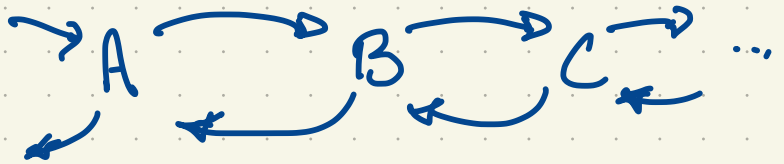
Call Stack & Heap Memory

CMPT 225

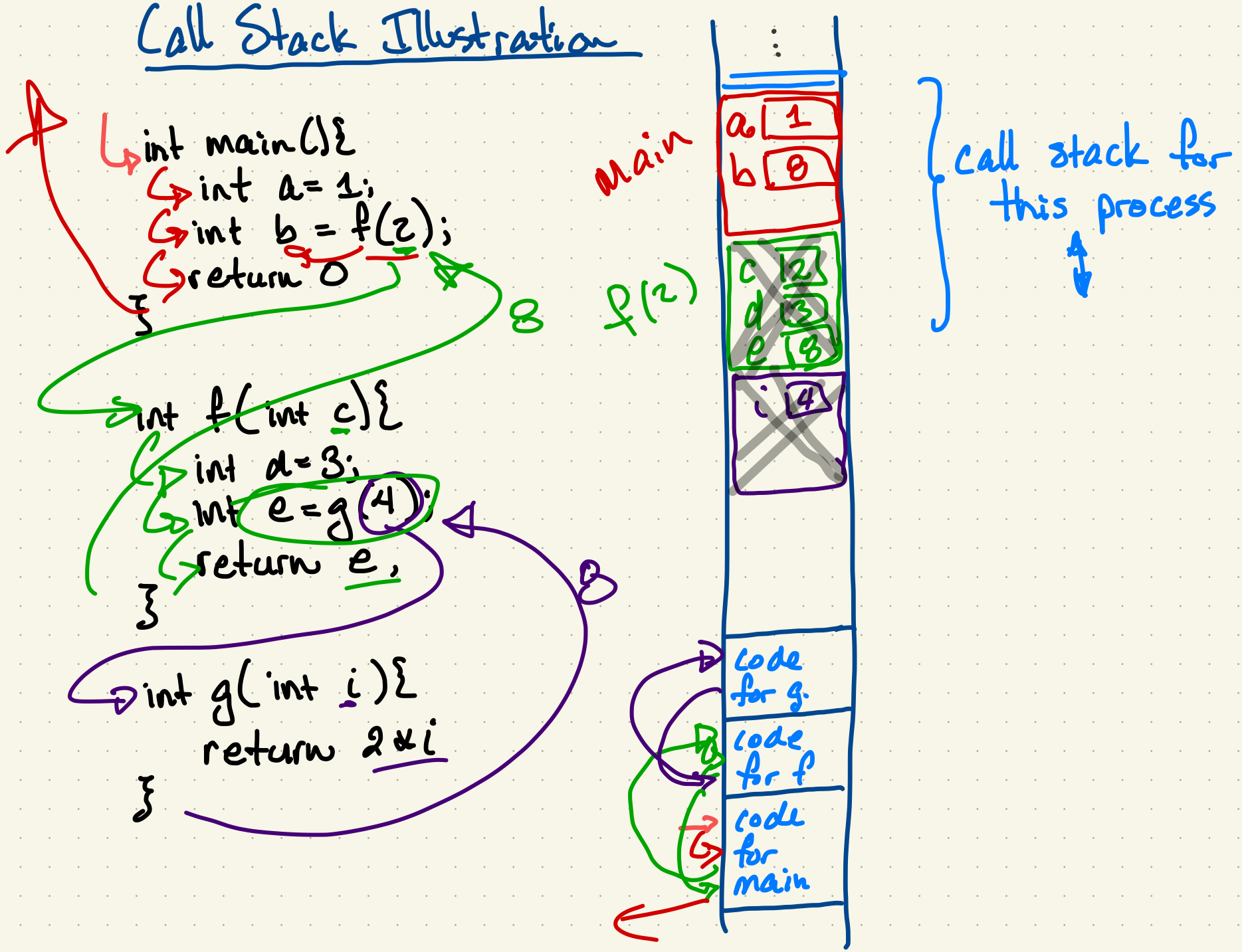
The call stack

- Suppose a function A calls another function B, which calls C.
- During execution, control passes from (the code for) A, to B, then to C.
- When execution of C ends, control must return to B, and then to A:



- At each function call, the system records where control should return to by pushing an activation record on the call stack
- The call stack also records all local variables, including the arguments to the function call.

Call Stack Illustration



```
/* A program to demonstrate printing out partial contents of the call stack */
#include
#include
using namespace std;

int print_stack(int k, int j){
   cout << "print_stack() begins" << endl;

   cout << "argument k is at &k=" << &k << "  and k=" << k << endl;
   cout << "argument j is at &j=" << &j << "  and j=" << j << endl;

   int CCC[2] = { 77777777, 88888888 } ;

   cout << "Peeking from &j up, for the space of k ints" << endl ;
   int *p = (&j)+k ;
   for( int l = k ; l > 0 ; l-- ){
       cout << p << ": " <<  setw(8) << hex << *p << " =  " << setw(11) << dec << *p << endl ;
       p -= j ;// subtractin j from an int pointer sets it to the j-th previous int
   }
   cout << "End of: print_stack()" << endl;
}

int ffff(int fun_arg){
    cout << "fun() begins" << endl;

    cout << "fun_arg is at &fun_arg=" << &fun_arg << endl;

    int BBB[2] = { 444444444, 555555555 } ;
    cout << "BBB is at BBB=" << BBB << endl;

    print_stack(40,+1);

    cout << "fun ends" << endl;
}

int main(){
    cout << "main() begins\n";

    int XXXX = 999999999 ;

    int AAAA[2] = { 111111111, 222222222 } ;//

    ffff( 333333333 );

    cout << "main() ends" << endl ;
}
```

Sample Output

```
main() begins
fun() begins
fun_arg is at &fun_arg=0x7ffff3ef9ecc
BBB is at BBB=0x7ffff3ef9ed0
print_stack() begins
argument k is at &k=0x7ffff3ef9e6c  and k=40
argument j is at &j=0x7ffff3ef9e68  and j=1
Peeking from &j up, for the space of k ints
0x7ffff3ef9f08: 5c21d9c4 =   1545722308
0x7ffff3ef9f04:        0 =            0
0x7ffff3ef9f00:        0 =            0
0x7ffff3ef9efc: 3b9ac9ff =    999999999
0x7ffff3ef9ef8:        0 =            0
0x7ffff3ef9ef4:  d3ed78e =    222222222
0x7ffff3ef9ef0:  69f6bc7 =    111111111
0x7ffff3ef9eec:        0 =            0
0x7ffff3ef9ee8:   400c92 =      4197522
0x7ffff3ef9ee4:     7fff =        32767
0x7ffff3ef9ee0: f3ef9f00 =   -202400000
0x7ffff3ef9edc:        0 =            0
0x7ffff3ef9ed8:        0 =            0
0x7ffff3ef9ed4: 211d1ae3 =    555555555
0x7ffff3ef9ed0: 1a7daf1c =    444444444
0x7ffff3ef9ecc: 13de4355 =    333333333
0x7ffff3ef9ec8: f3ef9fe0 =   -202399776
0x7ffff3ef9ec4:        0 =            0
0x7ffff3ef9ec0:        0 =            0
0x7ffff3ef9ebc:        0 =            0
0x7ffff3ef9eb8:   400c3e =      4197438
0x7ffff3ef9eb4:     7fff =        32767
0x7ffff3ef9eb0: f3ef9ee0 =   -202400032
0x7ffff3ef9eac:        0 =            0
0x7ffff3ef9ea8:        0 =            0
0x7ffff3ef9ea4:     7fff =        32767
0x7ffff3ef9ea0: f3ef9fe0 =   -202399776
0x7ffff3ef9e9c:        0 =            0
0x7ffff3ef9e98:        0 =            0
0x7ffff3ef9e94:       30 =           48
0x7ffff3ef9e90: 5c01cbc0 =   1543621568
0x7ffff3ef9e8c:        9 =            9
0x7ffff3ef9e88: 5c260440 =   1545995328
0x7ffff3ef9e84:     7fff =        32767
0x7ffff3ef9e80: f3ef9e80 =   -202400128
0x7ffff3ef9e7c:       30 =           48
0x7ffff3ef9e78: 5c26b397 =   1546040215
0x7ffff3ef9e74:  54c5638 =     88888888
0x7ffff3ef9e70:  4a2cb71 =     77777777
0x7ffff3ef9e6c:       28 =           40
End of: print_stack()
fun ends
main() ends
```

xxxx

AAAA

BBBB

C

— K

— I would be here.

Dynamic Memory or Heap

- Variables declared in functions are stored on the call stack.
- These variables
  - are of fixed size
  - are destroyed when the function they are defined in terminates.
- We often want a function f to create data that can be used after f returns.
  - In particular, dynamic data structures require this!
- This data is stored in "the heap", a region of memory that is allocated dynamically as needed.

In C++:

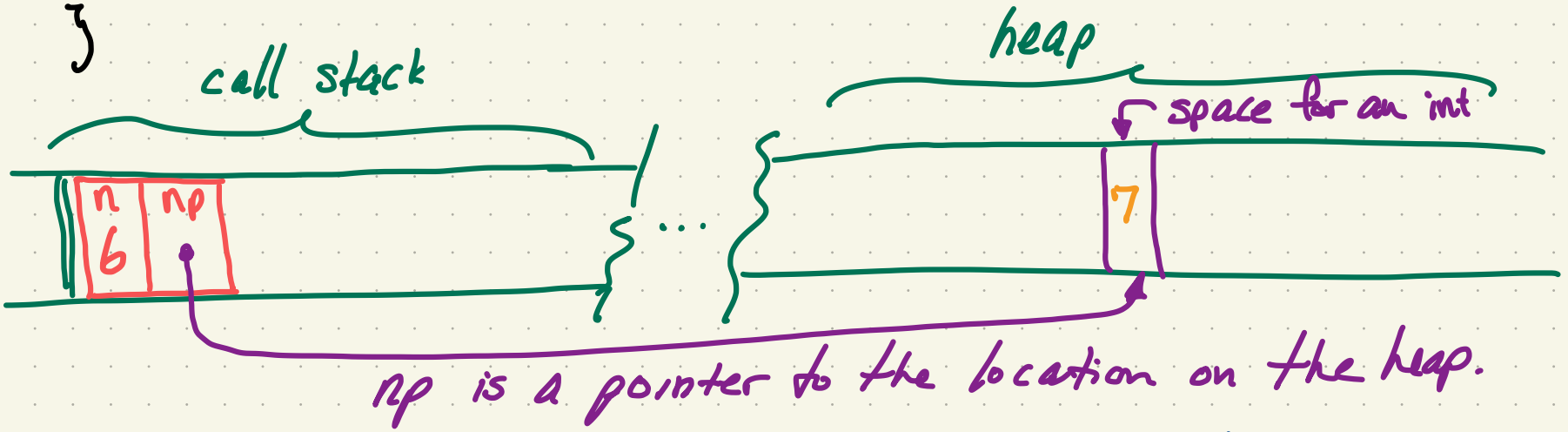
- Basic (or primitive) types can be stored on the call stack or on the heap.
- Objects (e.g. instances of classes) can be stored on the call stack or on the heap.
- Variables declared in functions are on the stack
- Allocation on the heap is denoted by "new".

Ex: Basic Types on Call Stack & Heap

```
f(){
    int n; // n is on stack
    n=6;
    int * np; // np is on stack
    np = new int; // new int is stored in heap
    *np = 7;      // np points to the location.
}
```

Ex: Basic Types on Call Stack & Heap

```
f(){
   int n; // n is on stack
   n = 6;
   int * np; // np is on stack
   np = new int; // new int is stored in heap
   *np = 7;      // np points to the location.
}
```

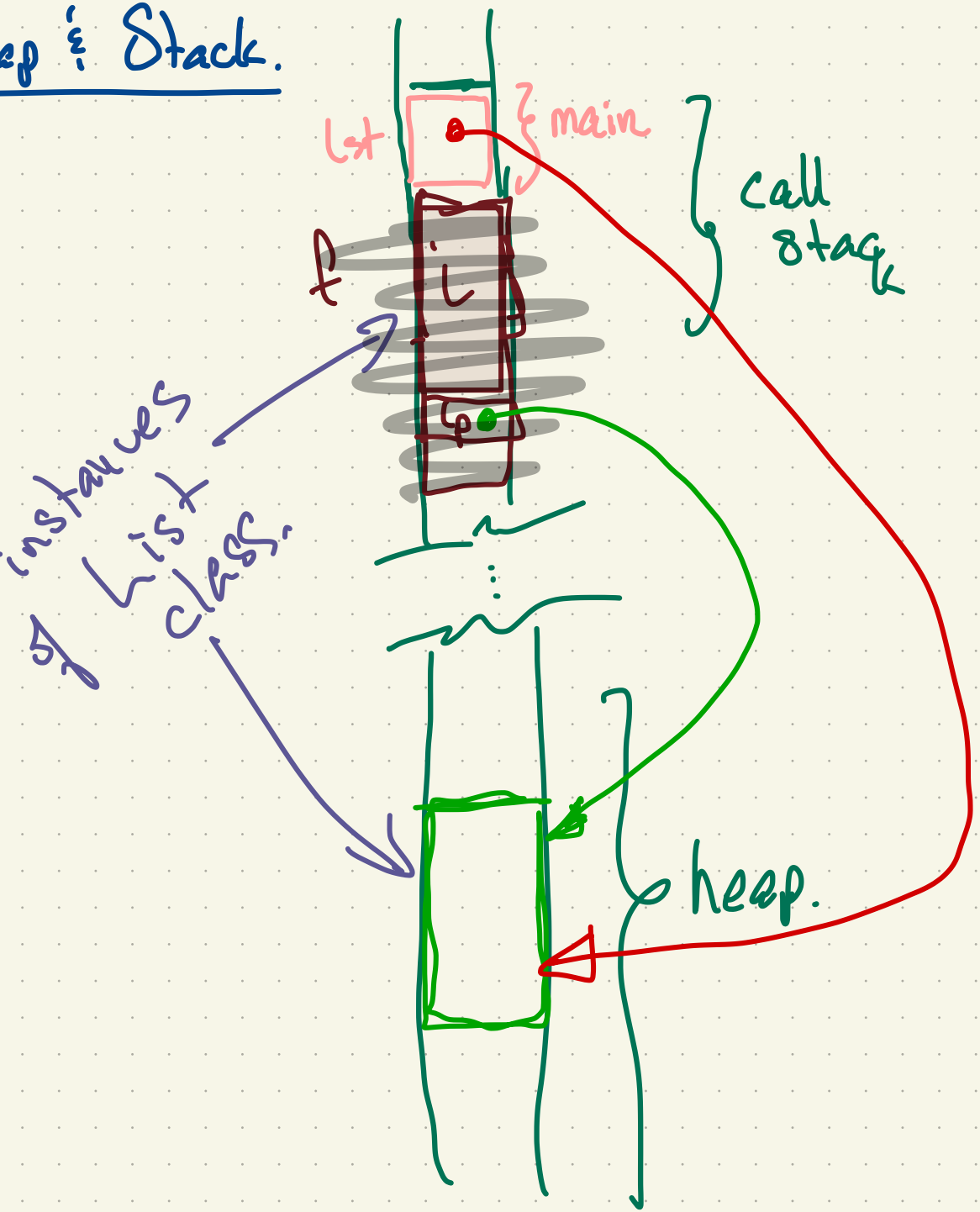


*When f ends, np is gone (the stack is popped),
but the space it pointed to is not.

Class Instances on Heap & Stack.

```
List * f(){
    List Li;
    List * Lp;
    :
    Lp = new List();
    :
    return Lp
}
```

```
main(){
    :
    List * lst = f();
    :
}
```



Class Instances on Heap & Stack.

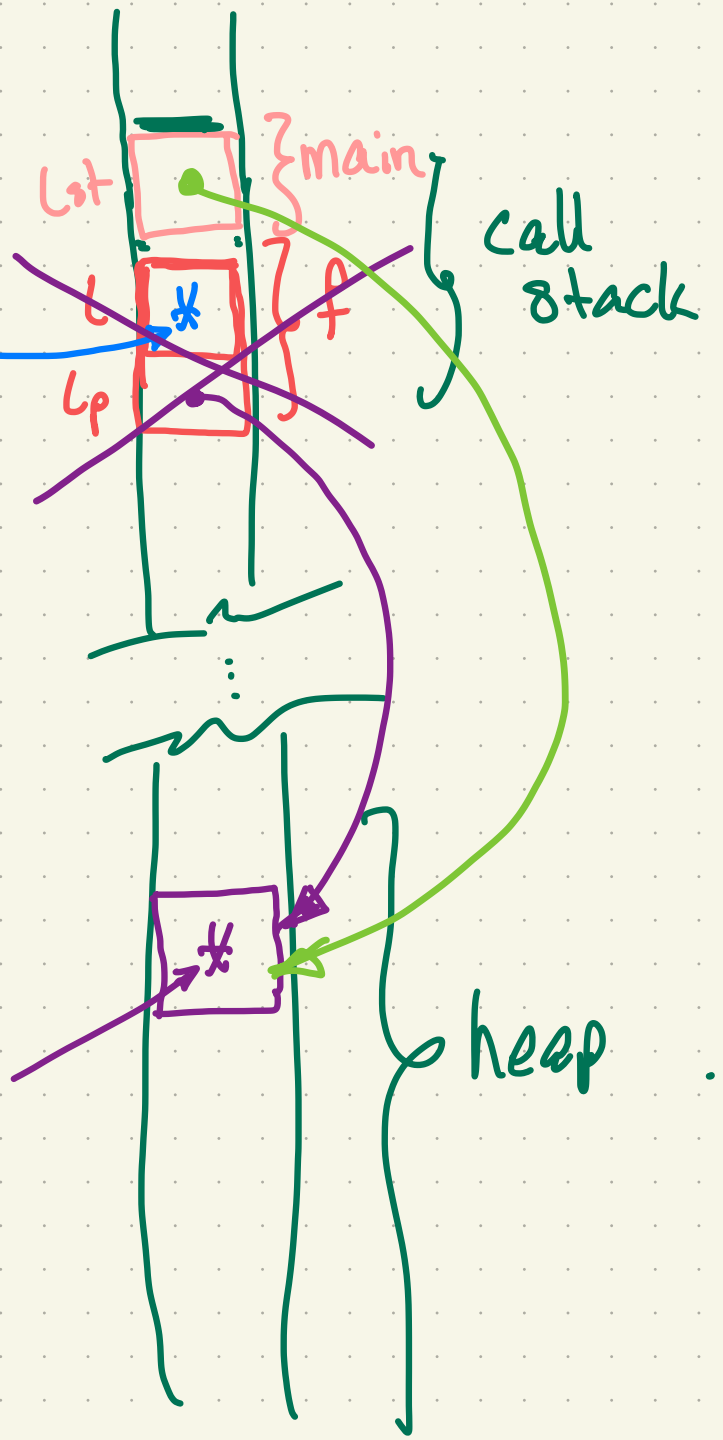
```
List * f(){
    List  L;
    List * Lp;
     :
    Lp = new List();
     :
    return Lp //returns pointer
              // to *
}
```

entire List object

```
main(){
     :
    List * lst = f();
     :
}
```

lst becomes a pointer to the List object *

entire List object.



Accessing Instance Members in C++

Suppose a class **Store** with

- a data member x. (an int)
- a function put(v) that stores v in x.
- a function get() that returns the value of x.

Consider this code fragment:

```
f(){
    ⋮
    Store s1;
    s1.put(5);
    y = s1.get();// y=5
    ⋮
   Store *s2;
   s2= new Store();
   S2.put(5);
   y = s2get()
   ⋮
   *s2.put(5);
   y= *S2.get();
   ⋮
   (*s2).put(5);
   y = (*s2).get();
   ⋮
   s2->put(5); // equiv. to (*s).put(5)
   y = s2->get(); // equiv. to y = (*s).get()
```

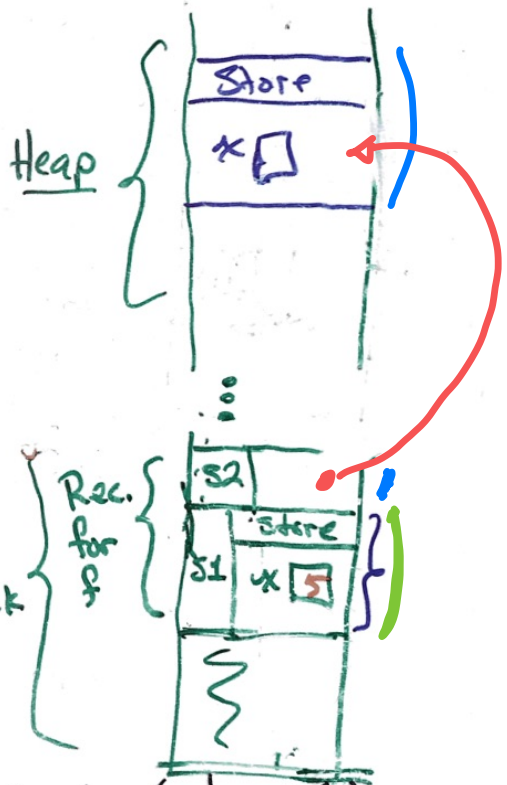
Accessing Instance Members in C++

Suppose a class **Store** with

- a data member x. (an int)
- a function put(v) that stores v in x.
- a function get() that returns the value of x.

Consider this code fragment:

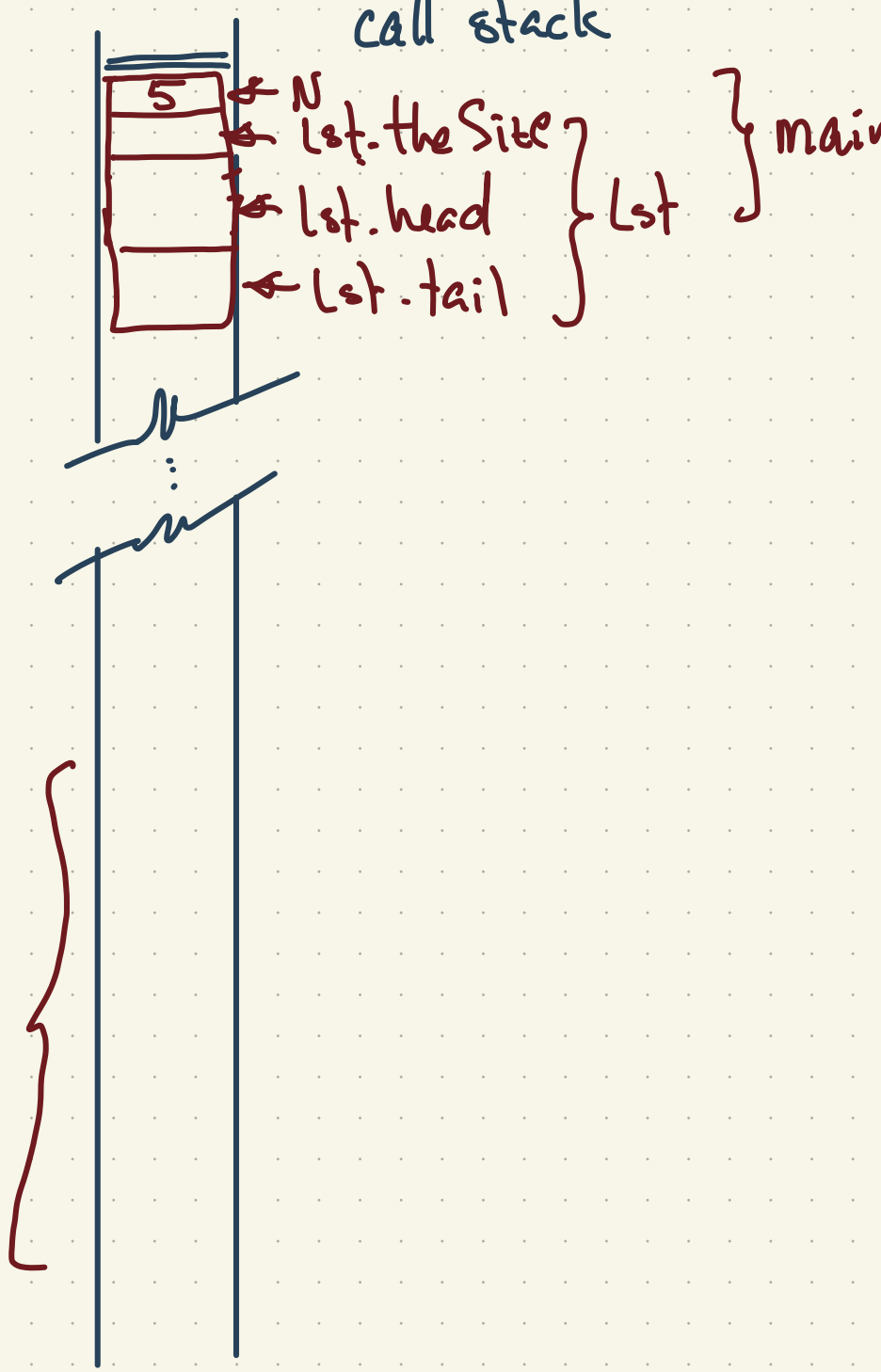
```
f(){
    Store s1;
    s1.put(5);
    y = s1.get(); // y = 5

    Store *s2;
    s2 = new Store(); XX
✗   s2.put(5);
✗   y = s2.get()

✗   *s2.put(5);
✗   y = *s2.get();

✓   (*s2).put(5);
✓   y = (*s2).get();

✓   s2->put(5); // equiv. to (*s).put(5)
✓   y = s2->get(); // equiv. to y = (*s).get()
```



Using the Textbook List Class

```
#include "dsexceptions.h"
#include "List.h"
using namespace std;

int main( )
{
    const int N = 5;
    List<int> lst;

    for( int i = N - 1; i > 0; --i )
    {
        lst.push_front( i );
    }

    return 0;
}
```

```
private:
   int    theSize;
   Node *head;
   Node *tail;

   void init( )
   {
       theSize = 0;
```



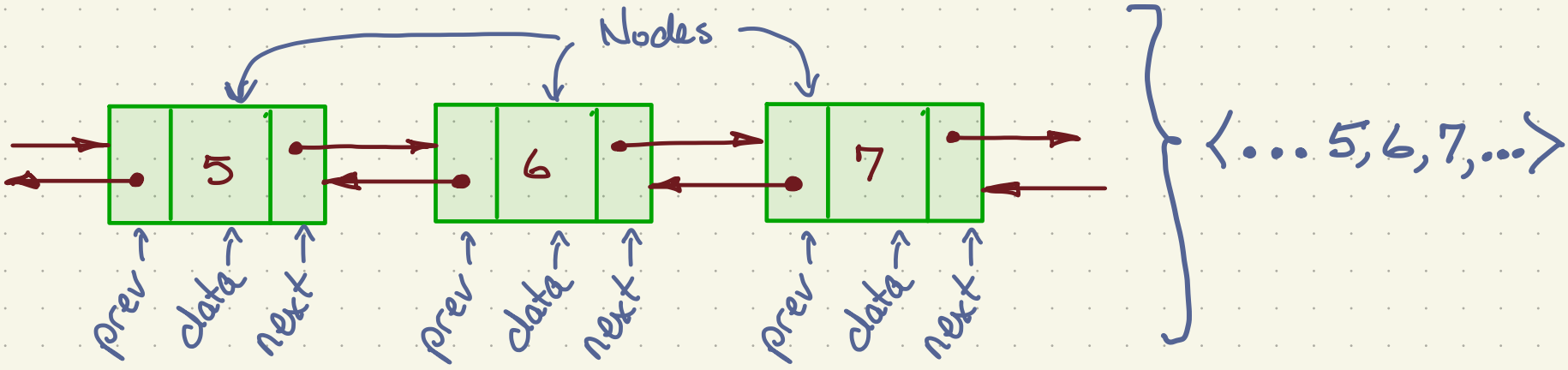
The List Class (A doubly-linked list implementation of a List ADT)

```
template <typename Object>
class List
{
  private:
    // The basic doubly linked list node.
    // Nested inside of List, can be public
    // because the Node is itself private
    struct Node
    {
        Object  data;
        Node   *prev;
        Node   *next;

        Node( const Object & d = Object{ }, Node * p = nullptr, Node * n = nullptr )
          : data{ d }, prev{ p }, next{ n } { }

        Node( Object && d, Node * p = nullptr, Node * n = nullptr )
          : data{ std::move( d ) }, prev{ p }, next{ n } { }
    };
```

list element

pointer to next node

pointer to previous node



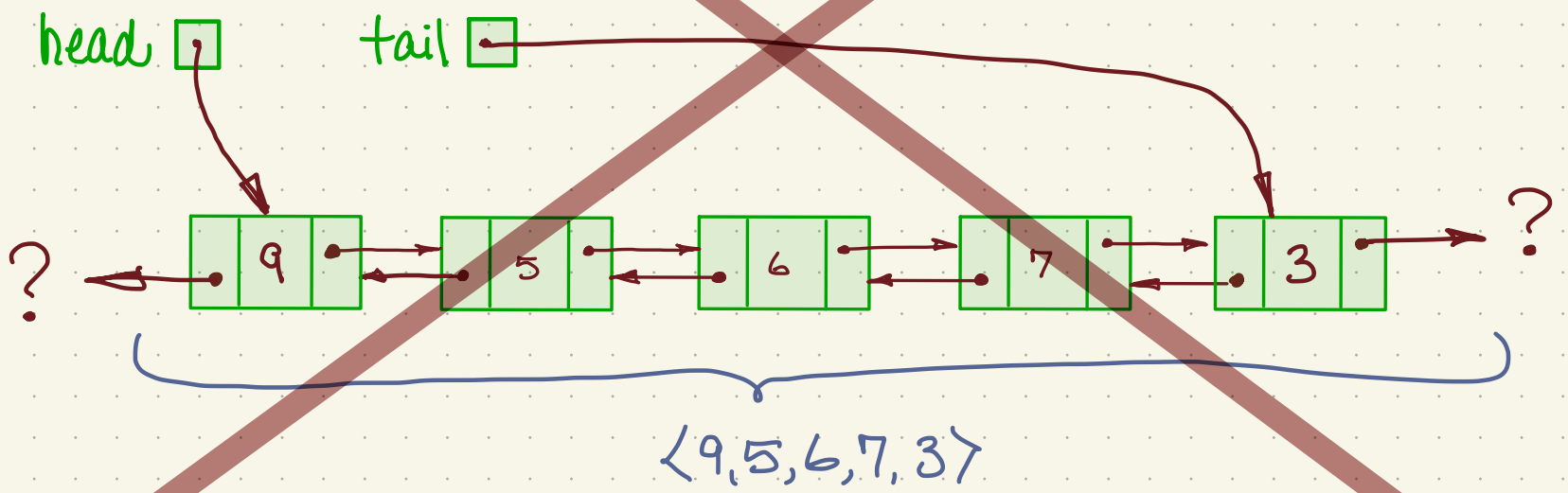
The List Class (A doubly-linked list implementation of a List ADT)

```
private:
  int    theSize;
  Node *head;
  Node *tail;

  void init( )
  {
      theSize = 0;
```

# of list elements

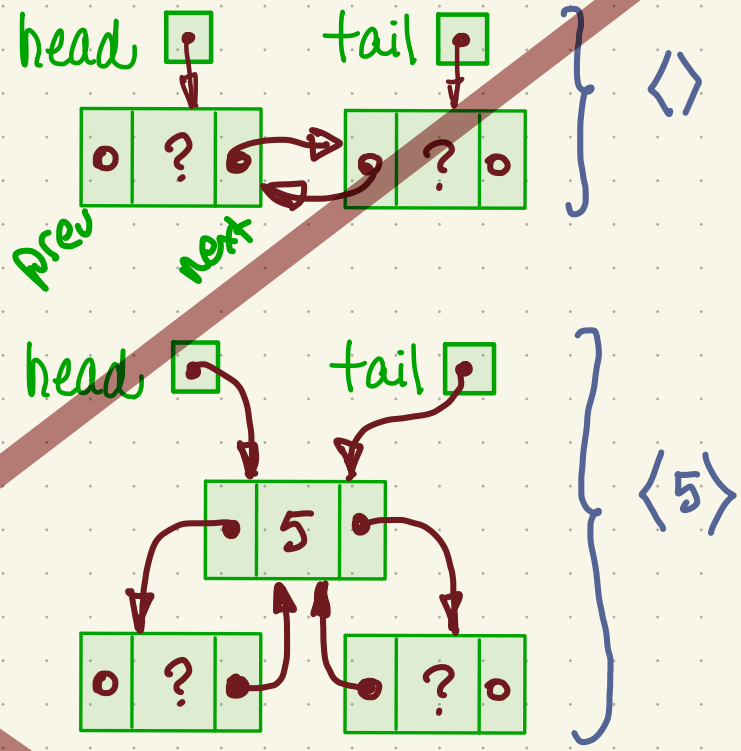
pointer to first list element

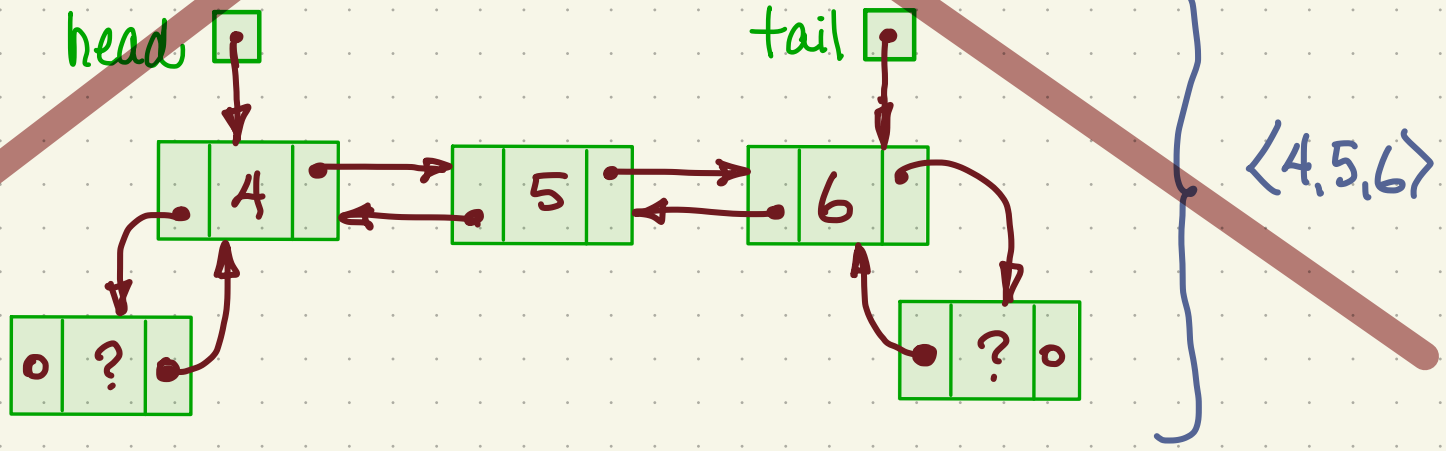
pointer to last list element



The List Class (A doubly-linked list implementation of a List ADT)

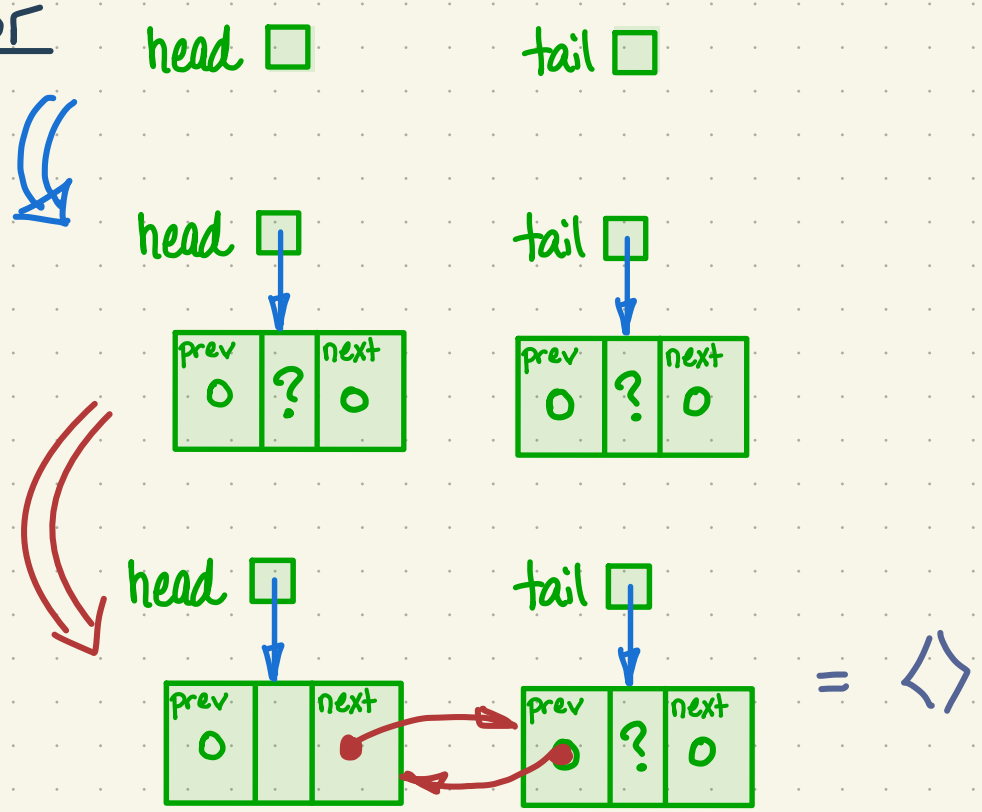
```
  private:
    int   theSize;
    Node *head;
    Node *tail;

    void init( )
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
    }
};
```





The List Class - Constructor

```
  private:
    int    theSize;
    Node *head;
    Node *tail;

    void init( )
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
    }
};
```




```
  struct Node
  {
      Object  data;
      Node   *prev;
      Node   *next;

      Node( const Object & d = Object{ }, Node * p = nullptr, Node * n = nullptr )
        : data{ d }, prev{ p }, next{ n } { }

      Node( Object && d, Node * p = nullptr, Node * n = nullptr )
        : data{ std::move( d ) }, prev{ p }, next{ n } { }
  };
```

The list Class - The iterators

Data member: `Node * current;` // a pointer to a Node.
// (the list iterators are
// implemented with pointers.)

Constructors:
```
iterator ( Node * p): const_ iterator { p } { }
const_iterator(Node *p): current { p }
```
// turns a pointer into an iterator.

function:
```
iterator end() { return iterator(tail)
```
// turns the tail pointer into the iterato "end".
// it corresponds to "just past the end"
// of the list.

The list Class - the push_back function

```
// add an element to the tail end of the list
void push_back( const Object & x ){ insert( end(), x );}
```

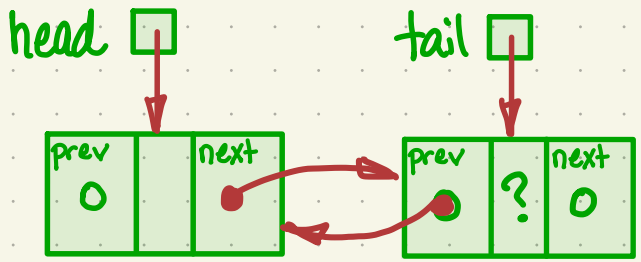
the end iterator ↑ element ↑ to add

```
iterator insert( iterator itr, const Object & x ){
    Node * p = itr.current; // turns the iterator into a pointer
    ++theSize; // increments size variable
    return iterator(    ) ← turns the pointer into an iterator

p->prev = p->prev->next = new Node(x, p->prev, p)
}
```

stores a pointer to N in p->prev and in p->prev->next.

makes a new node N

The list Class – Inserting the first element.




```
iterator insert( iterator itr, const Object & x ){
    Node * p = itr.current; //
    ++theSize;
    return iterator( * )
}
```

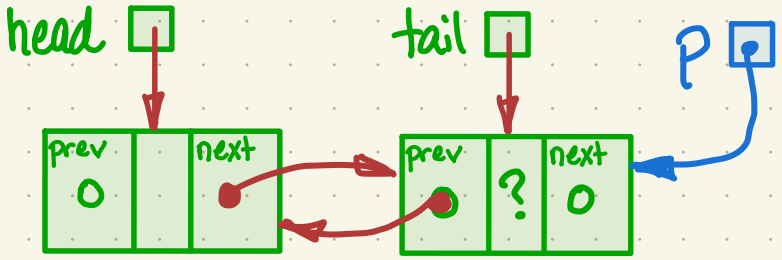
this is end

this is tail

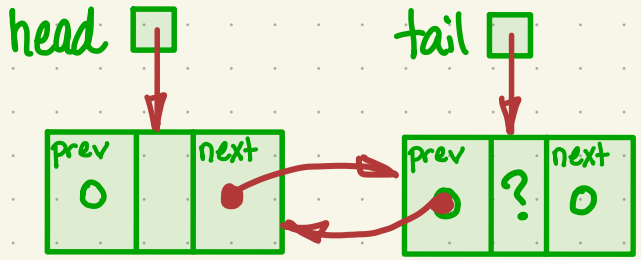
turns pointer into iterator

* p->prev = p->prev->next

= new Node(x, p->prev, p)



The list Class – Inserting the first element.




```
iterator insert( iterator itr, const Object & x ){
    Node * p = itr.current; //
    ++theSize;
    return iterator( * )
}
```

this is end

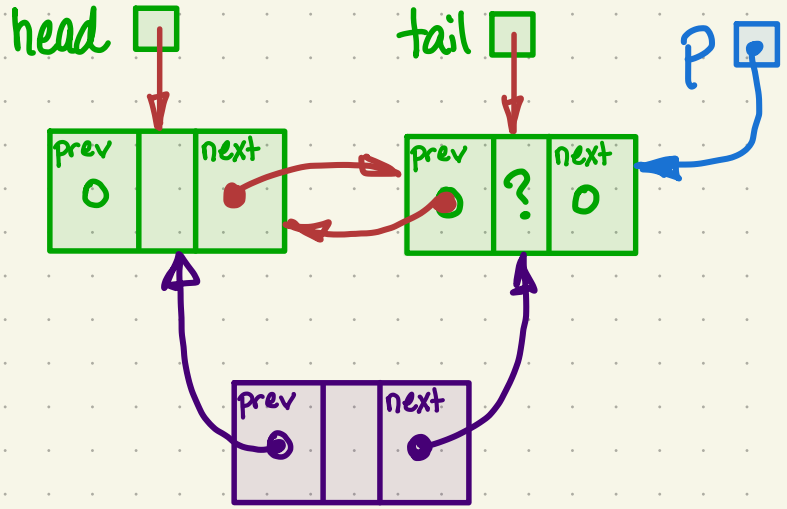
this is tail

turns pointer into iterator

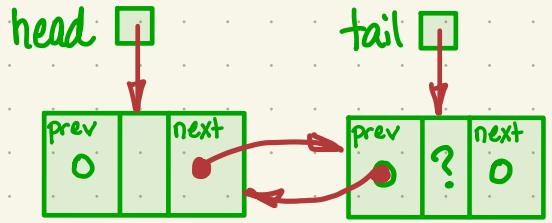
* `p->prev = p->prev->next`

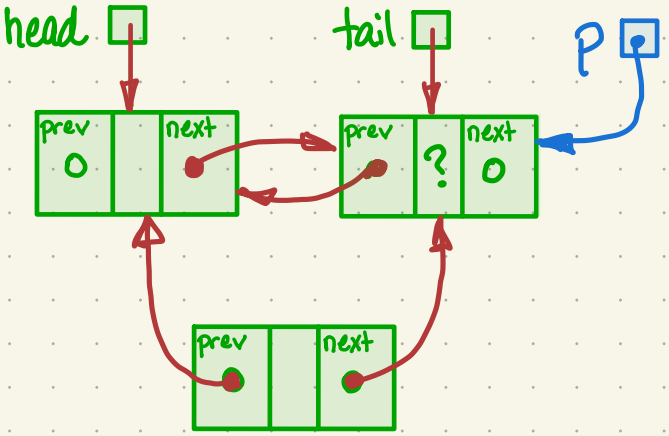
`= new Node(x, p->prev, p)`

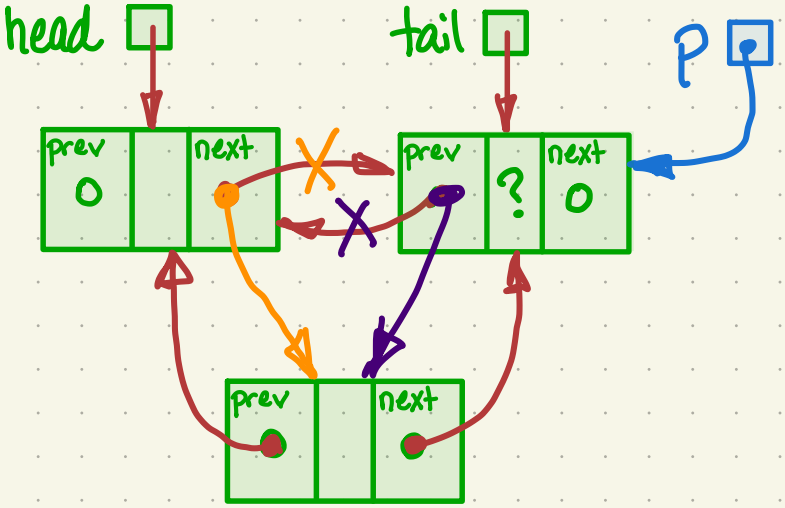
- new list element
- initial value of prev
- initial value of next



The list Class – Inserting the first element.







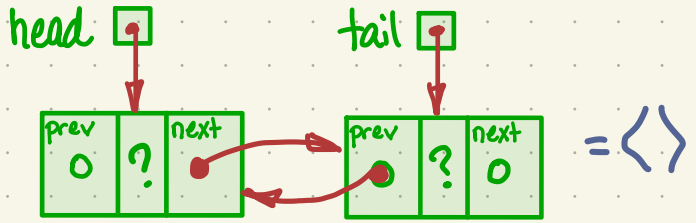

```
iterator insert( iterator itr, const Object & x ){
    Node * p = itr.current; //
    ++theSize;
    return iterator( * )   <- turns pointer into iterator
}
```

* `p->prev = p->prev->next`

`= new Node(x, p->prev, p)`

pointer to new node

The list Class – Inserting the first element.



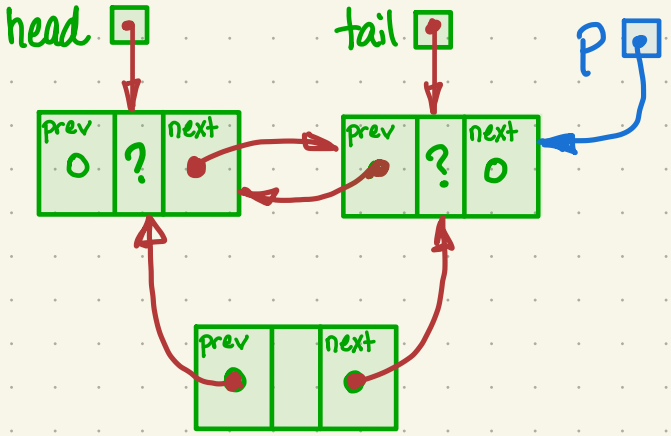
$= \langle \rangle$

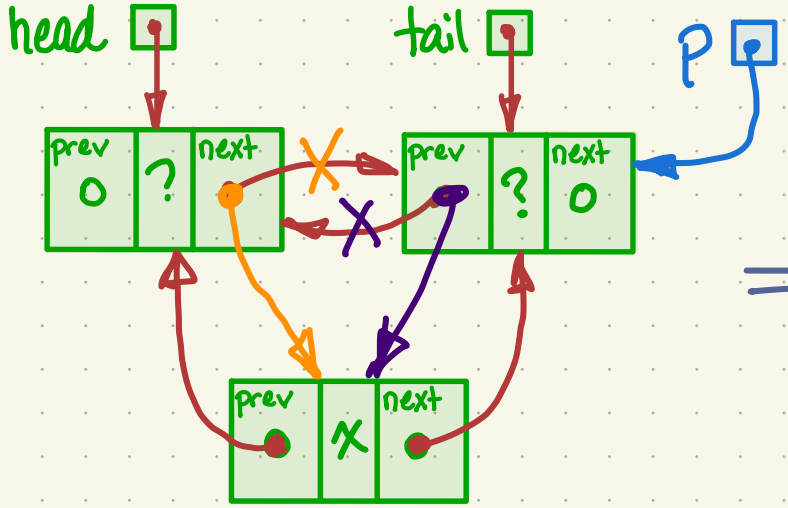
```
iterator insert( iterator itr, const Object & x ){
    Node * p = itr.current; //
    ++theSize;
    return iterator( * )   ← turns pointer into iterator
}
```

* `p->prev = p->prev->next`

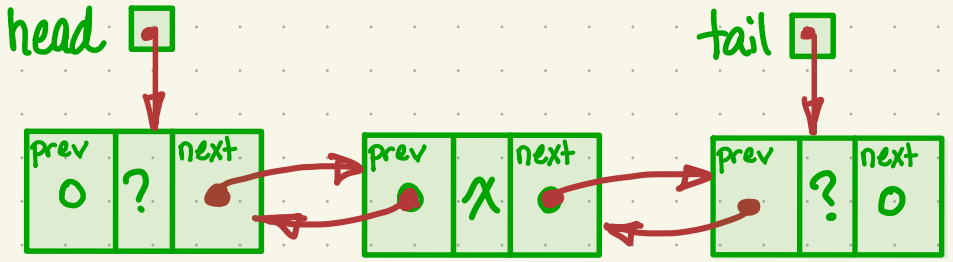
`= new Node(x, p->prev, p)`

pointer to new node





$\Rightarrow$

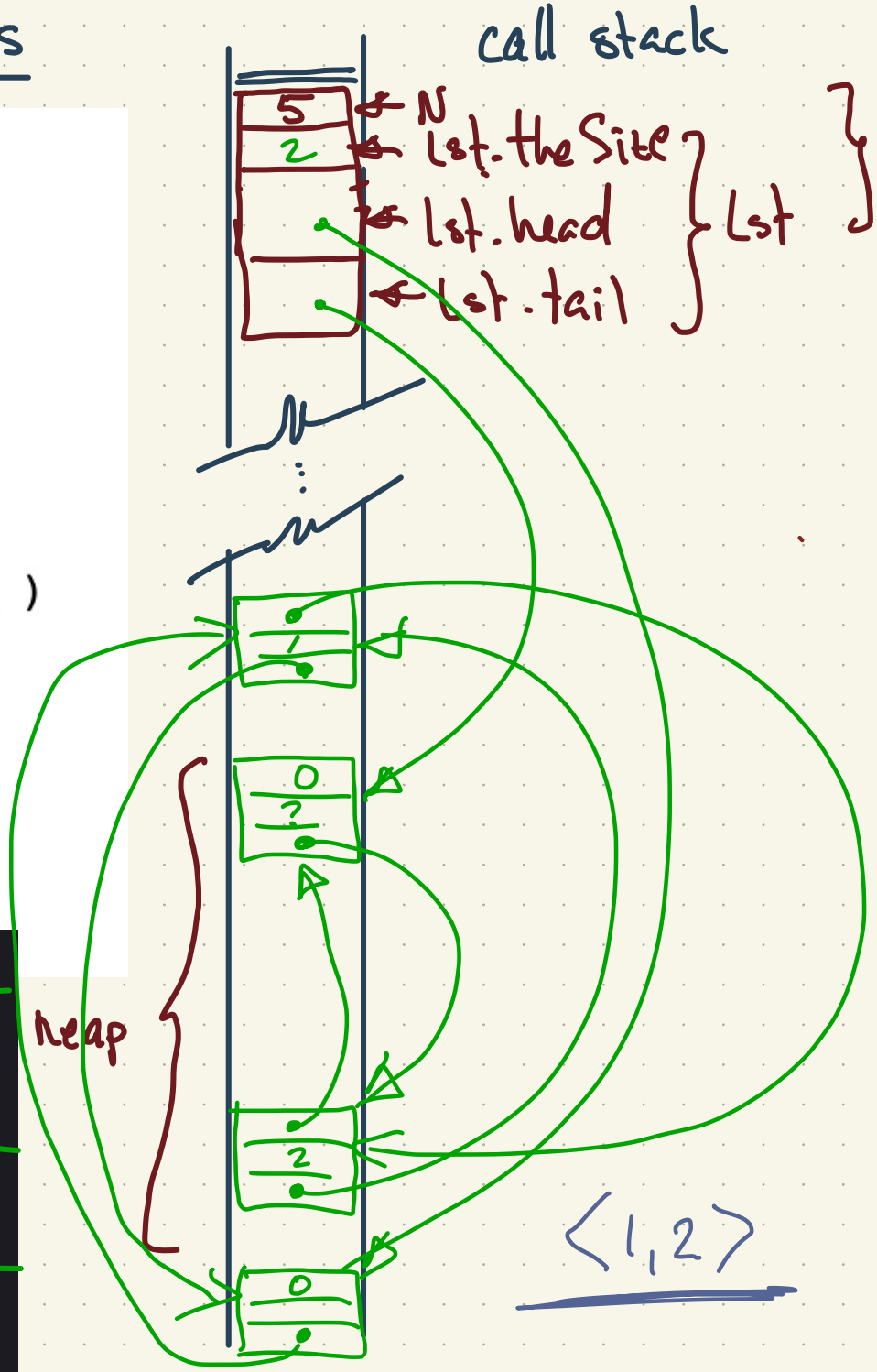


$= \langle x \rangle$

Using the Textbook List Class

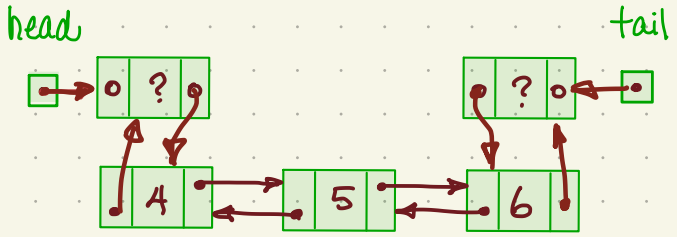
```
#include "dsexceptions.h"
#include "List.h"
using namespace std;

int main( )
{
    const int N = 5;
    List<int> lst;

    for( int i = N - 1; i > 0; --i )
    {
        lst.push_front( i );
    }

    return 0;
}
```

```
private:
  int    theSize;
  Node *head;
  Node *tail;

  void init( )
  {
      theSize = 0;
```

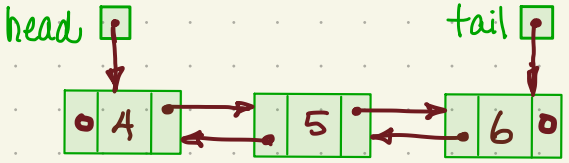


Linked List Ends

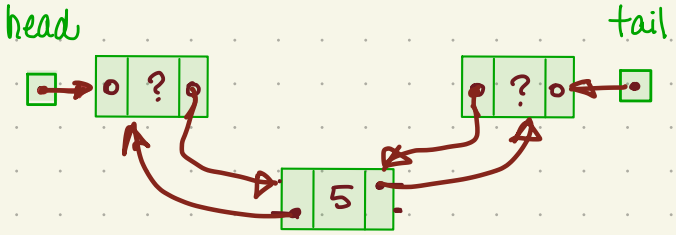
⟨4,5,6⟩:



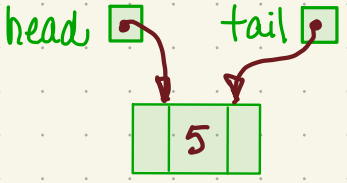
vs.



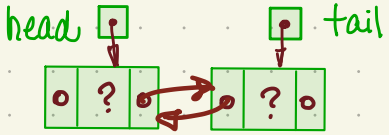
⟨5⟩:



vs.



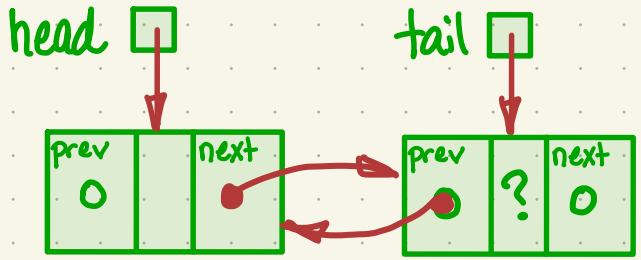
⟨⟩:



vs.

End

The list Class – Inserting the first element.




```
iterator insert( iterator itr, const Object & x ){
    Node * p = itr.current; // turns iterator into pointer
    ++theSize;
    return iterator( * ) ← turns pointer into iterator
}
```

* `p->prev = p->prev->next`

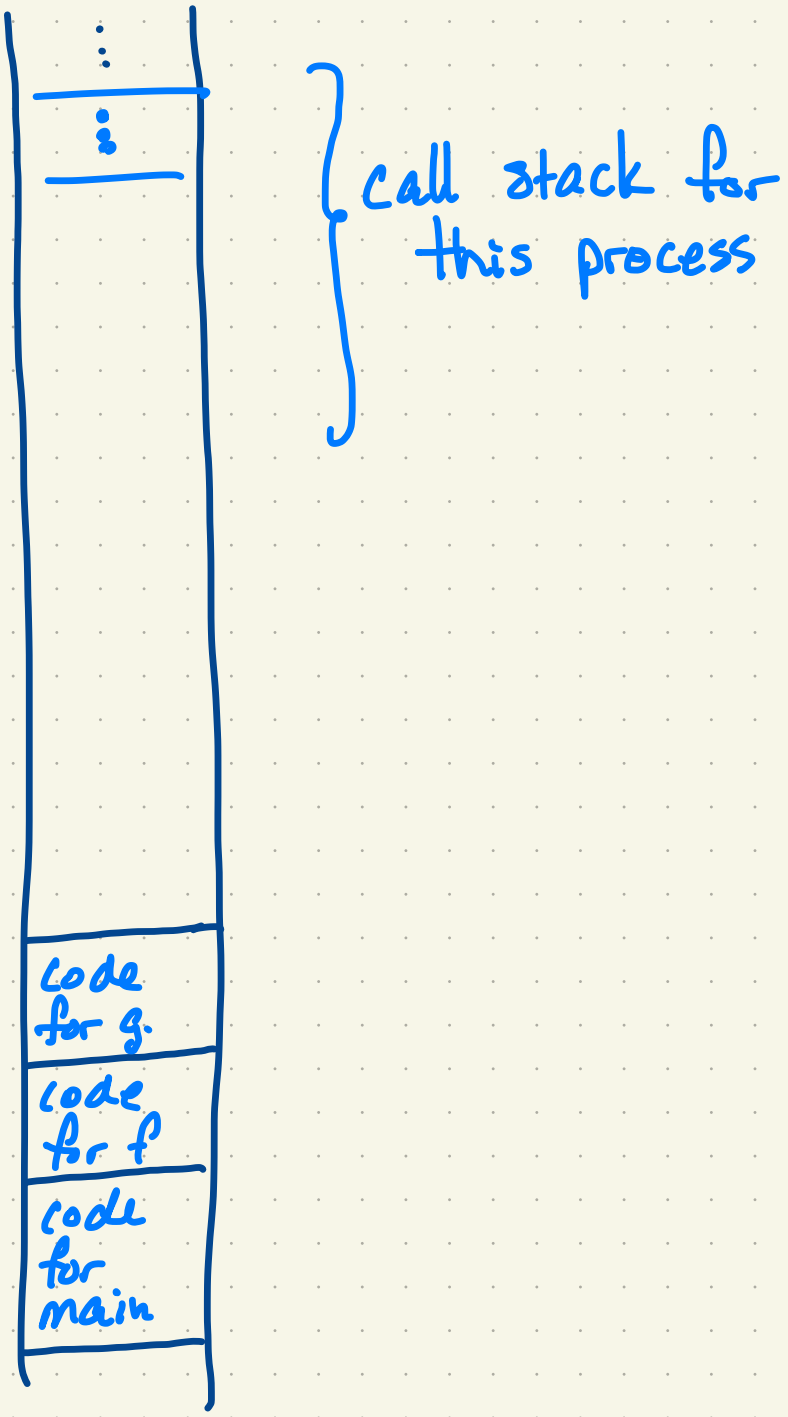
`= new Node(x, p->prev, p)`

```
void push_back( const Object & x){
    insert( end(), x );
}

iterator insert( iterator itr, const Object & x){
    Node * p = itr.current;
    ++ theSize;
    return iterator(
        p->prev = p->prev->next = new Node(x, p->prev, p));
}
```

```
/* A program to demonstrate printing out partial contents of the call stack */
#include
#include
using namespace std;

int print_stack(int k, int j){
   cout << "print_stack() begins" << endl;

   cout << "argument k is at &k=" << &k << "  and k=" << k << endl;
   cout << "argument j is at &j=" << &j << "  and j=" << j << endl;

   int CCC[2] = { 77777777, 88888888 } ;

   cout << "Peeking from &j up, for the space of k ints" << endl ;
   int *p = (&j)+k ;
   for( int l = k ; l > 0 ; l-- ){
       cout << p << ": " <<  setw(8) << hex << *p << " =  " << setw(11) << dec << *p << endl ;
       p -= j ;// subtractin j from an int pointer sets it to the j-th previous int
   }
   cout << "End of: print_stack()" << endl;
}

int ffff(int fun_arg){
    cout << "fun() begins" << endl;

    cout << "fun_arg is at &fun_arg=" << &fun_arg << endl;

    int BBB[2] = { 444444444, 555555555 } ;
    cout << "BBB is at BBB=" << BBB << endl;

    print_stack(40,+1);

    cout << "fun ends" << endl;
}

int main(){
    cout << "main() begins\n";

    int XXXX = 999999999 ;

    int AAAA[2] = { 111111111, 222222222 } ;//

    ffff( 333333333 );

    cout << "main() ends" << endl ;
}
```

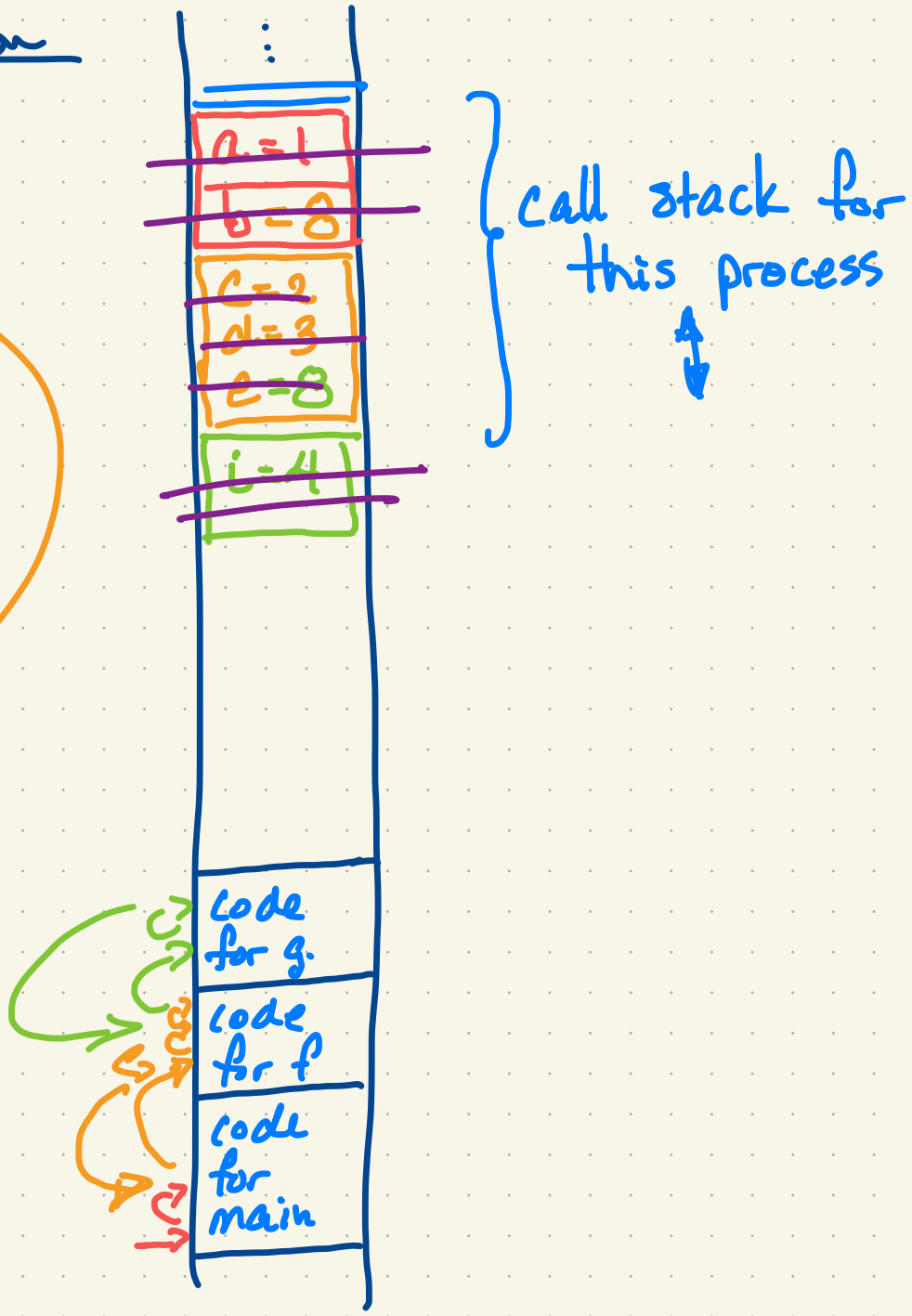
Call Stack Illustration

```
int main(){
    int a = 1;
    int b = f(2);
    return 0
}

int f(int c){
    int d = 3;
    int e = g(4);
    return e;
}

int g(int i){
    return 2*i
}
```



Call Stack Illustration

```
int main(){
    int a = 1;
    int b = f(2);
    return 0
}

int f(int c){
    int d = 3;
    int e = g(4);
    return e;
}

int g(int i){
    return 2*i
}
```



```
/* A program to demonstrate printing out partial contents of the call stack */
#include
#include
using namespace std;

int print_stack(int k, int j){
   cout << "print_stack() begins" << endl;

   cout << "argument k is at &k=" << &k << "  and k=" << k << endl;
   cout << "argument j is at &j=" << &j << "  and j=" << j << endl;

   int CCC[2] = { 77777777, 88888888 } ;

   cout << "Peeking from &j up, for the space of k ints" << endl ;
   int *p = (&j)+k ;
   for( int l = k ; l > 0 ; l-- ){
       cout << p << ": " <<  setw(8) << hex << *p << " =  " << setw(11) << dec << *p << endl ;
       p -= j ;// subtractin j from an int pointer sets it to the j-th previous int
   }
   cout << "End of: print_stack()" << endl;
}

int ffff(int fun_arg){
    cout << "fun() begins" << endl;

    cout << "fun_arg is at &fun_arg=" << &fun_arg << endl;

    int BBB[2] = { 444444444, 555555555 } ;
    cout << "BBB is at BBB=" << BBB << endl;

    print_stack(40,+1);

    cout << "fun ends" << endl;
}

int main(){
    cout << "main() begins\n";

    int XXXX = 999999999 ;

    int AAAA[2] = { 111111111, 222222222 } ;//

    ffff( 333333333 );

    cout << "main() ends" << endl ;
}
```

```
main() begins
fun() begins
fun_arg is at &fun_arg=0x7ffff3ef9ecc
BBB is at BBB=0x7ffff3ef9ed0
print_stack() begins
argument k is at &k=0x7ffff3ef9e6c  and k=40
argument j is at &j=0x7ffff3ef9e68  and j=1
Peeking from &j up, for the space of k ints
0x7ffff3ef9f08: 5c21d9c4 =   1545722308
0x7ffff3ef9f04:        0 =            0
0x7ffff3ef9f00:        0 =            0
0x7ffff3ef9efc: 3b9ac9ff =    999999999
0x7ffff3ef9ef8:        0 =            0
0x7ffff3ef9ef4:  d3ed78e =    222222222
0x7ffff3ef9ef0:  69f6bc7 =    111111111
0x7ffff3ef9eec:        0 =            0
0x7ffff3ef9ee8:   400c92 =      4197522
0x7ffff3ef9ee4:     7fff =        32767
0x7ffff3ef9ee0: f3ef9f00 =   -202400000
0x7ffff3ef9edc:        0 =            0
0x7ffff3ef9ed8:        0 =            0
0x7ffff3ef9ed4: 211d1ae3 =    555555555
0x7ffff3ef9ed0: 1a7daf1c =    444444444
0x7ffff3ef9ecc: 13de4355 =    333333333
0x7ffff3ef9ec8: f3ef9fe0 =   -202399776
0x7ffff3ef9ec4:        0 =            0
0x7ffff3ef9ec0:        0 =            0
0x7ffff3ef9ebc:        0 =            0
0x7ffff3ef9eb8:   400c3e =      4197438
0x7ffff3ef9eb4:     7fff =        32767
0x7ffff3ef9eb0: f3ef9ee0 =   -202400032
0x7ffff3ef9eac:        0 =            0
0x7ffff3ef9ea8:        0 =            0
0x7ffff3ef9ea4:     7fff =        32767
0x7ffff3ef9ea0: f3ef9fe0 =   -202399776
0x7ffff3ef9e9c:        0 =            0
0x7ffff3ef9e98:        0 =            0
0x7ffff3ef9e94:       30 =           48
0x7ffff3ef9e90: 5c01cbc0 =   1543621568
0x7ffff3ef9e8c:        9 =            9
0x7ffff3ef9e88: 5c260440 =   1545995328
0x7ffff3ef9e84:     7fff =        32767
0x7ffff3ef9e80: f3ef9e80 =   -202400128
0x7ffff3ef9e7c:       30 =           48
0x7ffff3ef9e78: 5c26b397 =   1546040215
0x7ffff3ef9e74:  54c5638 =     88888888
0x7ffff3ef9e70:  4a2cb71 =     77777777
0x7ffff3ef9e6c:       28 =           40
End of: print_stack()
fun ends
main() ends
```