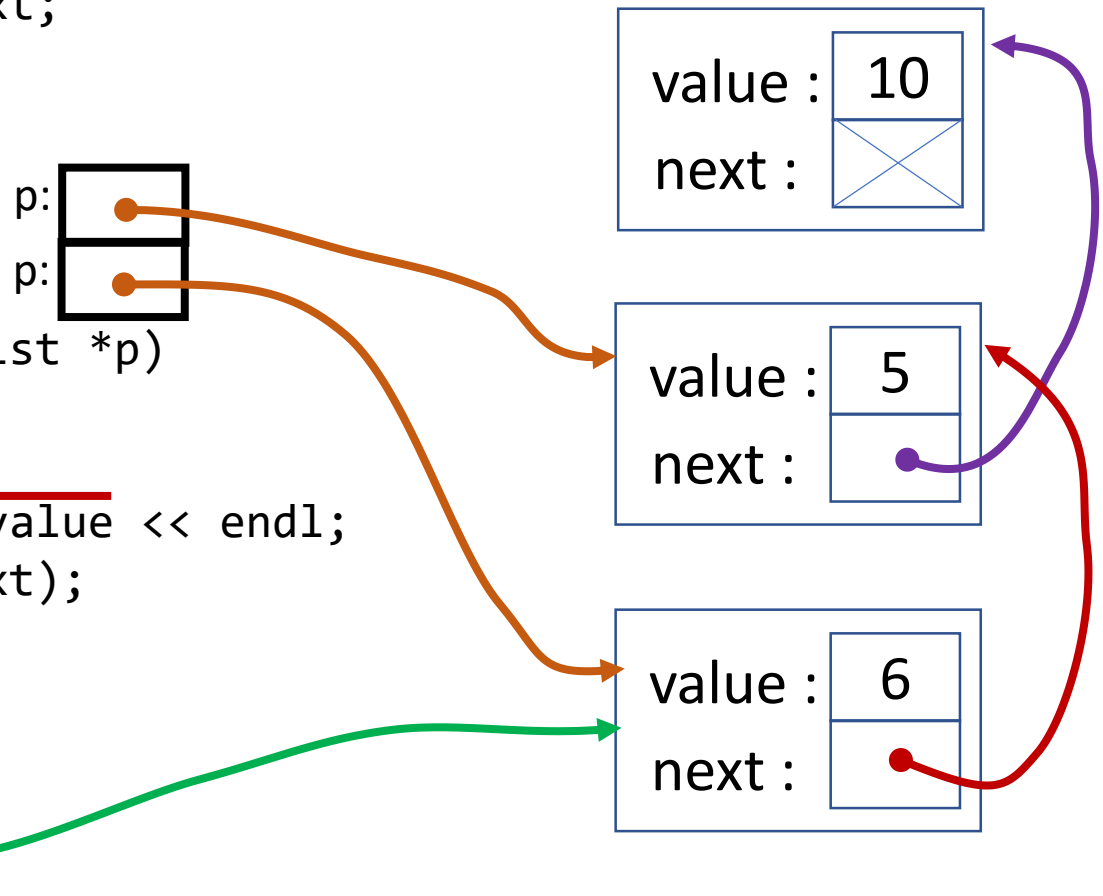
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
```

Printing: forwards

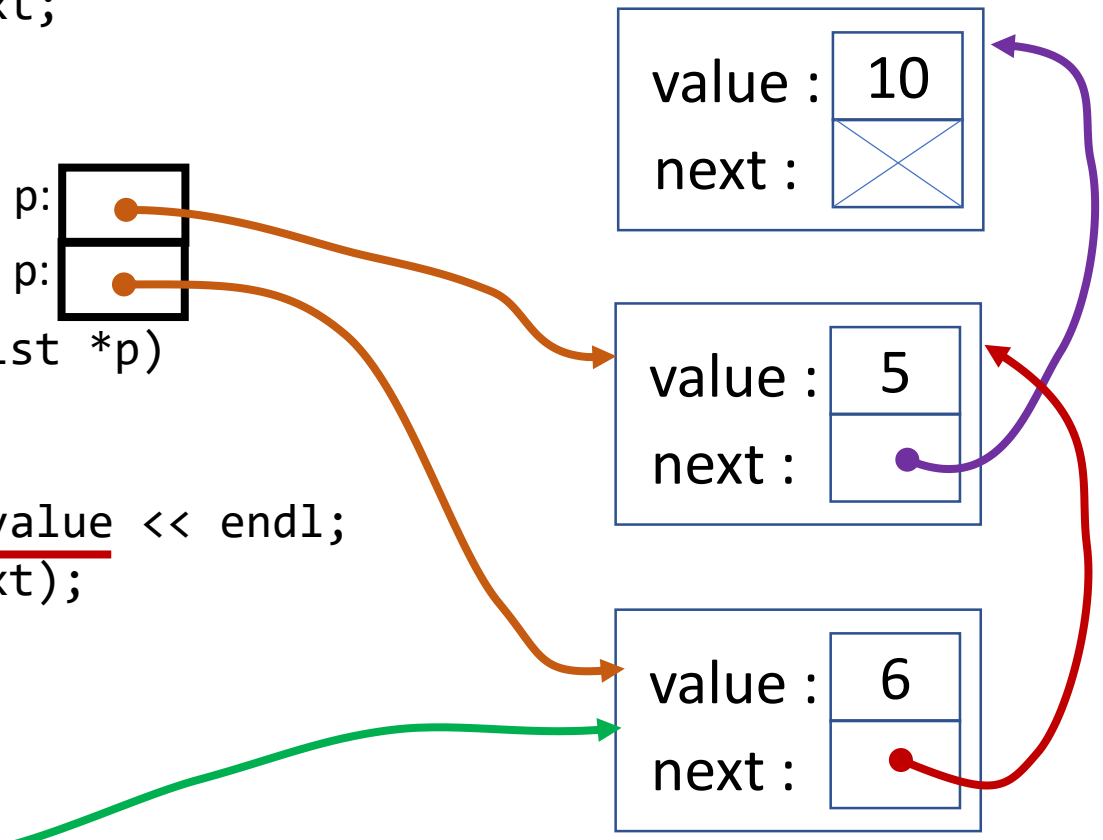
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
```

Printing: forwards

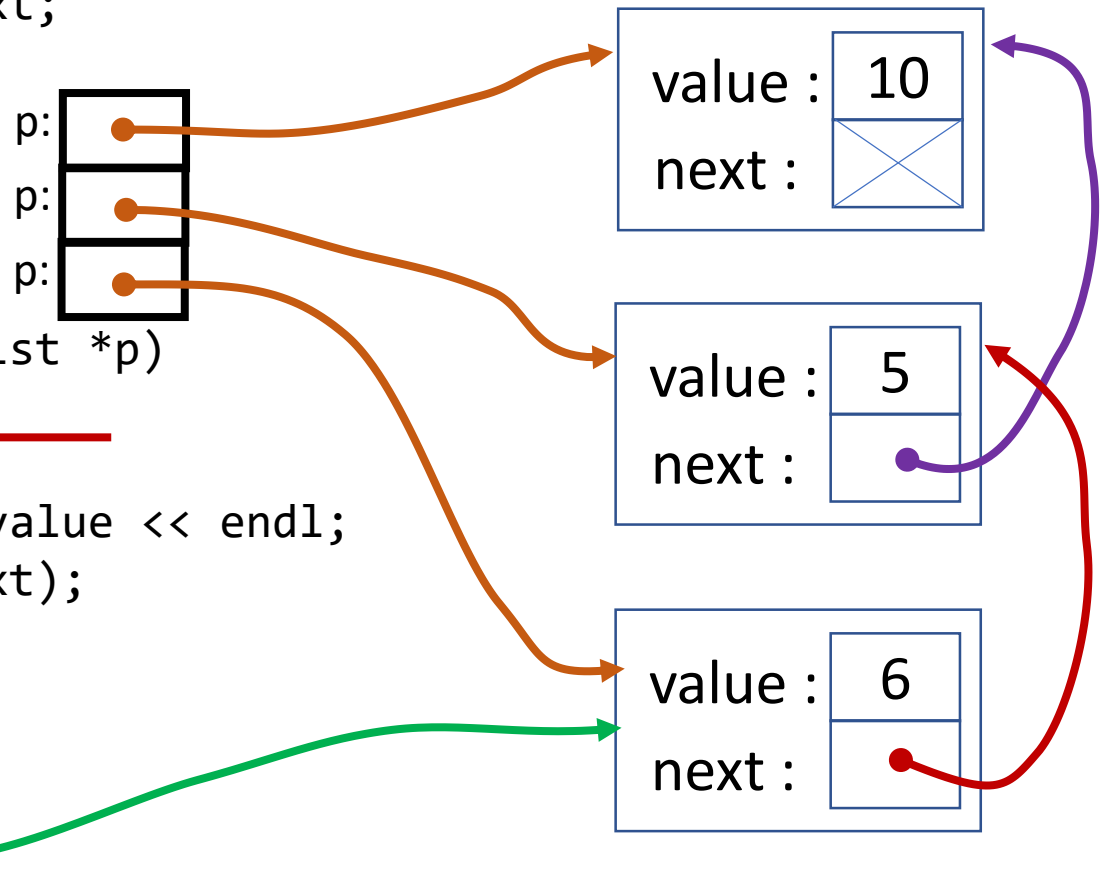
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
5
```

Printing: forwards

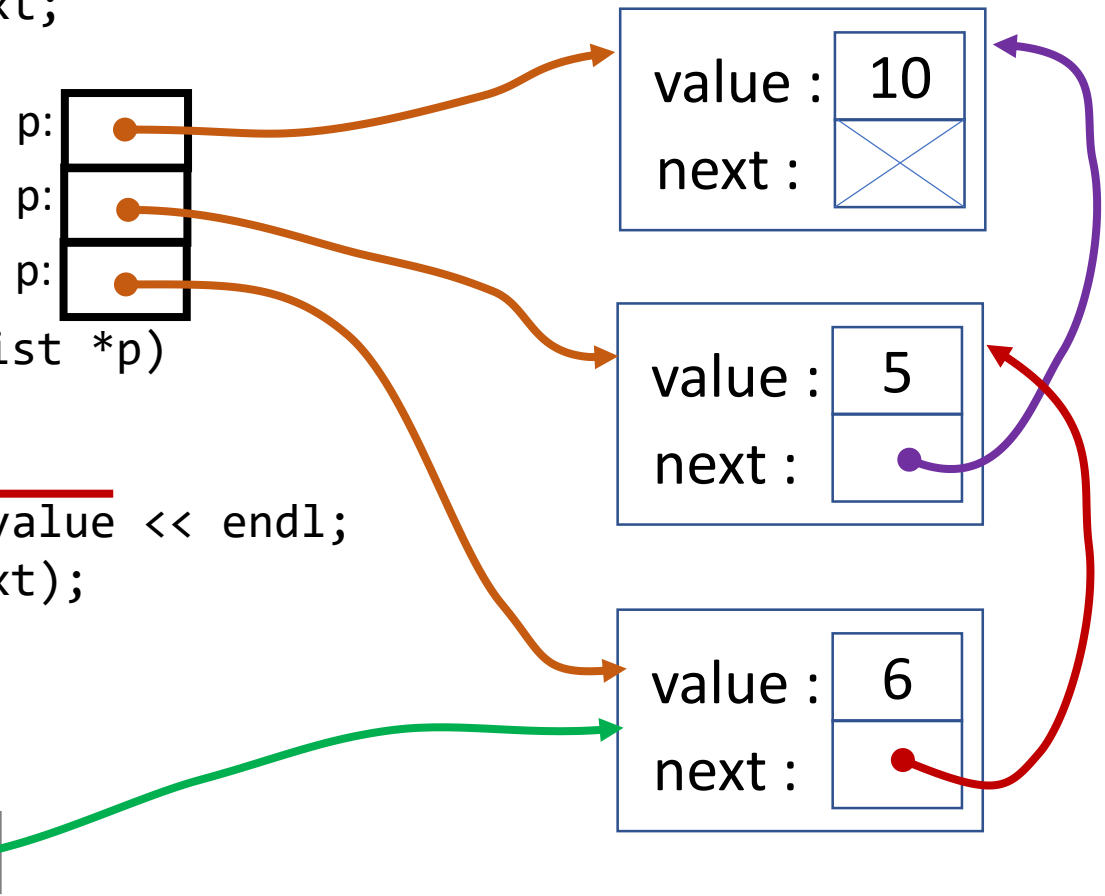
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
5
```

Printing: forwards

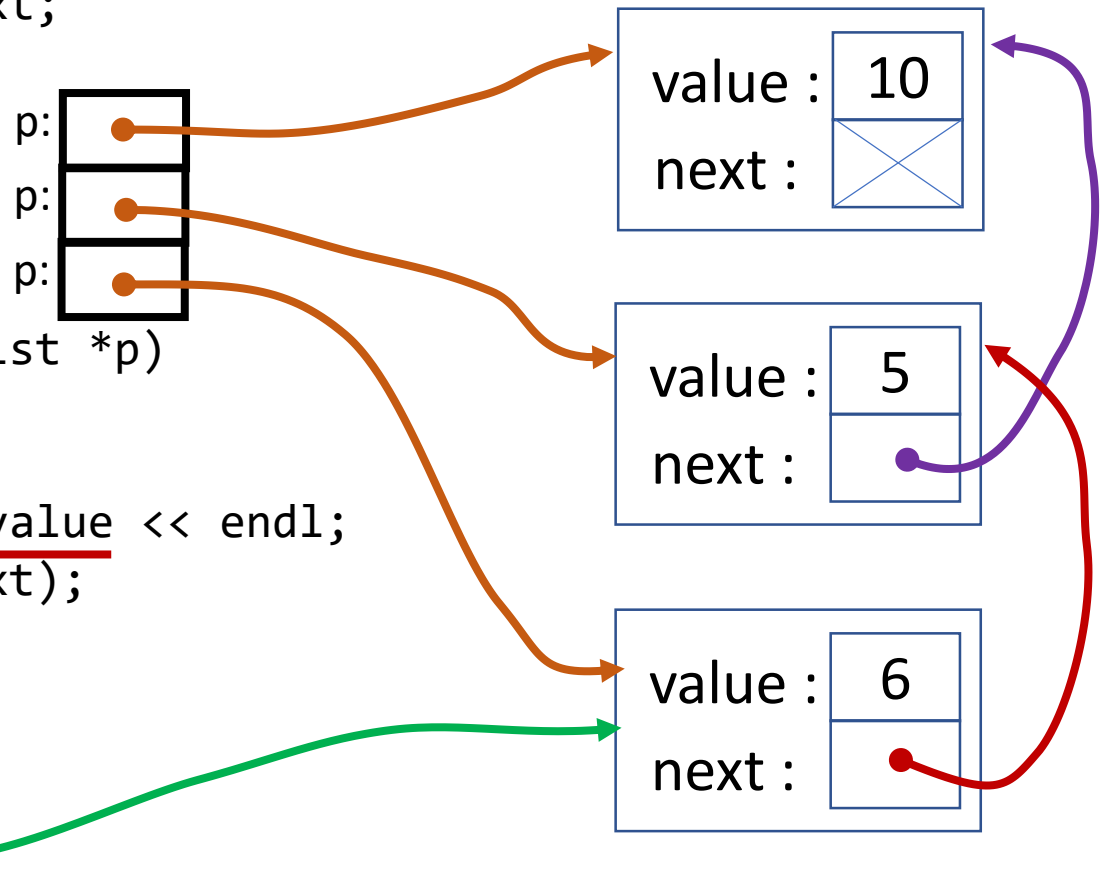
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
5
```

Printing: forwards

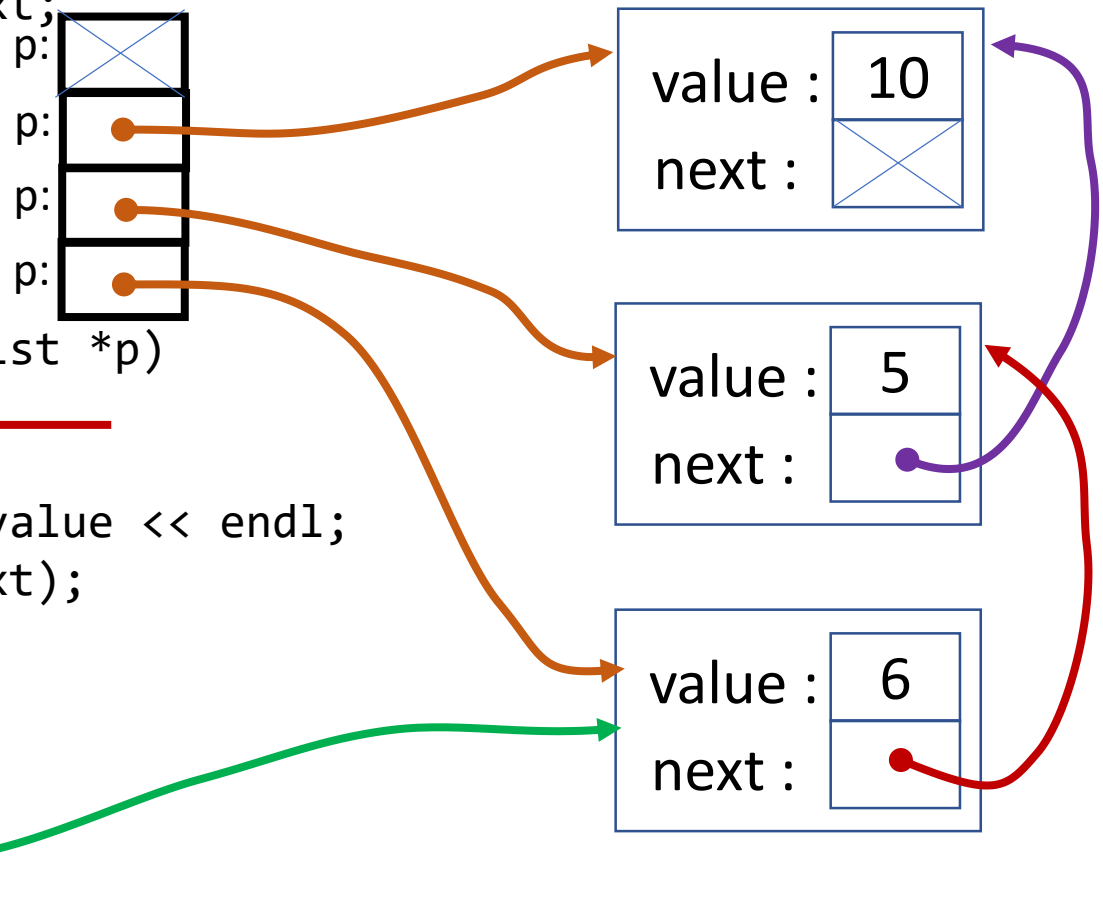
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
5
10
```

Printing: forwards

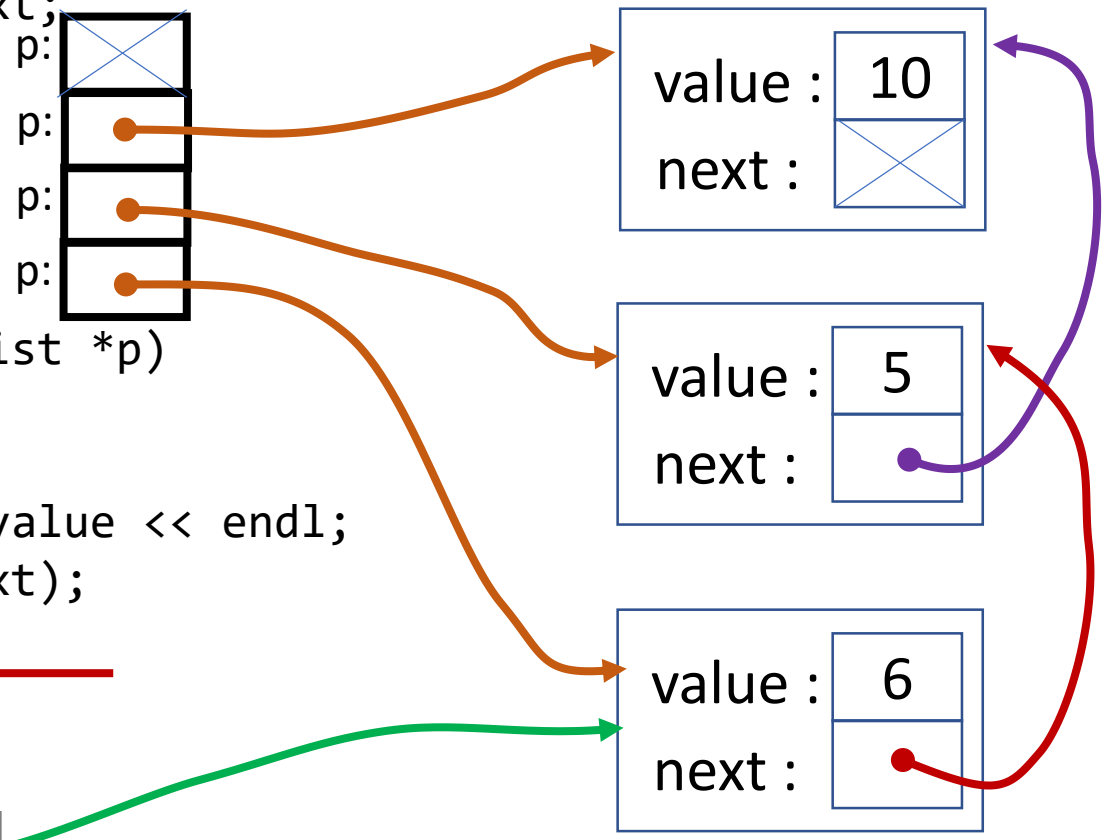
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
5
10
```

Printing: forwards

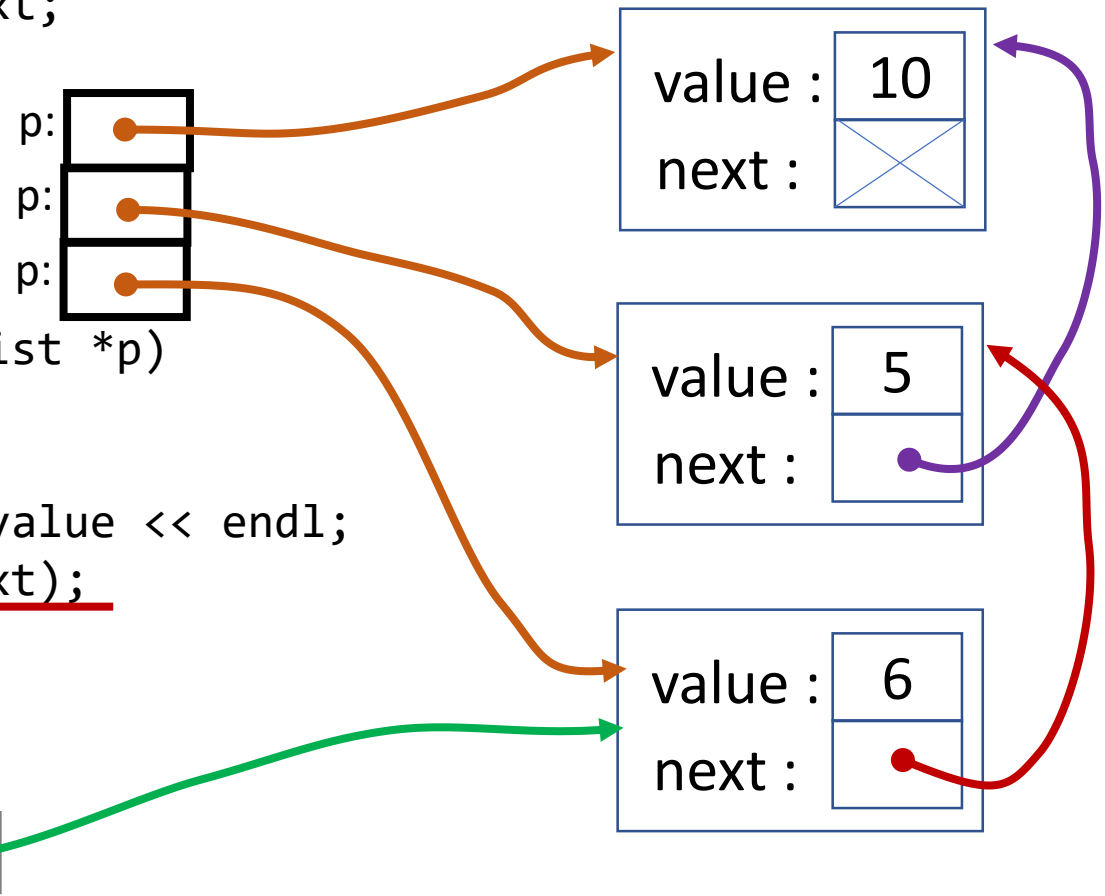
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
5
10
```

Printing: forwards

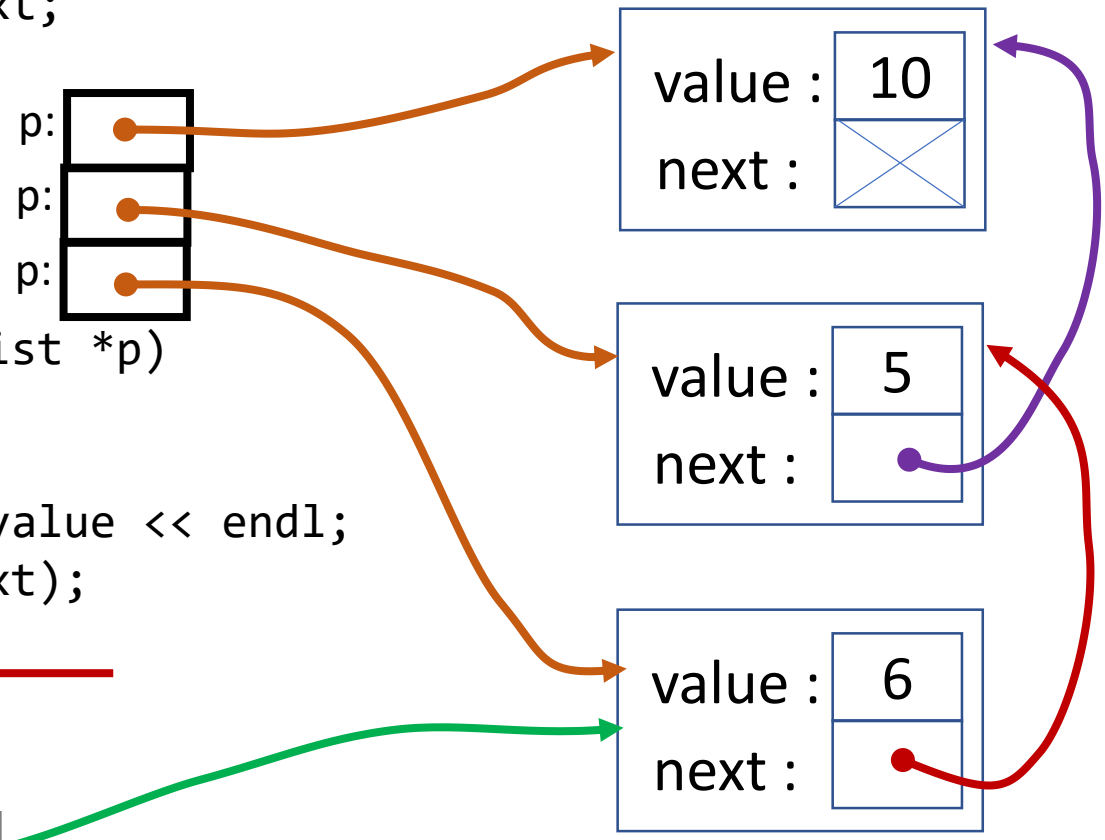
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
5
10
```

Printing: forwards

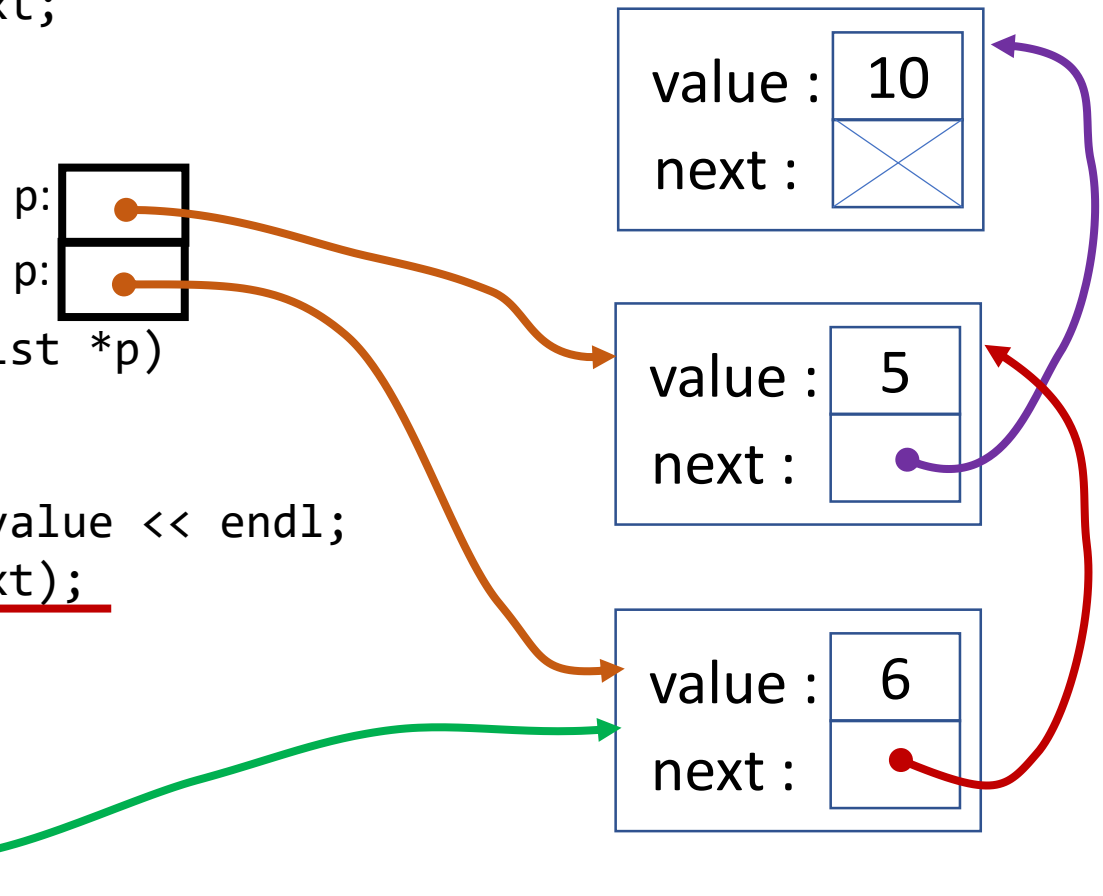
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
5
10
```

Printing: forwards

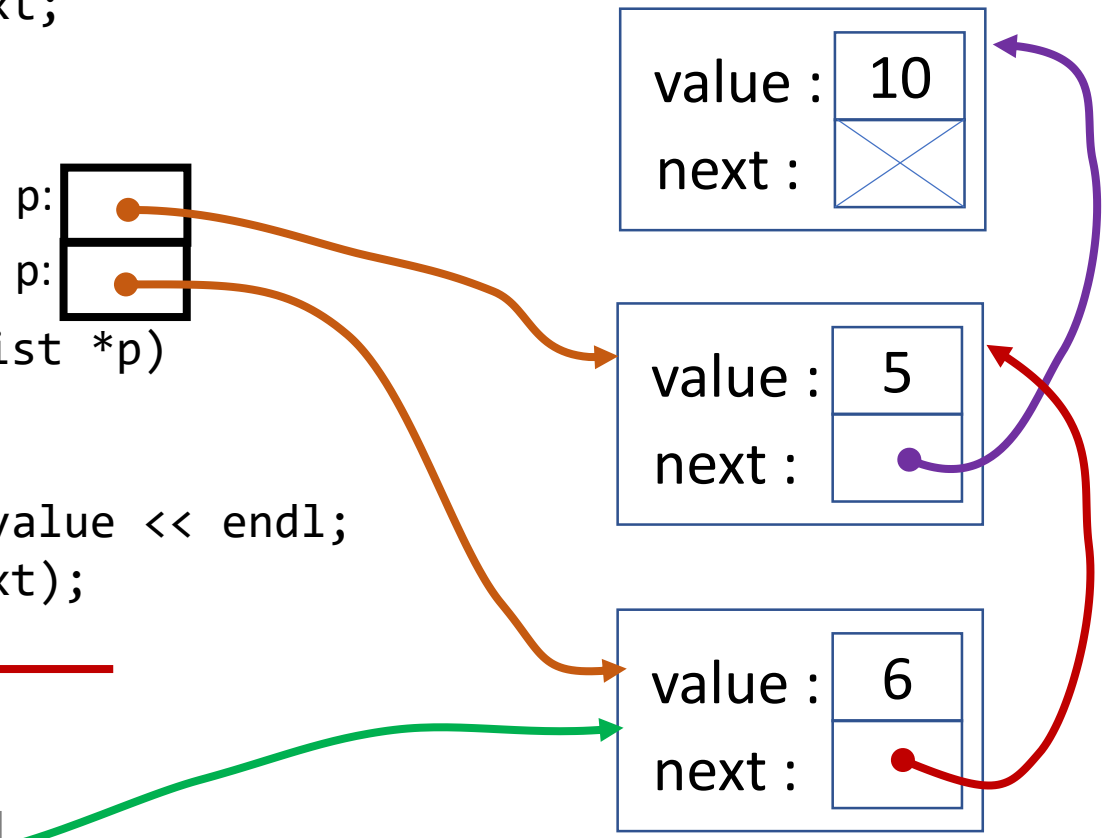
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
6
5
10
```

Printing: forwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

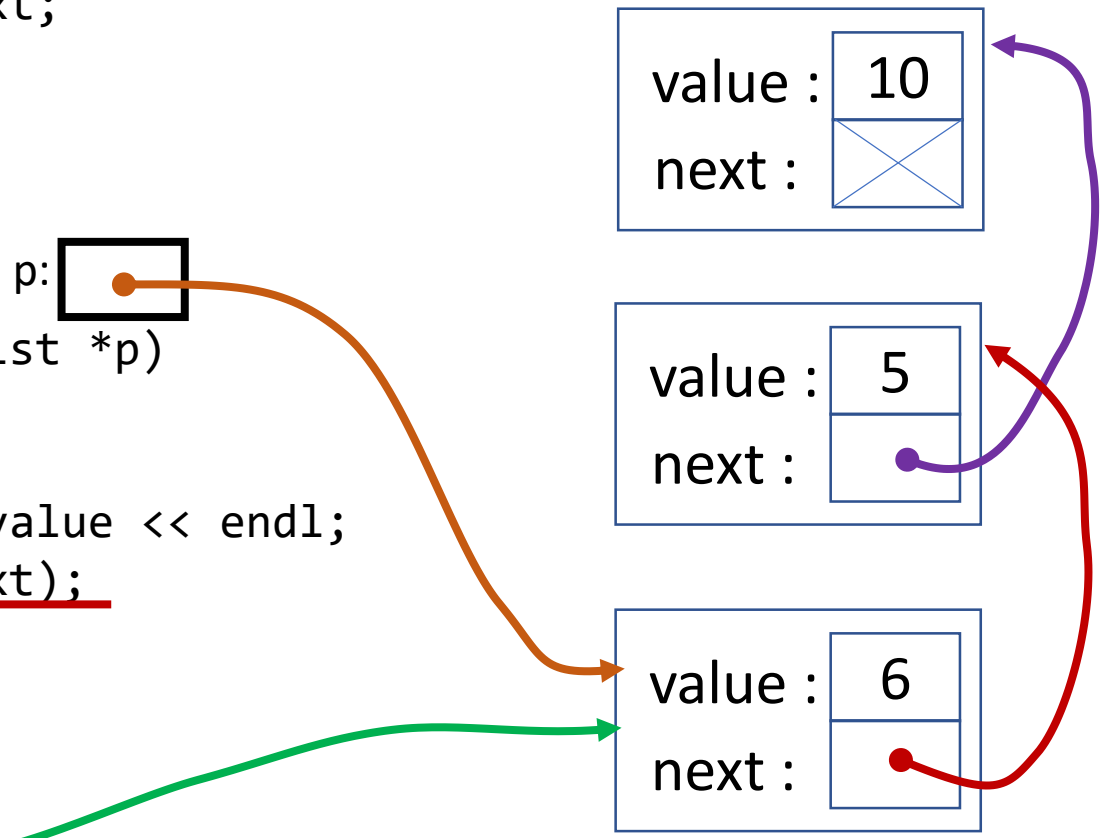



```
$ ./print-list
6
5
10
```

Printing: forwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

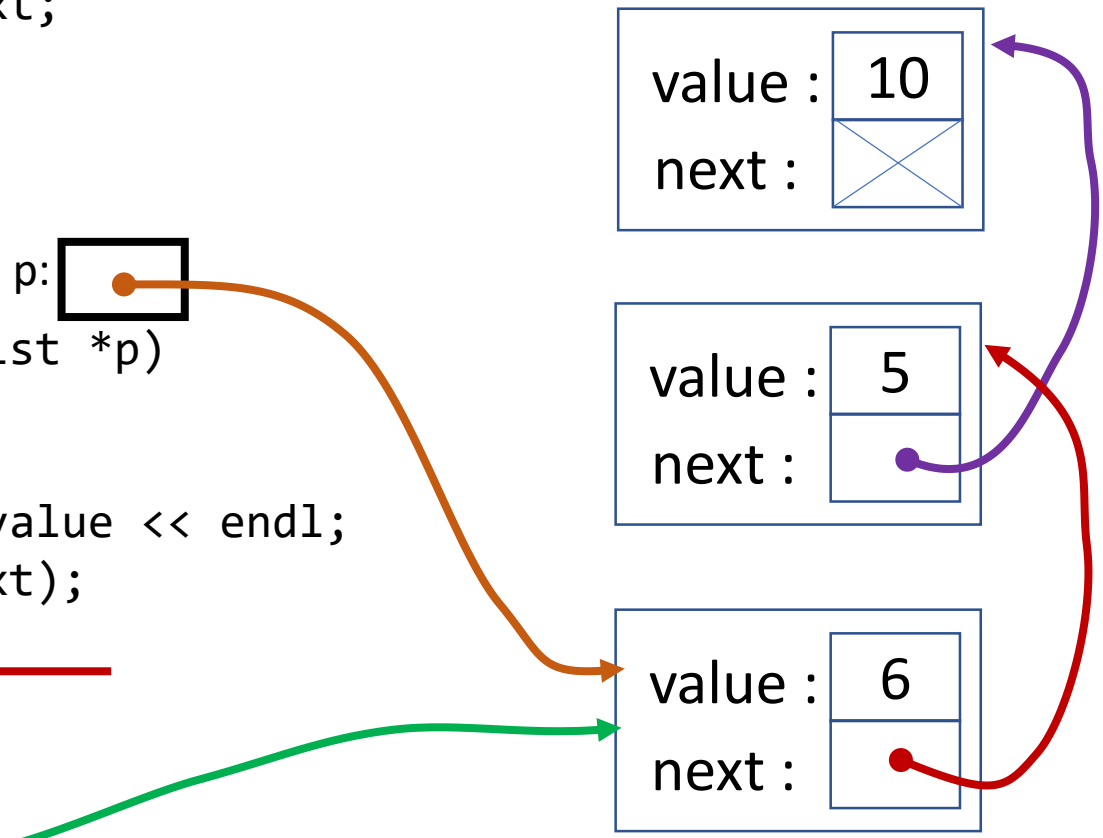



```
$ ./print-list
6
5
10
```

Printing: forwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

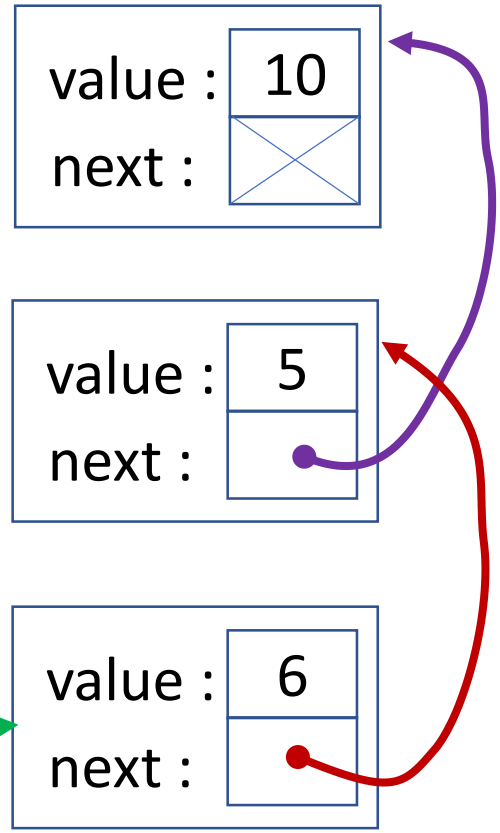



```
$ ./print-list
6
5
10
```

Printing: forwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
}
```




```
$ ./print-list
6
5
10
```

Printing: forwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        cout << p->value << endl;
        print(p->next);
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

Printing: backwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

Printing: backwards

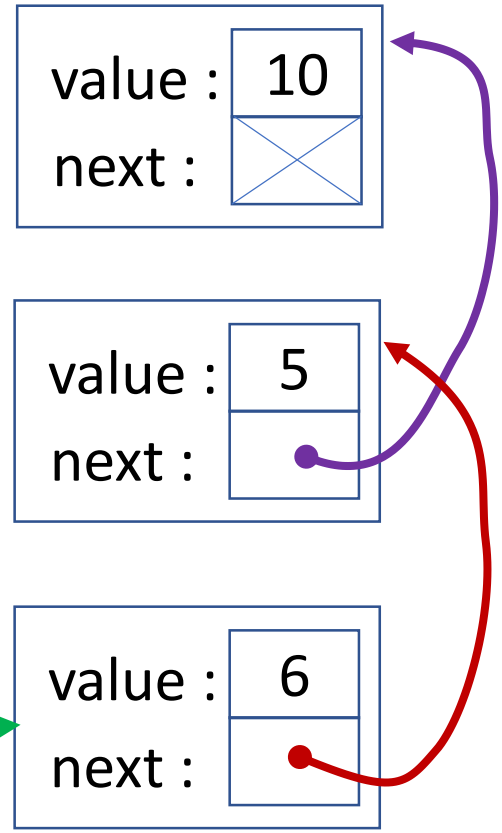
```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
         print(p->next);
         cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

Printing: backwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

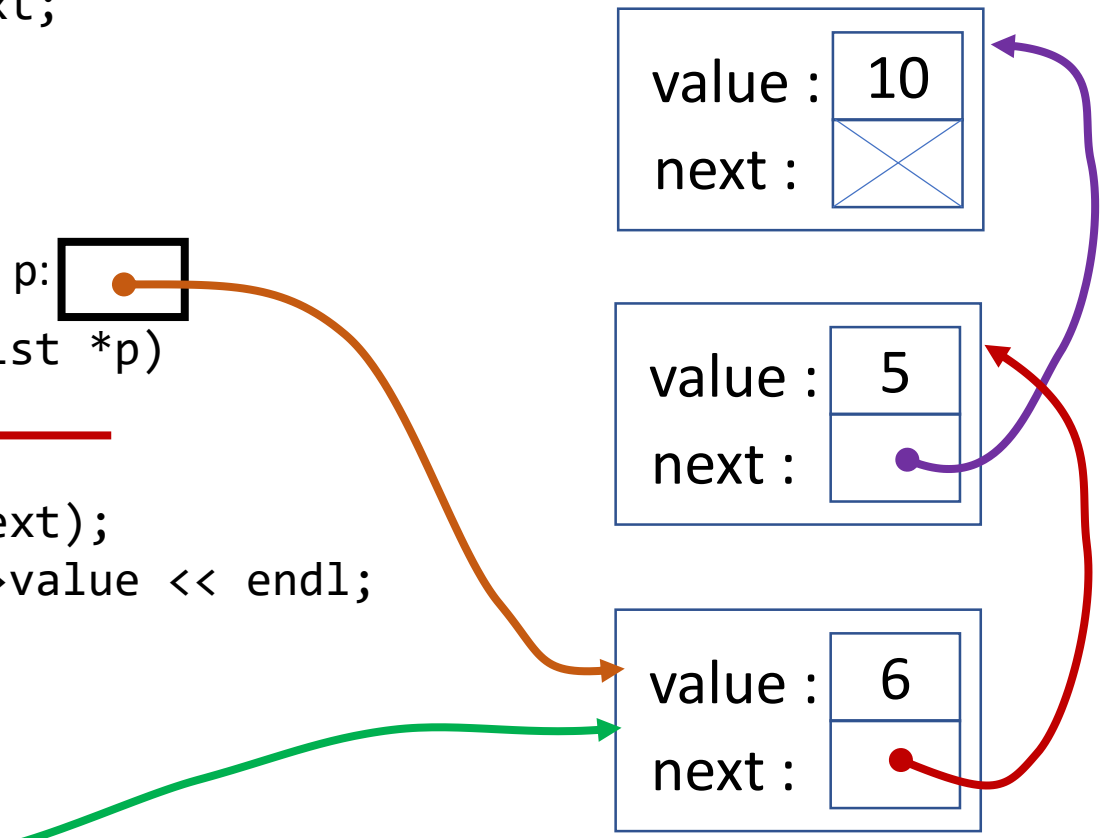



```
$ ./print-list
```

Printing: backwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

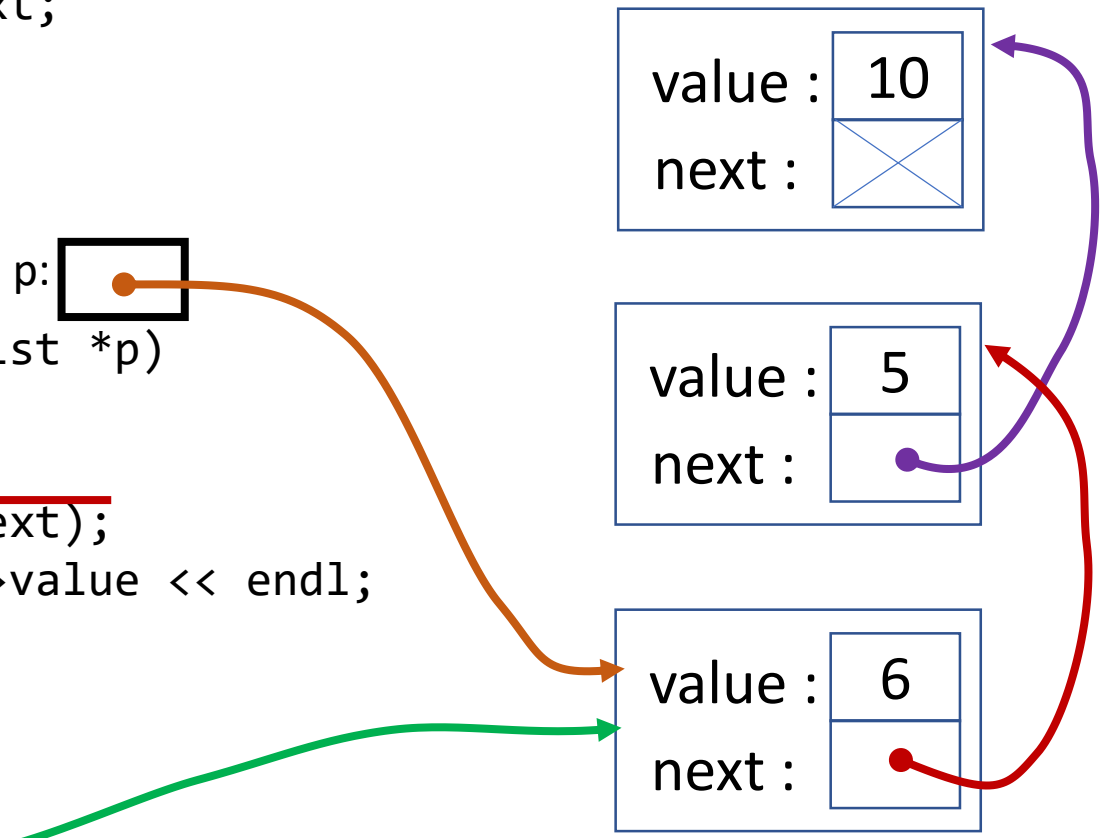
```
void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Printing: backwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

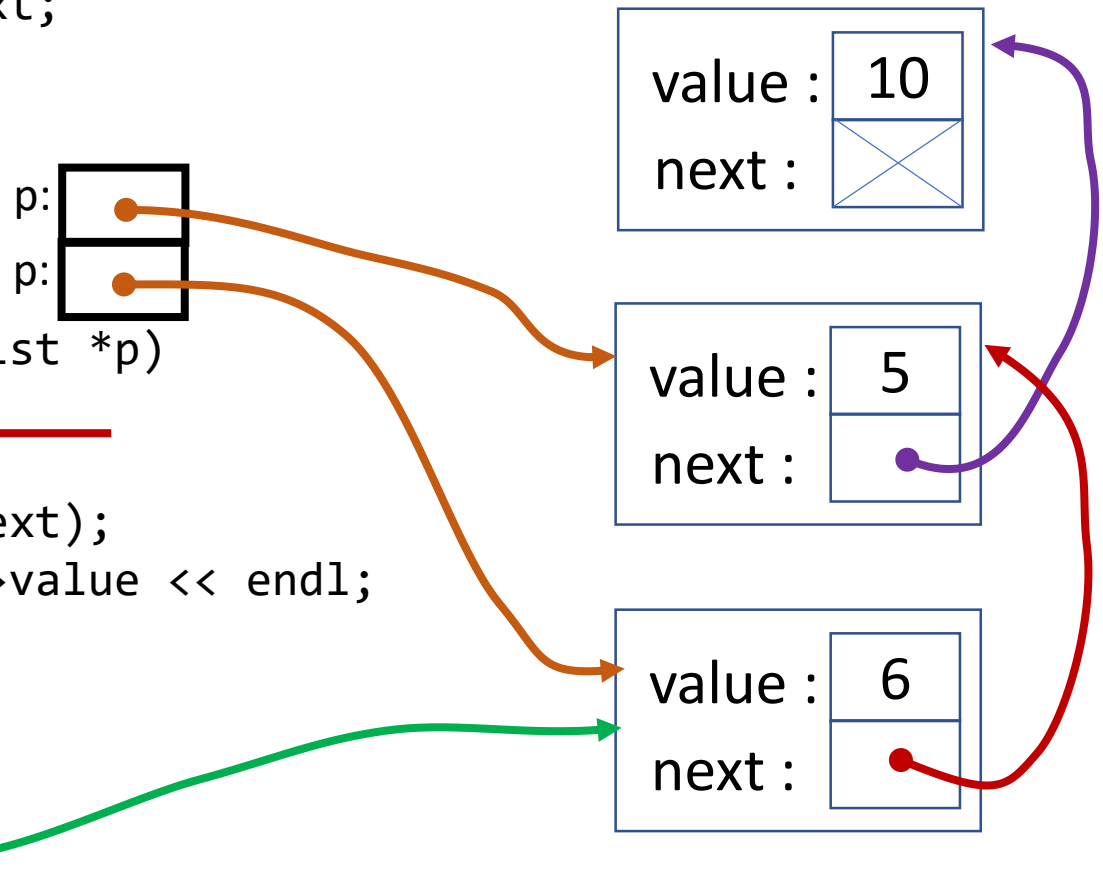



```
void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

```
$ ./print-list
```

Printing: backwards

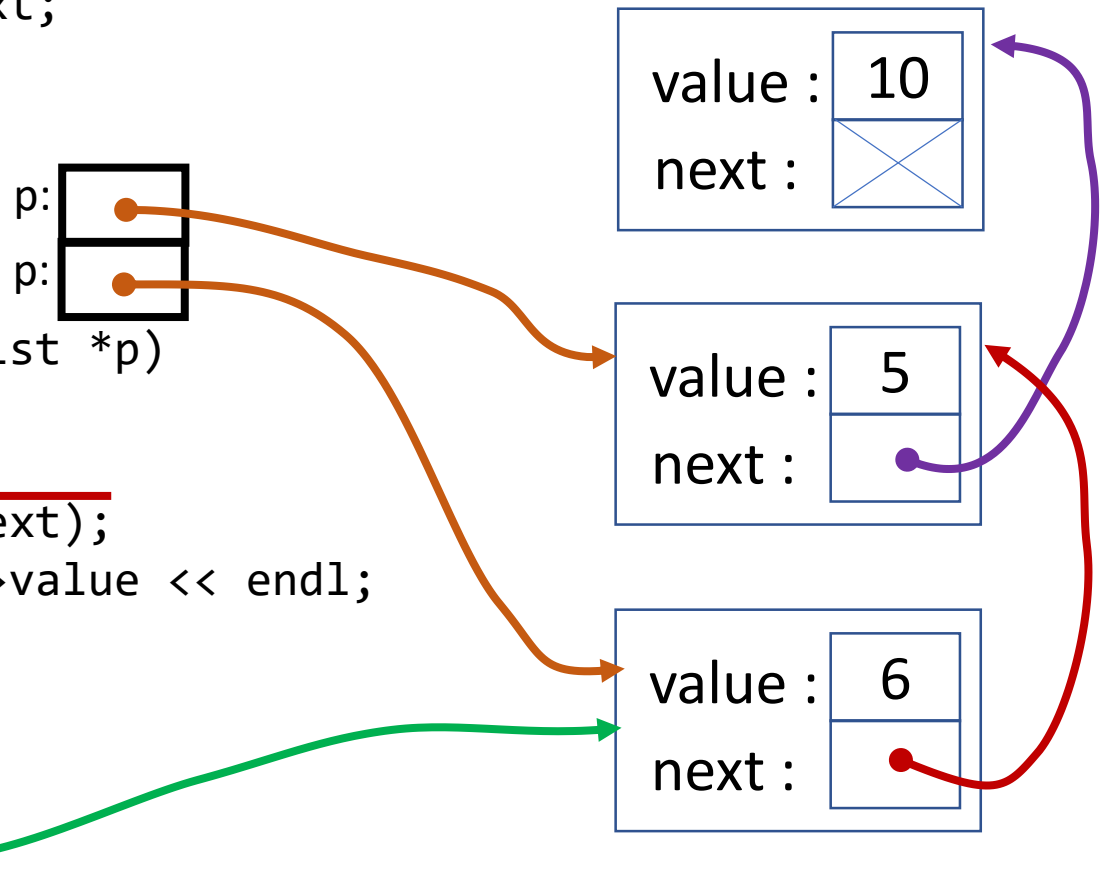
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Printing: backwards

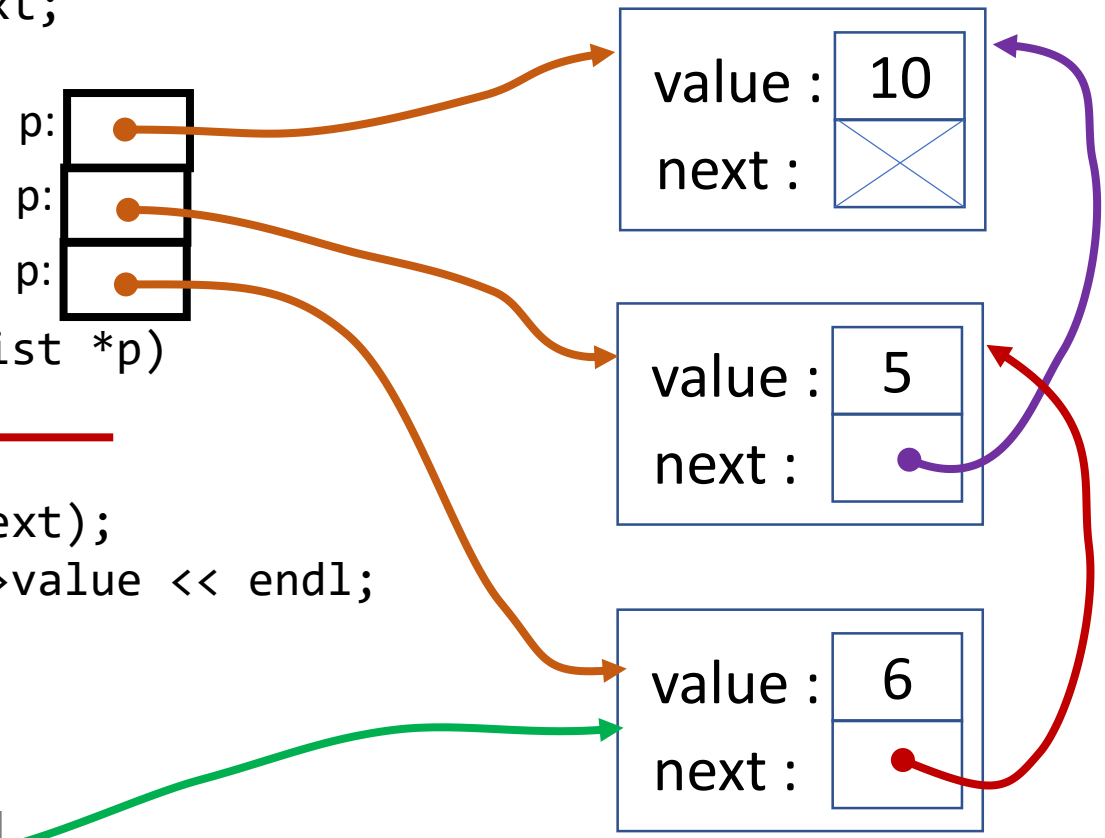
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Printing: backwards

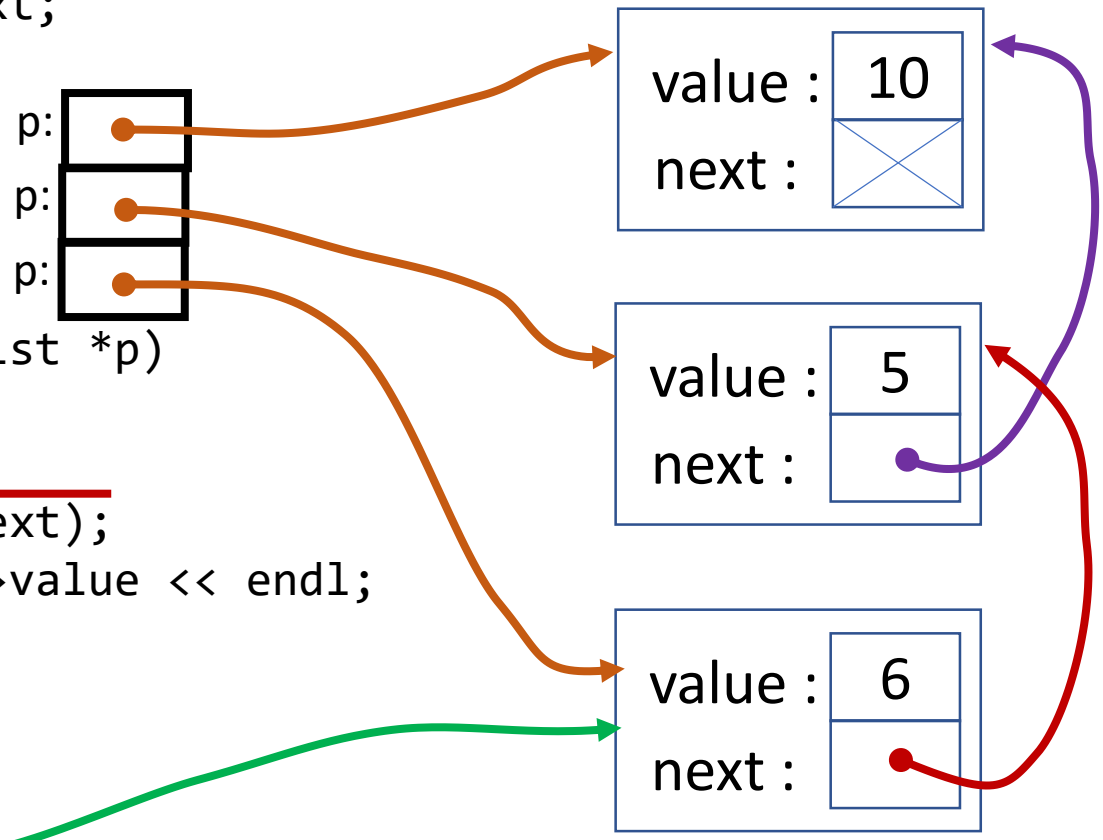
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Printing: backwards

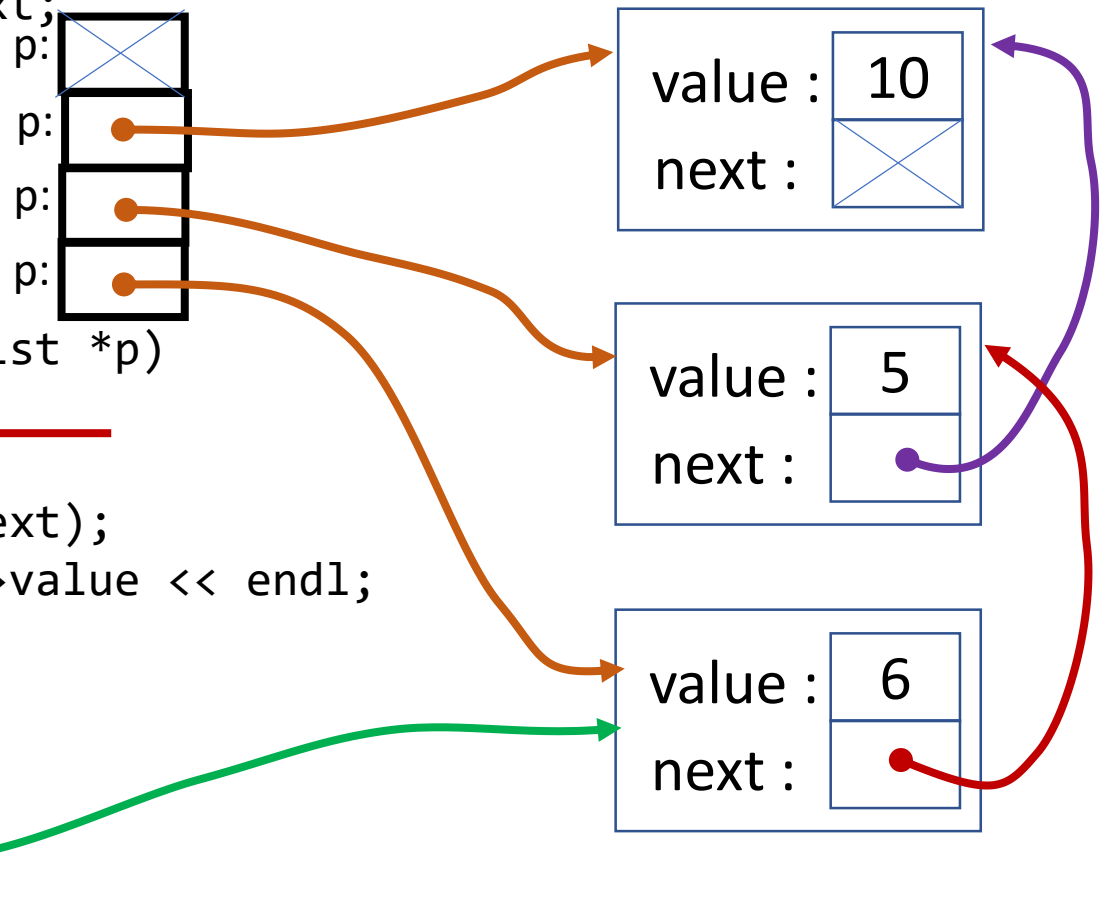
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Printing: backwards

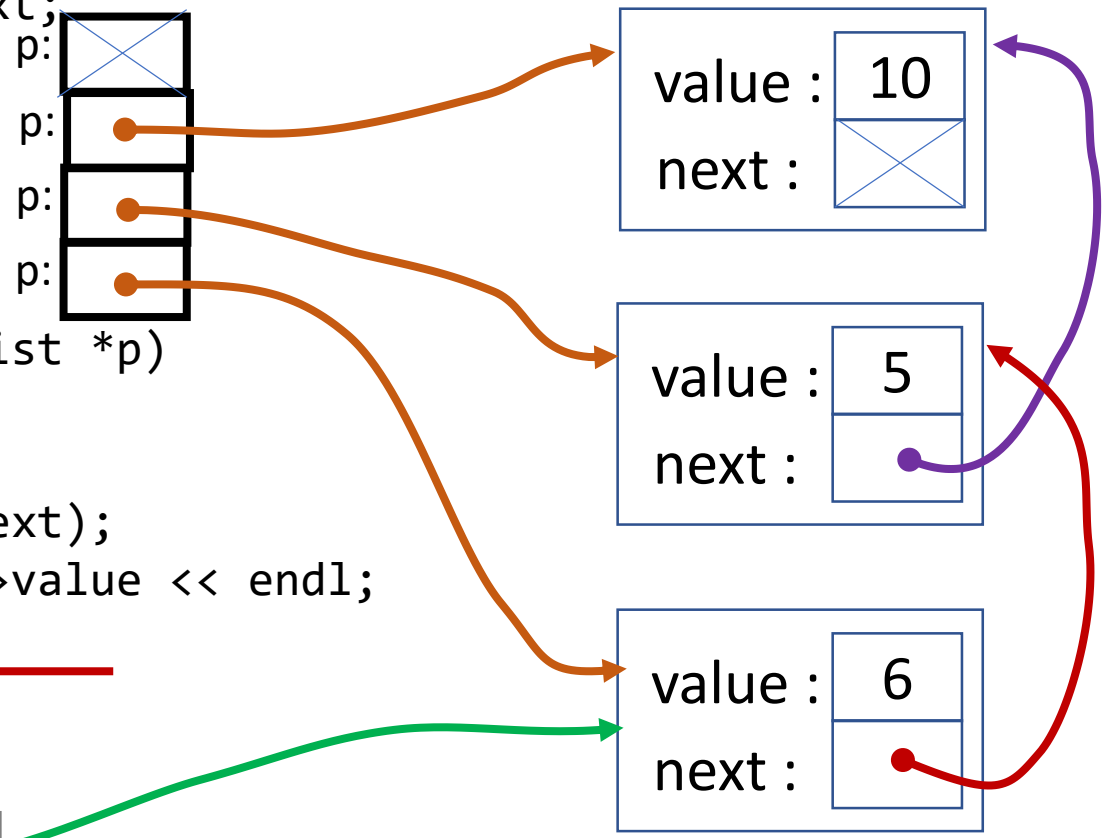
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Printing: backwards

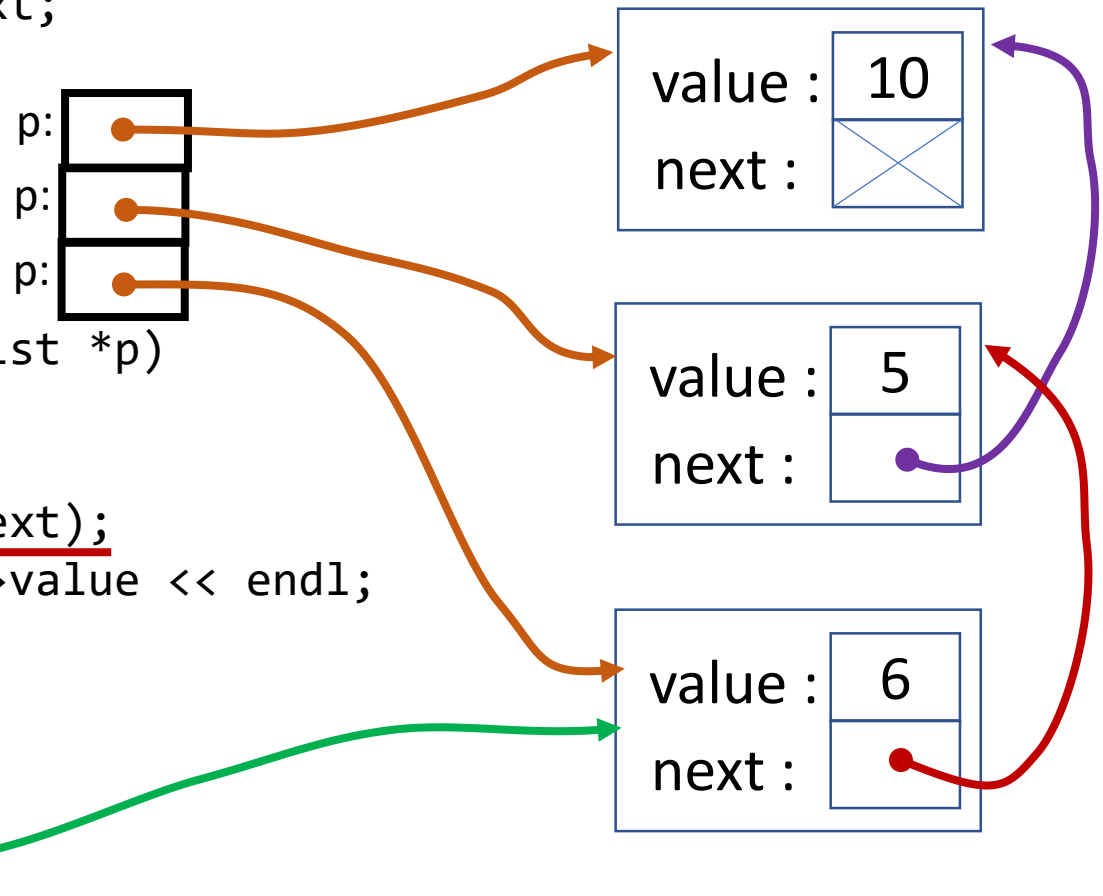
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Printing: backwards

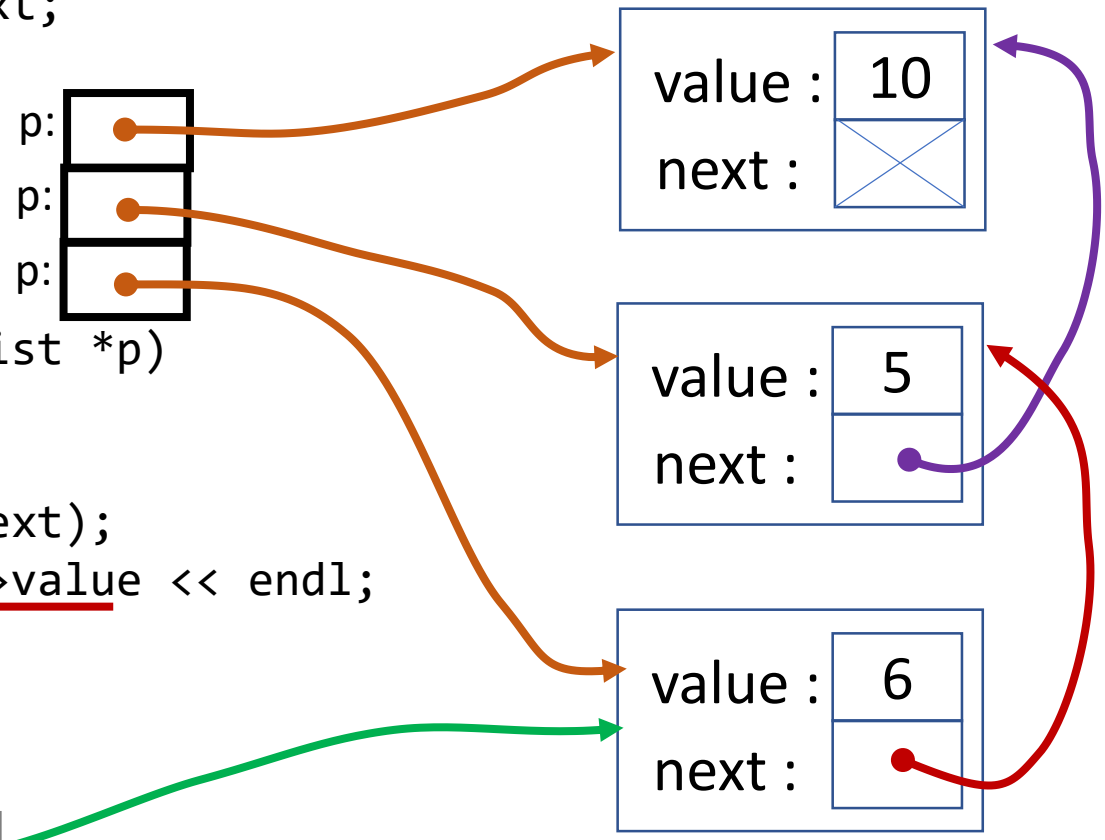
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

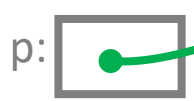



```
$ ./print-list
```

Printing: backwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

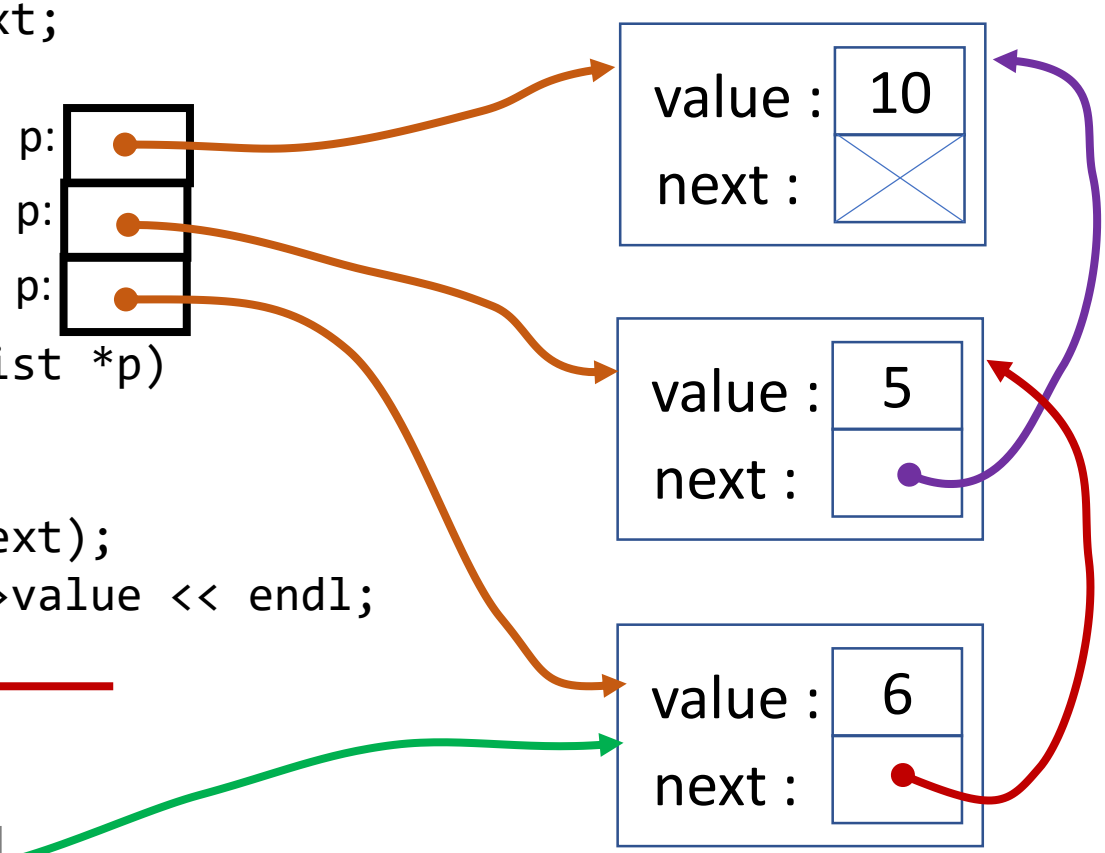





```
$ ./print-list
10
```

Printing: backwards

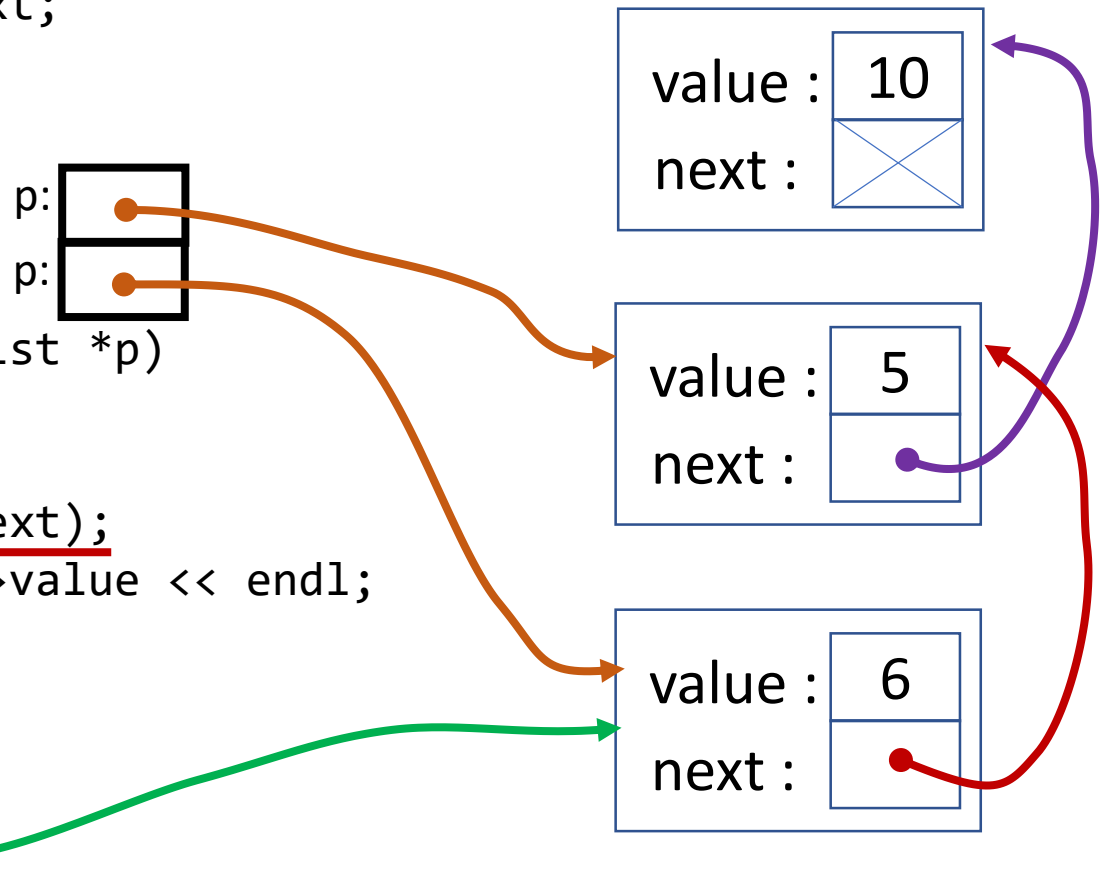
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
10
```

Printing: backwards

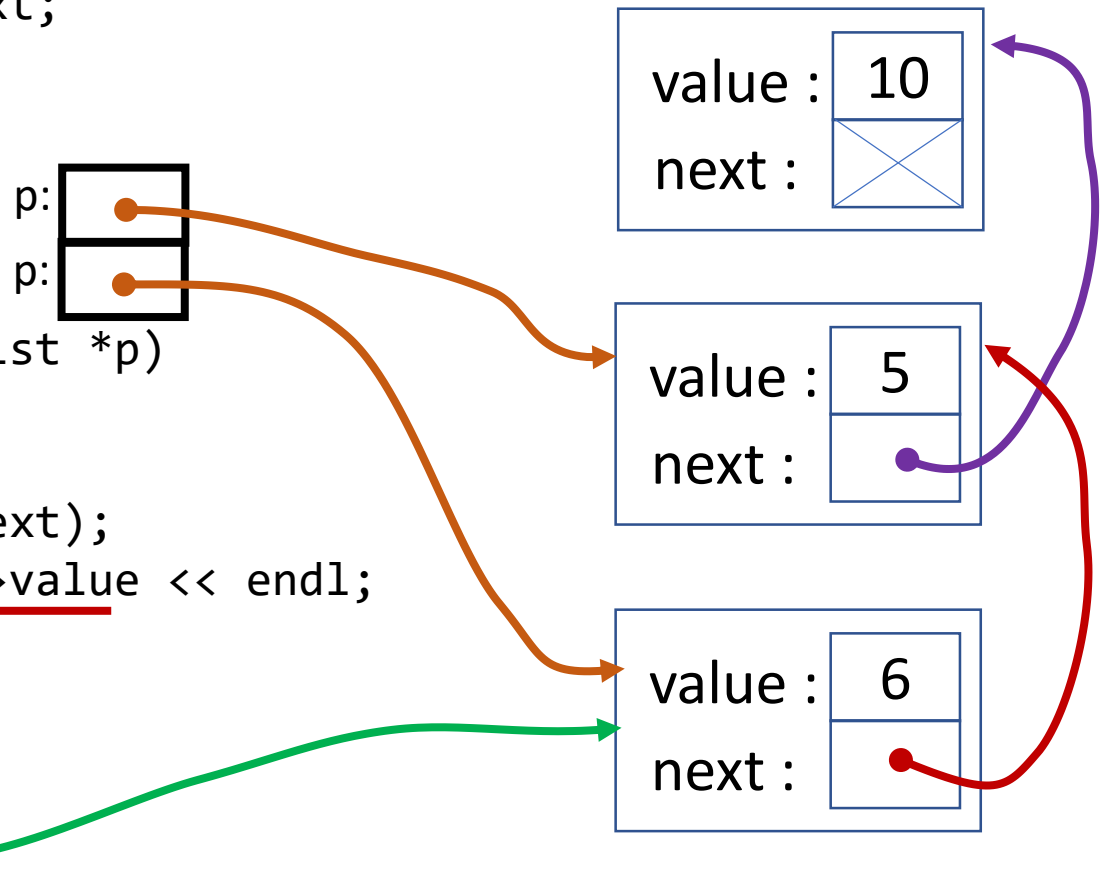
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
10
```

Printing: backwards

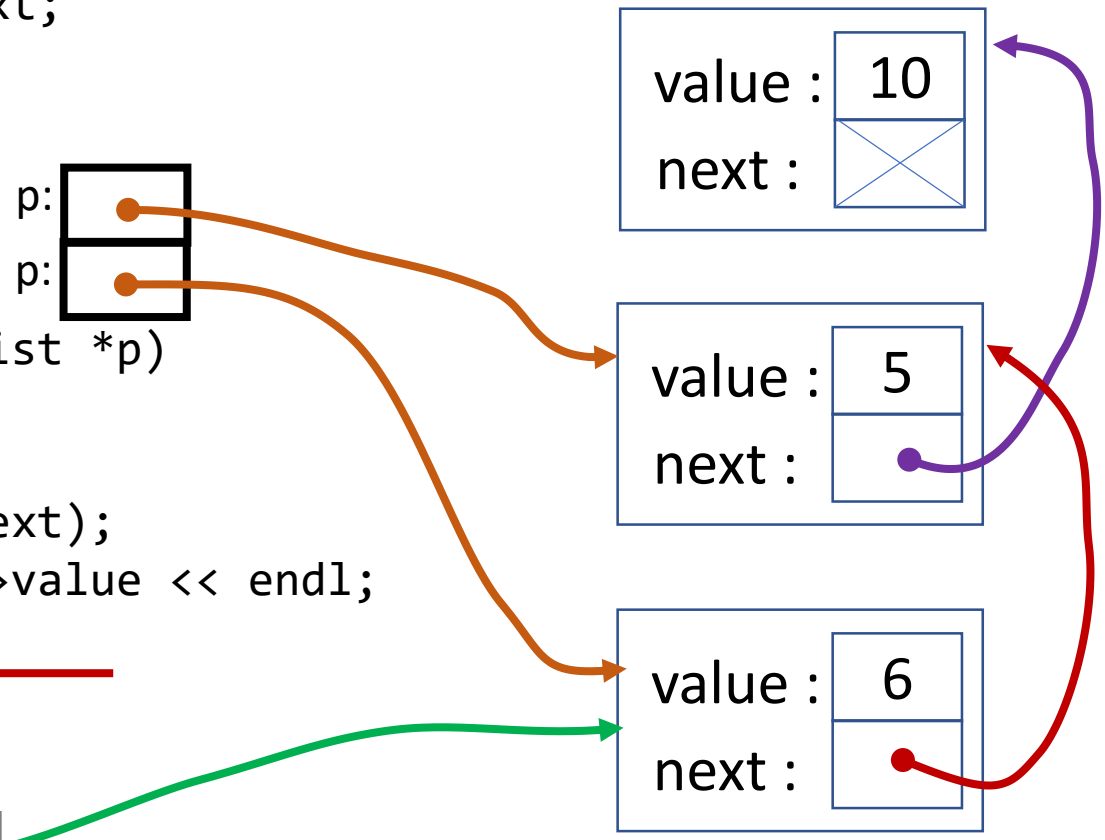
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
10
5
```

Printing: backwards

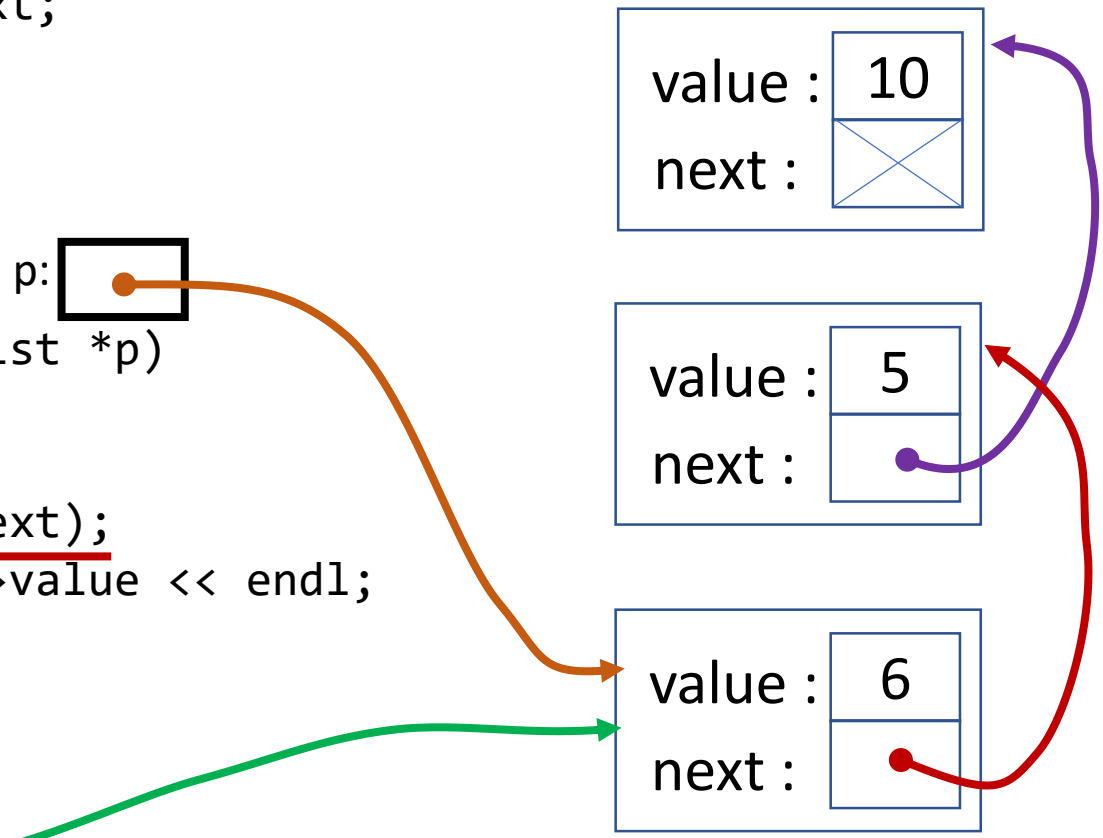
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
10
5
```

Printing: backwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```



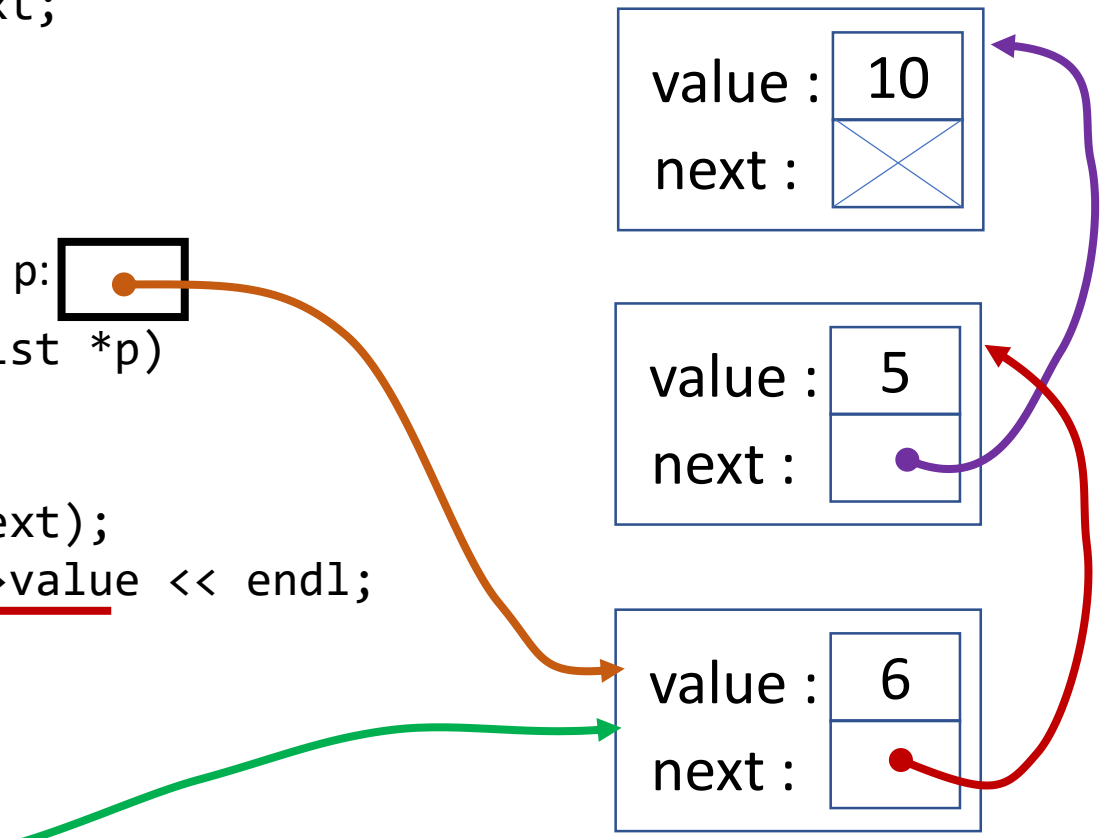

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

```
$ ./print-list
10
5
```

Printing: backwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

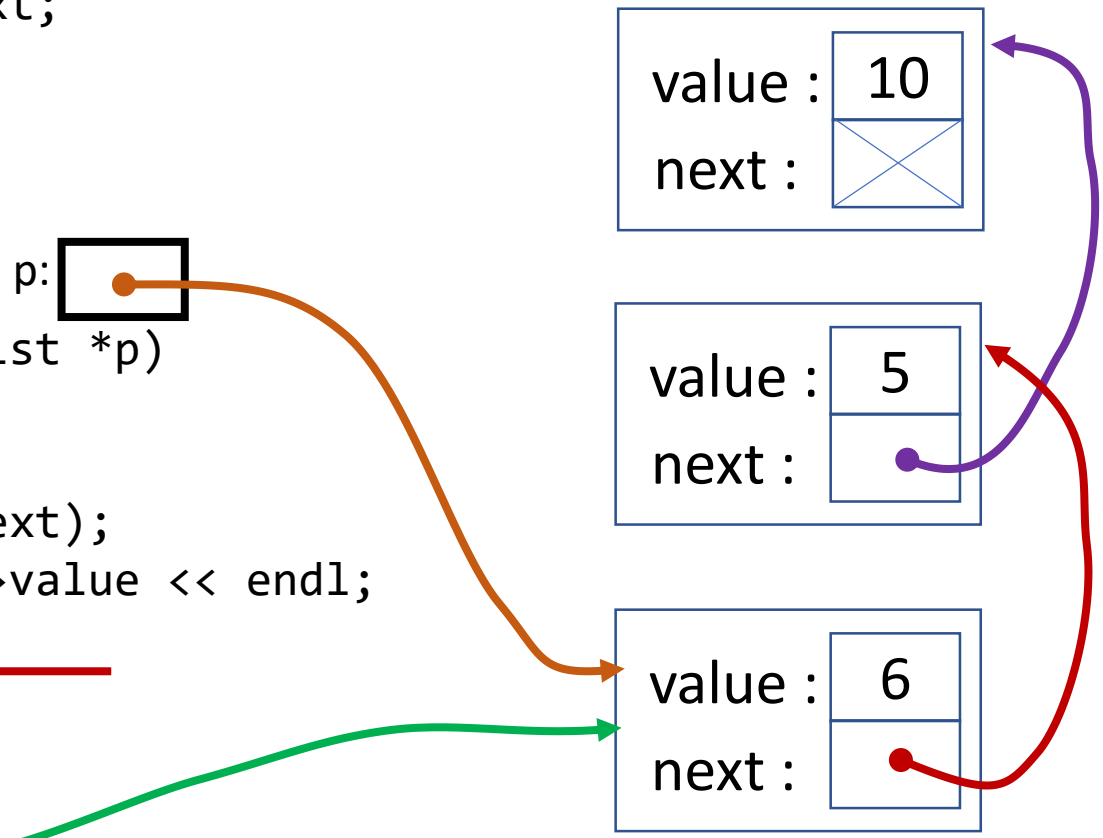



```
$ ./print-list
10
5
6
```

Printing: backwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

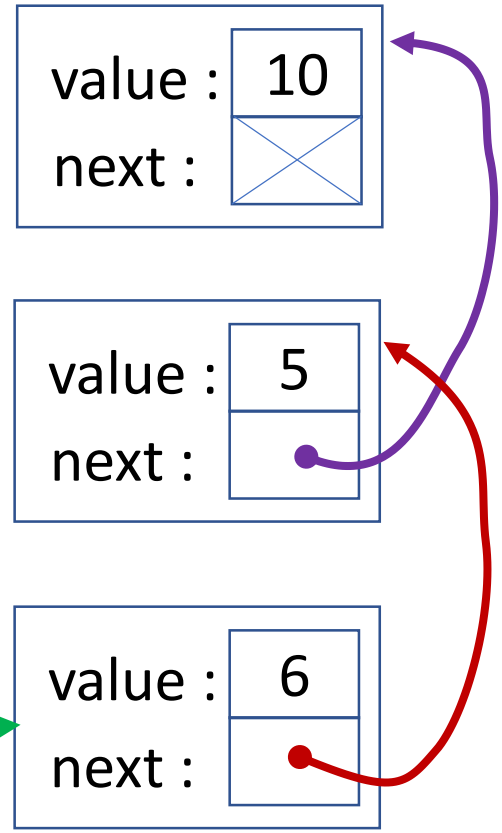



```
$ ./print-list
10
5
6
```

Printing: backwards

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
        cout << p->value << endl;
    }
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
}
```




```
$ ./print-list
10
5
6
```

Getting the base-case right

- In a linked data-structure `nullptr` happens
  - We use it to represent an empty list
  - It naturally becomes the recursive base case
- Make sure you get the base-case right
  - Is it legal to call your function of `nullptr` ?
  - Or: is it legal to call your function on an empty list?

- Check if the list is `nullptr` *before* using it

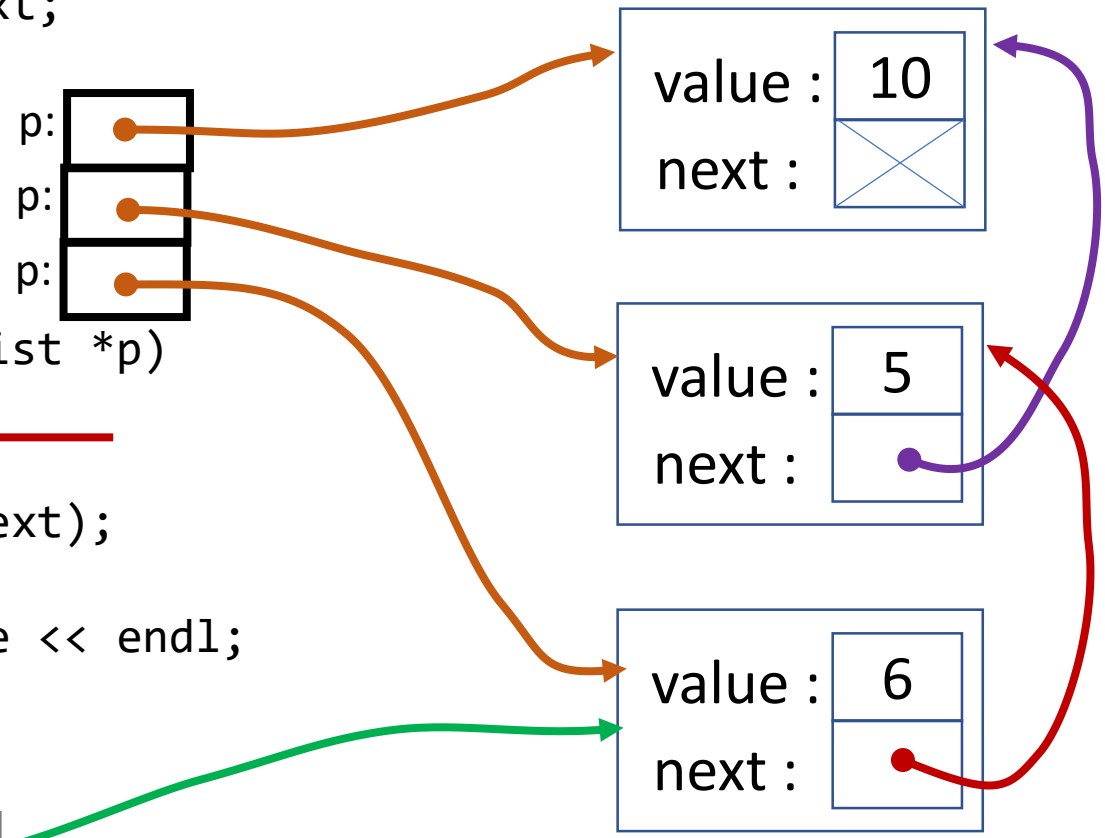
Printing: *danger!*

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
         print(p->next);
    }
    cout << p->value << endl;
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

Printing: *danger!*

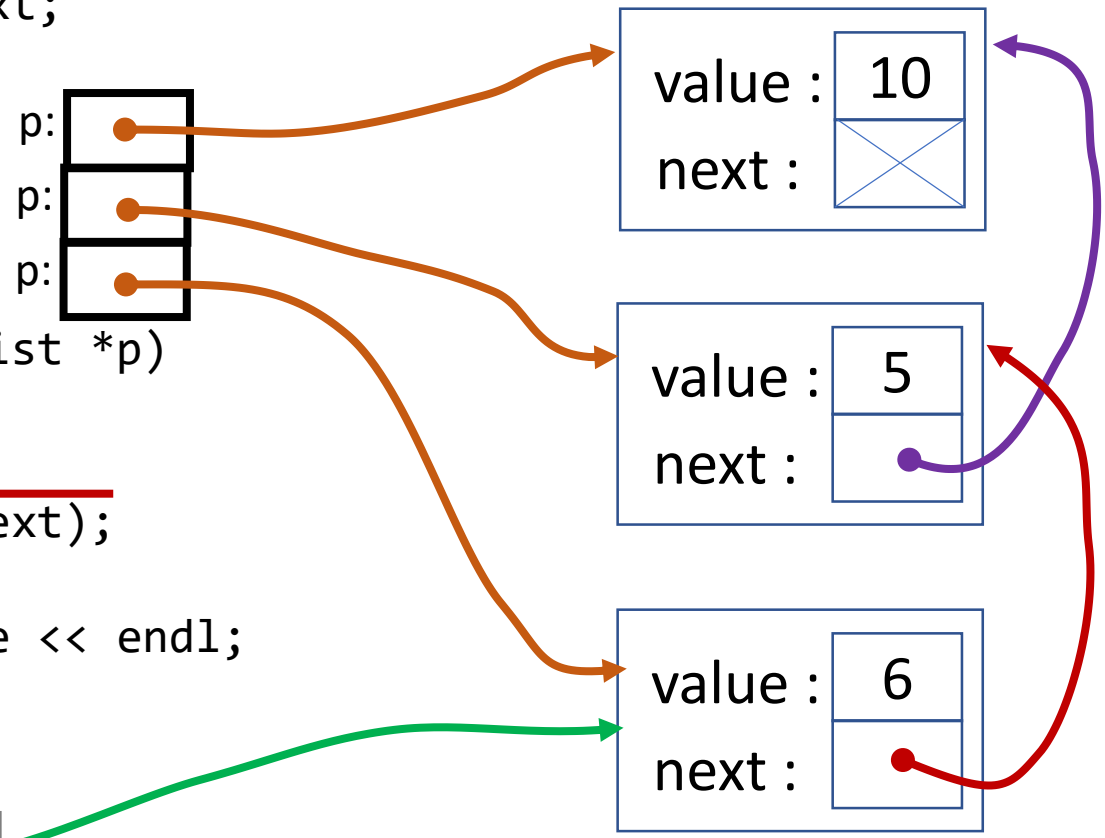
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
    }
    cout << p->value << endl;
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Printing: *danger!*

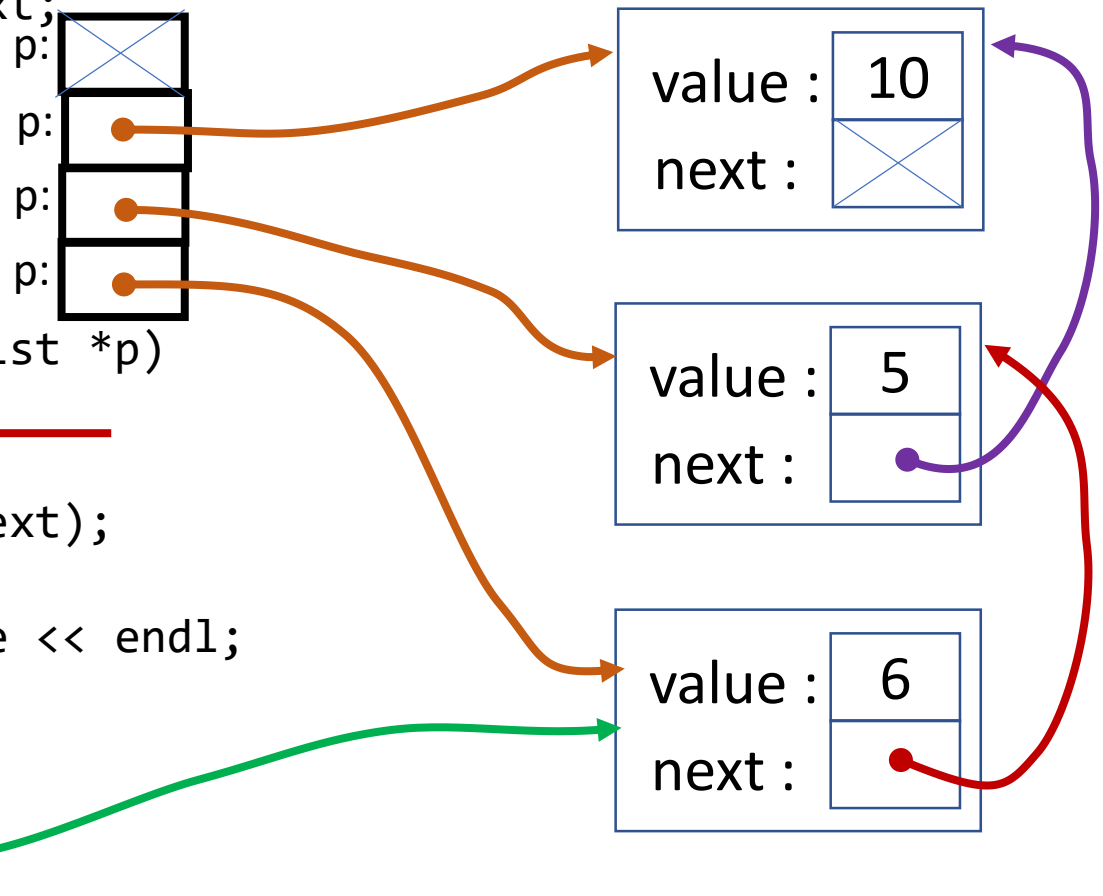
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
    }
    cout << p->value << endl;
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Printing: *danger!*

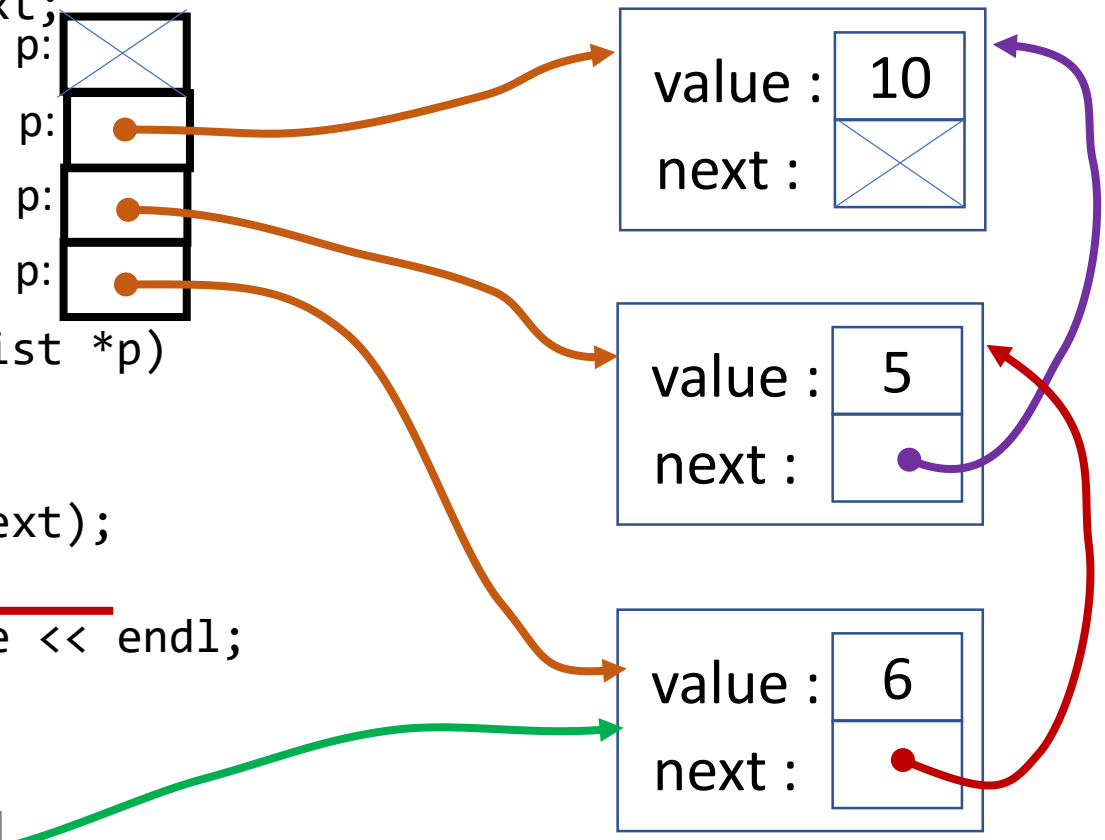
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
    }
    cout << p->value << endl;
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Printing: *danger!*

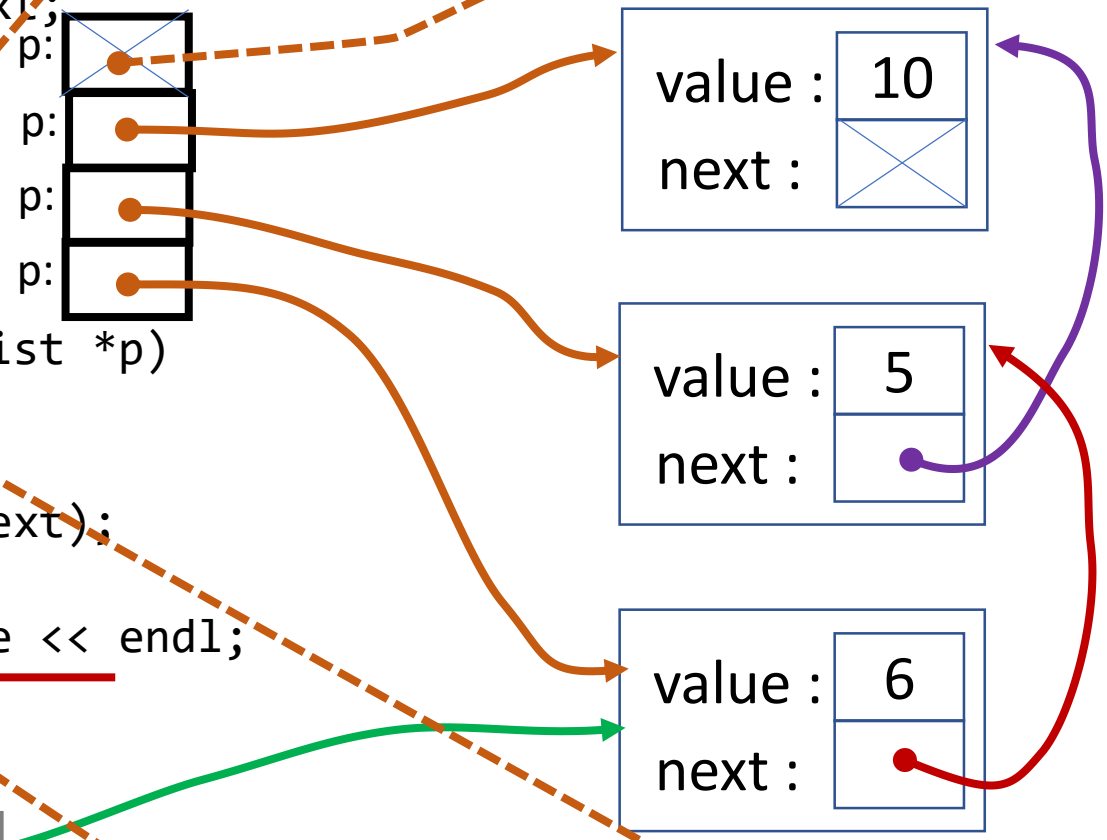
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
    }
    cout << p->value << endl;
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

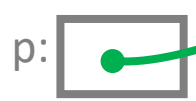



```
$ ./print-list
```

Printing: *danger!*

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
    }
    cout << p->value << endl;
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
}
```






```
$ ./print-list
```

Defending against errors

It is a good idea to program defensively

1. Try to anticipate things that could go wrong
2. Actively guard against those situations in the code

The assert function is an easy way of doing this

```
#include <cassert>
```

```
assert( condition );
```

Defending against errors

It is a good idea to program defensively

1. Try to anticipate things that could go wrong
2. Actively guard against those situations in the code

The assert function is an easy way of doing this

```
#include <cassert>

int f(int *p)
{
    assert( p!=nullptr  );
    // Do something with p
}
```

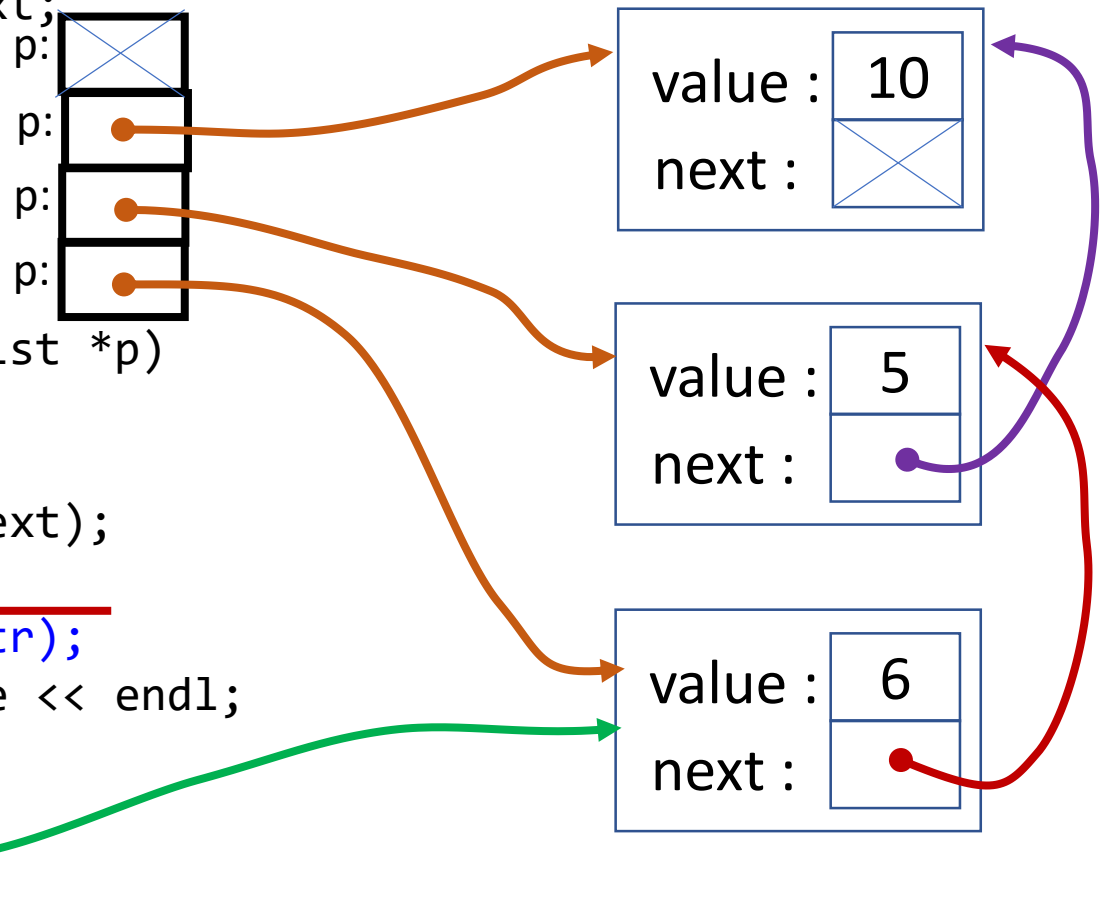
Defensive code

```
struct my_int_list
{
    int value;
    my_int_list *next;
};
```

```
void print(my_int_list *p)
{
    if(p!=nullptr){
         print(p->next);
    }
    assert(p!=nullptr);
    cout << p->value << endl;
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```

Defensive code

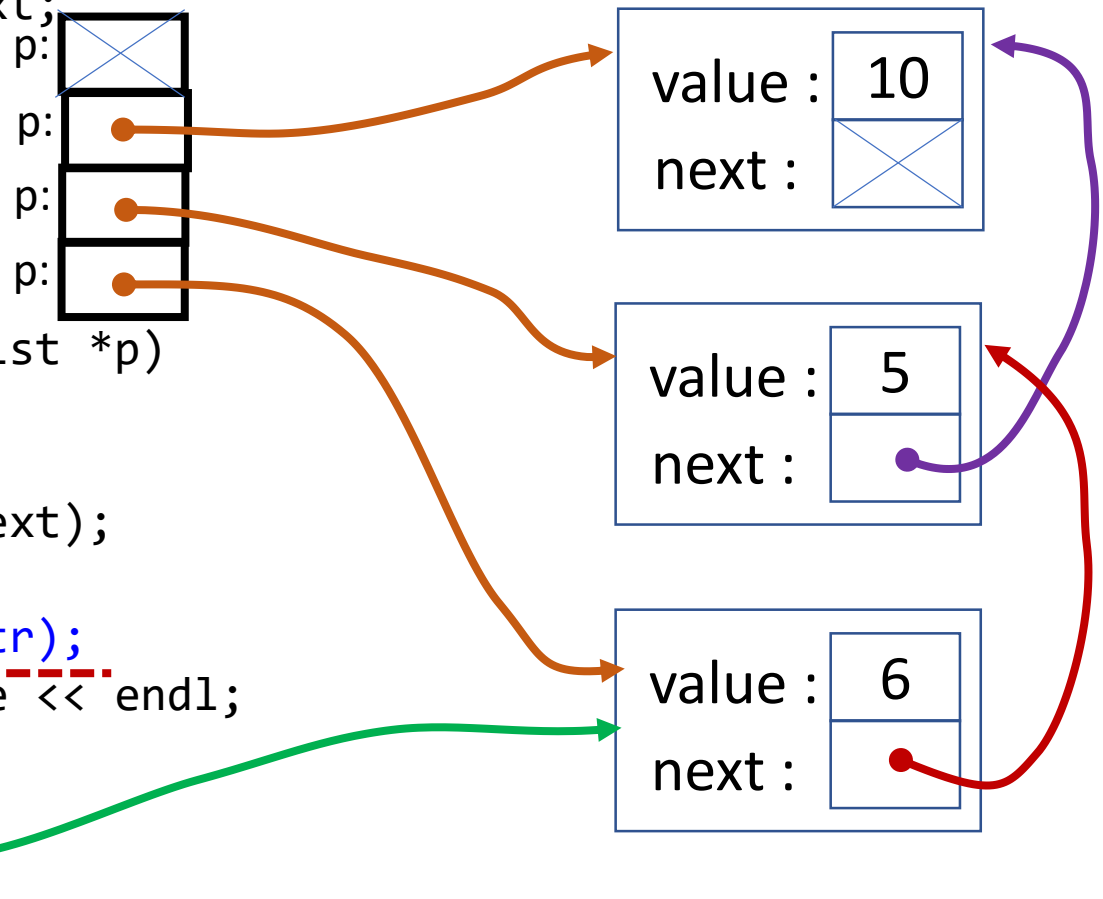
```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
    }
    assert(p!=nullptr);
    cout << p->value << endl;
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
```

Defensive code

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void print(my_int_list *p)
{
    if(p!=nullptr){
        print(p->next);
    }
    assert(p!=nullptr);
    cout << p->value << endl;
}

int main()
{
    int_my_list *p = nullptr;
    ... // Our previous build
    print(p);
)
```




```
$ ./print-list
assertion failed: p!=nullptr
$
```

Advantages of assert

- Assert terminates the program immediately
  - Fails at exactly the location the error happened
  - Stops anything else bad happening
- Assert also helps with debugging
  - Prints the expression that failed
  - *Prints the exact line of code that failed*
- A failed assertion is still a crashed program
  - But... at least you can find out where it happened and why

More operations : list length

```
// Return the number of values in the list p
int length(my_int_list *p);
```

More operations : list length

```
// Return the number of values in the list p
int length(my_int_list *p)
{
    if(p == nullptr){
        // An empty list has length zero
        return 0;
    }else{
        // Find the length of the rest of the list
        int len_next = length(p->next);
        // ... and this node makes it one longer
        return 1 + len_next;
    }
}
```

More operations : list length

```
// Return the number of values in the list p
int length(my_int_list *p)
{
    if(p == nullptr){
        return 0;
    }else{
        return 1 + length(p->next);
    }
}
```

More operations : list length

```
// Return the number of values in the list p
int length(my_int_list *p)
{
    int count=0;
    while( p!=nullptr ){
        count++;
        p = p->next;
    }
    return count;
}
```

More operations : finding a value

```
// Given a list p and value x either:
// - Return the first node containing x
// - Return nullptr if x not found
my_int_list *find(my_int_list *p, int x);
```

More operations : finding a value

```
// Given a list p and value x either:
// - Return the first node containing x
// - Return nullptr if x not found
my_int_list *find(my_int_list *p, int x)
{
    if(p->value==x){
        return p;
    }else if(p == nullptr){
        return p;
    }else{
        return find(p->next, x);
    }
}
```

What happens for the empty list?

More operations : finding a value

```
// Given a list p and value x either:
// - Return the first node containing x
// - Return nullptr if x not found
my_int_list *find(my_int_list *p, int x)
{
    if(p == nullptr){
        return p;
    }else if(p->value==x){
        return p;
    }else{
        return find(p->next, x);
    }
}
```

More operations : finding a value

```
// Given a list p and value x either:
// - Return the first node containing x
// - Return nullptr if x not found
my_int_list *find(my_int_list *p, int x)
{
    if(p == nullptr){
        return p;
    }
    // We are now guaranteed p is non-null
    assert( p!=nullptr );
    if(p->value==x){
        return p;
    }else{
        return find(p->next, x);
    }
}
```

More operations : deleting a value

```
// Remove the first value matching x
// This function may modify the list p
my_int_list *remove(my_int_list *p, int x)
{
    if(p == nullptr){
        return p;
    }else if( p->value == x){
        my_int_list *res = p->next;
        delete[] p;
        return res;
    }else{
         p->next = remove( p->next , x);
         return p;
    }
}
```

Re-using code

Functions + types = API

```
// Represents a list of integers
struct my_int_list;

// Create an empty list
my_int_list *create_list();

// Add x to the front of list p
my_int_list *push_front(my_int_list *p, int x);

// Remove value from the front of list p
my_int_list *pop_front(my_int_list *p);

// Remove the first value of x from list p
my_int_list *remove(my_int_list *p, int x);
```

How do we re-use or share code?

- We've create a nice piece of functionality
  - A carefully designed API
  - Documented functionality and semantics
  - An implementation of the functions
  - Large test-suite to show that it is correct

- We now want to re-use and share this code
  1. *Use it now*: it's one component of a larger program
  2. *Use it later*: you'll need the functionality in future work
  3. *Give to others*: they might find the functionality useful

```cpp
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);


void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}

void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

test_my_int_list.cpp

```cpp
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);

void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}

void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

test_my_int_list.cpp


```cpp
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);
```


Declarations: types and functions

- Describe the *public* API
- The part users care about

```cpp
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);

void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}

void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

test_my_int_list.cpp

```cpp
void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}
```

Implementation: the actual "code"

- Describes the internal operation
- The part users want to ***use***
- They *don't* care how it works

```cpp
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);


void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}

void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

test_my_int_list.cpp


```cpp
void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```


Testing: checks the implementation

- Provides evidence that it works
- Needed during implementation
- Supports future development
- The tests are very valuable

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);
```

```
void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}
```

```
void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

test_my_int_list.cpp

```cpp
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);

void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}

void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

test_my_int_list.cpp

```cpp
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);

void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}

void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

my_int_list.hpp

Copy to “header” file

Extension `.hpp` means
C++ header file by convention

```cpp
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);


void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}

void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

test_my_int_list.cpp

```cpp
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);

void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}

void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

my_int_list.hpp

Split the sections up

- Header file
  - Public API
  - Implementation
- Source file
  - Testing code

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);
```

```
void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}
```

my_int_list.hpp

```
void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

test_my_int_list.cpp

Use #include to bring back code

```
#include "my_int_list.hpp"
```

```
#include "my_int_list.hpp"

void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

test_my_int_list.cpp

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);

void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}
```

my_int_list.hpp

Use #include in new programs

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
my_int_list *push_front(my_int_list *p, int x);
my_int_list *pop_front(my_int_list *p);

void destroy(my_int_list *p)
{
    if(p!=nullptr){
        destroy(p->next);
        delete[] p;
    }
}

int length(my_int_list *p)
{
    if( p==nullptr ){
        return 0;
    }else{
        return 1 + length(p);
    }
}

my_int_list *push_front(my_int_list *p, int x)
{
    my_int_list *res=new my_int_list[1];
    res->value=x;
    res->next=p;
    return res;
}

my_int_list *pop_front(my_int_list *p)
{
    assert( p!=nullptr );
    my_int_list *res=p->next;
    delete[] p;
    return res;
}
```

my_int_list.hpp

```
#include "my_int_list.hpp"

void test_destroy()
{
    my_int_list *p=nullptr;
    destroy(p); // Check it doesn't crash
}

void test_push_front()
{
    my_int_list *p=nullptr;
    p=push_front(p, 0);
    assert( length(p) == 1 );
    destroy(p);
}

int main()
{
    test_destroy();
    test_push_front();
}
```

test_my_int_list.cpp

```
#include "my_int_list.hpp"

#include "my_int_list.hpp"

int main()
{
  int x;
  my_int_list *p=nullptr;
  while(cin >> x){
    p = push_front(p, x);
  }
}
```

my_great_program.cpp

Avoiding double inclusion

Headers can #include other headers

```
struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);
```

my_int_vec.hpp

```
#include "my_int_vec.hpp"

int read(my_int_vec *a, int index)
{
    // ...
}
```

my_int_vec_read.hpp

```
#include "my_int_vec.hpp"

my_int_vec *reverse(my_int_vec *p)
{
    // ...
}
```

my_int_vec_reverse.hpp

```
#include "my_int_vec_reverse.hpp"
#include "my_int_vec_read.hpp"
```

my_awesome_program.cpp

Avoiding double inclusion

Headers can #include other headers

You can ***declare*** a struct or function many times

- As long as the declarations match

You cannot ***define*** a struct of function twice

We can use “include guards” to fix this

This is a hack from the C language, sadly still here

```cpp
#ifndef my_int_vec_reverse_hpp
#define my_int_vec_reverse_hpp

struct my_int_list
{
    int value;
    my_int_list *next;
};

void destroy(my_int_list *p);
int length(my_int_list *p);

#endif
```

my_int_vec.hpp

```cpp
#ifndef my_int_vec_read_hpp
#define my_int_vec_read_hpp

#include "my_int_vec.hpp"

int read(my_int_vec *a, int index)
{
    // ...
}

#endif
```

my_int_vec_read.hpp

```cpp
#ifndef my_int_vec_reverse_hpp
#define my_int_vec_reverse_hpp

#include "my_int_vec.hpp"

my_int_vec *reverse(my_int_vec *p)
{
    // ...
}

#endif
```

my_int_vec_reverse.hpp

```cpp
#include "my_int_vec_reverse.hpp"
#include "my_int_vec_read.hpp"
```

my_awesome_program.cpp

Breaking up programs

- Splitting your programs up manages complexity
  - Define a clear API for a piece of functionality
  - Make it available in a re-useable and testable header
  - We've already benefitted from this via standard library

System headers

```
#include <string>
#include <cmath>
```

Looks for files in compiler defined locations

User headers

```
#include "my_int_list.hpp"
#include "find_roots.hpp"
```

Looks for files relative to the including source file

Managing change in your code

- Restructuring code makes it easier to maintain
  - Split large files up
  - Hide unnecessary detailed
  - Keep implementation and test separate

- It is also where things can go wrong
  - Accidently losing a function during copy and paste
  - Deleting the wrong file
  - Breaking a previous working program

New features

1. Decide on a feature to implement
2. Write the feature
3. Test the feature
4. Capture a snapshot of the working program
   - Files: create a zip
   - Git: create a `commit`
5. *Ideally*: test the snapshot somewhere else
   - Files: extract the zip somewhere new and test
   - Git: check out the commit in a different place and test

Fixing bugs

1. Find a bug in the program
2. Add a test-case for the bug
3. Fix the bug
4. Capture a snapshot of the working program
5. *Ideally*: test the snapshot somewhere else

Managing your portfolio

- We're mid-way between using files and commits
  - The value of source control is beginning to become clear
  - But: we can't fully switch to git yet
- *However*: you can still use the ideas
  - When you complete an exercise, add a basic test
  - e.g. create a script in each set
    - Does this file compile?
    - Does this script run?
- Your submission is like releasing software
  - Submit your final version to blackboard
  - *Then*: download onto a different machine and test it

Trees: intro

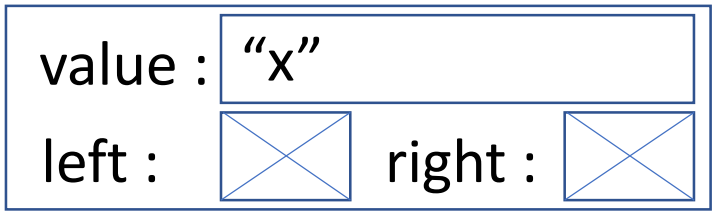
Vectors versus lists

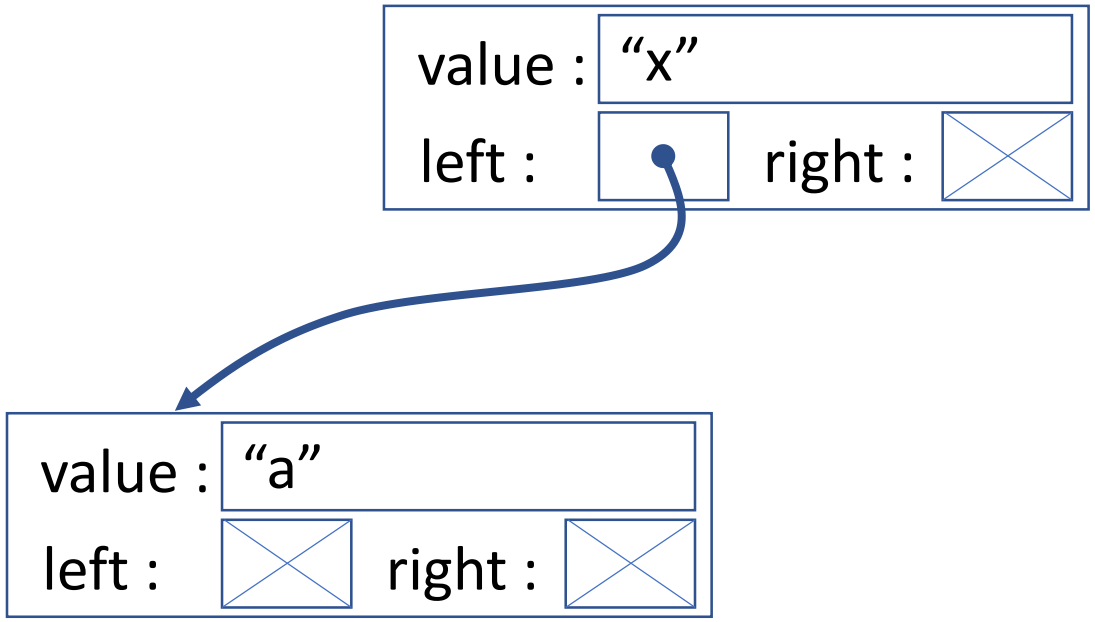
- Vectors are very good for random accesses
  - Can read or write at any index in one step
  - Can only *efficiently* insert at the end
- Linked lists are very good for insertions
  - Can insert or remove at the front in one step
  - Any other operation may be slow
- Both vector and list are oriented towards positions
  - What value is at a given position?
- How can we access efficiently by value?
  - Is value x present?
  - Remove value x if it exists

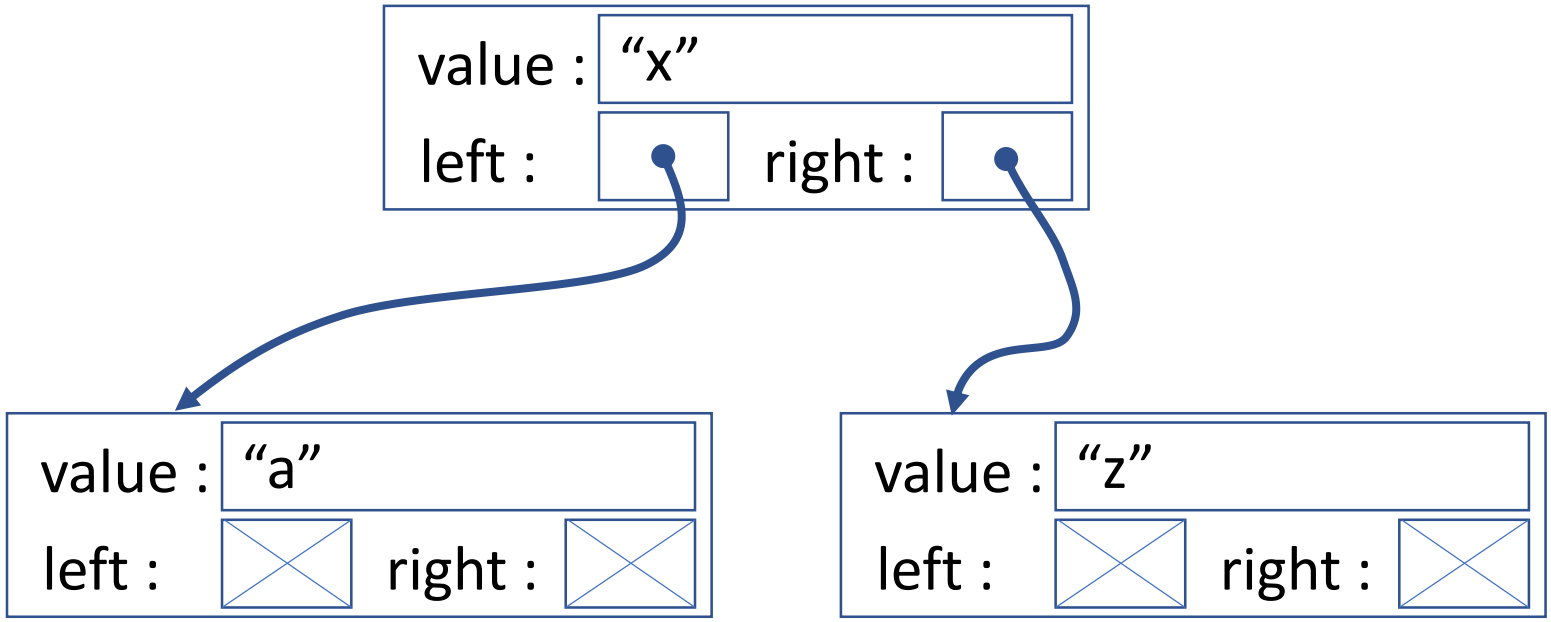
A simple tree

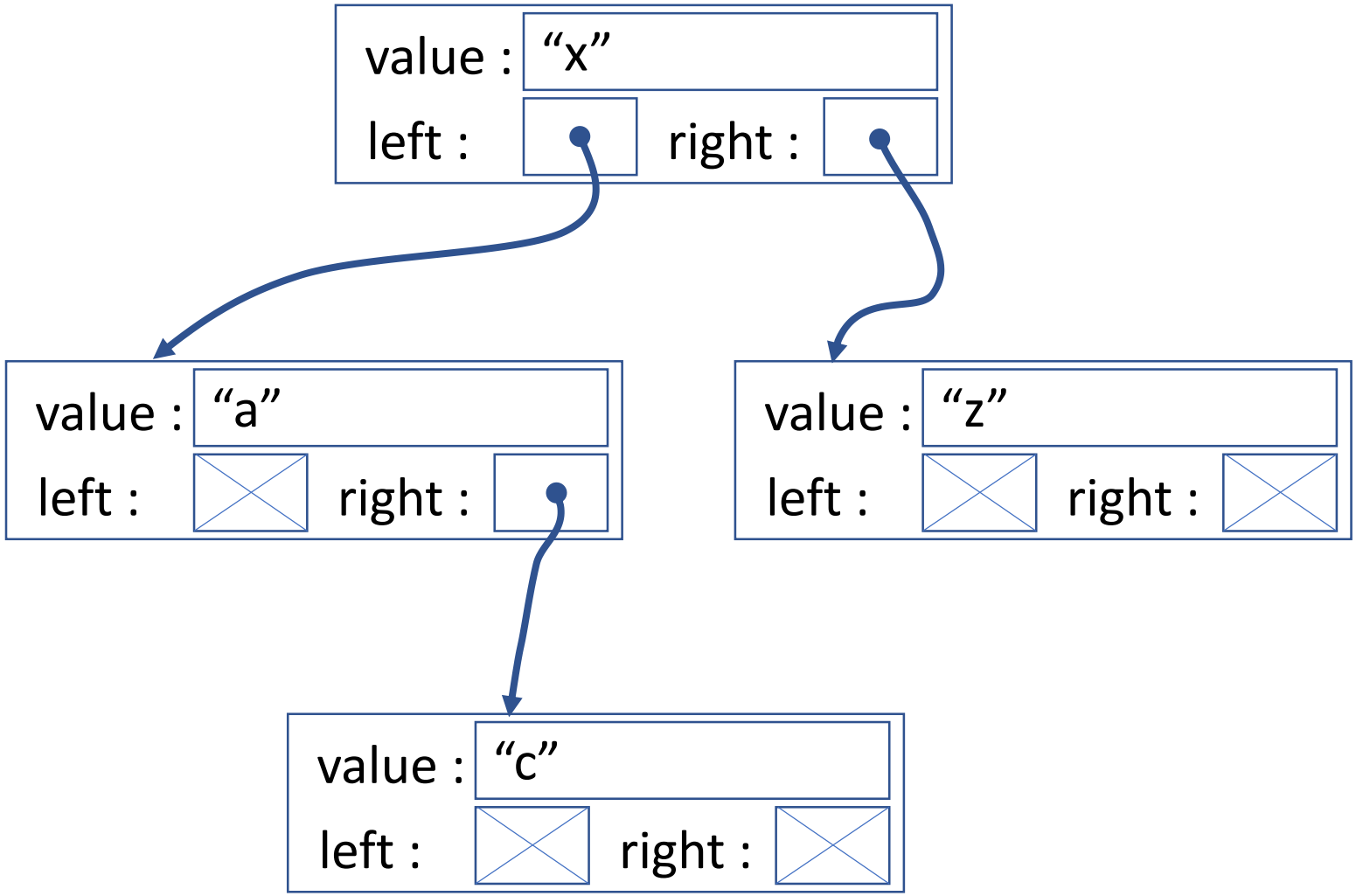
```
struct my_string_tree
{
    string value;
    my_string_tree *left;
    my_string_tree *right;
};
```

- A tree consists of nodes, with each node having:
  - A value
  - A pointer to a left sub-tree
  - A pointer to a right sub-tree
- We also have some constraints:
  - Every value in the left sub-tree is less than our value
  - Every value in the right sub-tree is not less than our value


value :
“x”
left :
right :


value :
“x”
left :
right :
value :
“a”
left :
right :


value :
“x”
left :
right :
value :
“a”
left :
right :
value :
“z”
left :
right :



Open question: how does this help us *efficiently* find nodes by value?