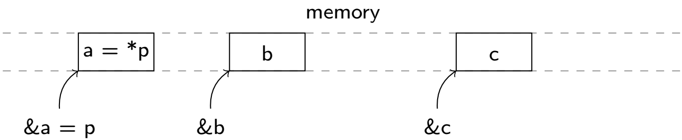
Informatics 3.
Lecture 3: Dynamic memory handling

Kristóf Kovács

Budapest University of Technology and Economics

2024-03-05

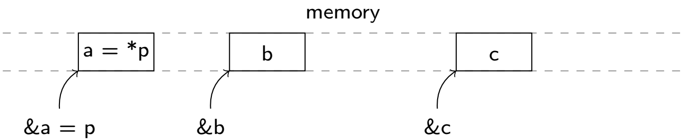
- We can directly reference parts of the memory with pointers:




```c
int a, b, c;
int *p = &a;
*p = 5
```

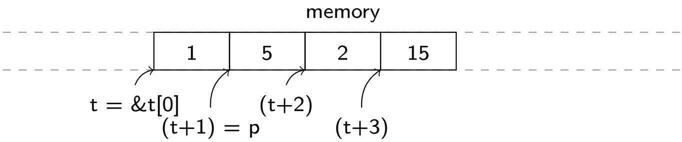
Previously

- We can directly reference parts of the memory with pointers:




```c
int a, b, c;
int *p = &a;
*p = 5
```

- Every pointer is an array and every array is a pointer:




```c
int t[] = {1, 5, 2, 15};
int *p = t + 1;
int x = *(t + 1); // 5
```

Dynamic memory allocation

- From this point onward we're writing C++ code.

Dynamic memory allocation

- From this point onward we're writing C++ code.
- We can manually allocate memory with the **new** keyword. This returns a pointer that points to the newly allocated memory:

```
int *p = new int;
*p = 5;
cout << "pointer: " << p << " mem: " << *p << endl;
// pointer: 0xa000004d0 mem: 5
```

Dynamic memory allocation

- From this point onward we're writing C++ code.
- We can manually allocate memory with the **new** keyword. This returns a pointer that points to the newly allocated memory:

```
int *p = new int;
*p = 5;
cout << "pointer: " << p << " mem: " << *p << endl;
// pointer: 0xa000004d0 mem: 5
```

- The above uses the C++ style output. We will no longer use **stdio.h**, **printf** or **scanf** from now on.

Dynamic memory allocation

- From this point onward we're writing C++ code.
- We can manually allocate memory with the **new** keyword. This returns a pointer that points to the newly allocated memory:

```
int *p = new int;
*p = 5;
cout << "pointer: " << p << " mem: " << *p << endl;
// pointer: 0xa000004d0 mem: 5
```

- The above uses the C++ style output. We will no longer use **stdio.h**, **printf** or **scanf** from now on.
- There is dynamic memory handling in C as well. However it is more complicated so we'll stick to C++ from now on.

C++ compilation

- Almost everything works the same way.

C++ compilation

- Almost everything works the same way.
- Those who work in editors/IDEs (Codeblocks, Visual Studio, etc.) should use the **.cpp** file extension instead of **.c** from now on.

C++ compilation

- Almost everything works the same way.
- Those who work in editors/IDEs (Codeblocks, Visual Studio, etc.) should use the **.cpp** file extension instead of **.c** from now on.
- Those who work in a command line should use **g++** instead of **gcc**.

iostream

- Let's sidetrack a bit to check out the new input/output.

iostream

- Let's sidetrack a bit to check out the new input/output.
- Two easy to remember keywords: **cin** és **cout**.

iostream

- Let's sidetrack a bit to check out the new input/output.
- Two easy to remember keywords: **cin** és **cout**.
- We'll replace **scanf** with **cin**:

```
int a;
float f;
cin >> a;
cin >> f;
```

iostream

- Let's sidetrack a bit to check out the new input/output.
- Two easy to remember keywords: **cin** és **cout**.
- We'll replace **scanf** with **cin**:

```
int a;
float f;
cin >> a;
cin >> f;
```

- We don't have to specify the type of the variable with this.

iostream

- Let's sidetrack a bit to check out the new input/output.
- Two easy to remember keywords: **cin** és **cout**.
- We'll replace **scanf** with **cin**:

```
int a;
float f;
cin >> a;
cin >> f;
```

- We don't have to specify the type of the variable with this.
- The output, **cout** doesn't require output types either and it can be "chained":

```
int a = 5;
float f = 6.4;
cout << "a: " << a << endl << "f: " << f << endl;
```

iostream

- Let's sidetrack a bit to check out the new input/output.
- Two easy to remember keywords: **cin** és **cout**.
- We'll replace **scanf** with **cin**:

```
int a;
float f;
cin >> a;
cin >> f;
```

- We don't have to specify the type of the variable with this.
- The output, **cout** doesn't require output types either and it can be "chained":

```
int a = 5;
float f = 6.4;
cout << "a: " << a << endl << "f: " << f << endl;
```

- **endl** is a new line (end line).

iostream

- **cin** and **cout** are defined in the **iostream** library. From here on we'll use that instead of the **stdio.h** library.

iostream

- **cin** and **cout** are defined in the **iostream** library. From here on we'll use that instead of the **stdio.h** library.
- The C++ libraries have no file extensions. This is how they're differentiated from C libraries (where .h is common).

iostream

- cin and cout are defined in the iostream library. From here on we'll use that instead of the stdio.h library.
- The C++ libraries have no file extensions. This is how they're differentiated from C libraries (where .h is common).
- A complete program showing cin and cout:

```
#include <iostream>
using namespace std;
int main(void) {
  float x;
  cin >> x;
  cout << "value of x: " << x << endl;
  return 0;
}
```

iostream

- cin and cout are defined in the iostream library. From here on we'll use that instead of the stdio.h library.
- The C++ libraries have no file extensions. This is how they're differentiated from C libraries (where .h is common).
- A complete program showing cin and cout:

```
#include <iostream>
using namespace std;
int main(void) {
  float x;
  cin >> x;
  cout << "value of x: " << x << endl;
  return 0;
}
```

- The using namespace std line will come up later. For now let's just copy and paste it after iostream always.

Dynamically allocated array

- We can manually allocate an array as well:

```
int *t;
t = new int[3];
t[0] = 5; t[1] = 6; t[2] = 7;
```

Dynamically allocated array

- We can manually allocate an array as well:

```
int *t;
t = new int[3];
t[0] = 5; t[1] = 6; t[2] = 7;
```

- The elements of a dynamically allocated array are still next to each other in the memory.

Dynamically allocated array

- We can manually allocate an array as well:

```
int *t;
t = new int[3];
t[0] = 5; t[1] = 6; t[2] = 7;
```

- The elements of a dynamically allocated array are still next to each other in the memory.
- Now we can create an array with the length specified by the user:

```
int *t;
int n;
cin >> n;
t = new int[n];
for(int i = 0; i < n; i++) {
  t[i] = i * i;
}
```

Dynamically allocated array

- We can manually allocate an array as well:

```
int *t;
t = new int[3];
t[0] = 5; t[1] = 6; t[2] = 7;
```

- The elements of a dynamically allocated array are still next to each other in the memory.
- Now we can create an array with the length specified by the user:

```
int *t;
int n;
cin >> n;
t = new int[n];
for(int i = 0; i < n; i++) {
  t[i] = i * i;
}
```

- New feature: we can define the cycle variable in **for**.

Returning an array

- Let's try to return an array in a function.

Returning an array

- Let's try to return an array in a function.

```
int* fv() {
  int t[] = {1, 2, 5};
  return t;
}
int main(void) {
  int *a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

- Let's try to return an array in a function.

```
int* fv() {
  int t[] = {1, 2, 5};
  return t;
}
int main(void) {
  int *a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

- This leads to the `Segmentation Fault` error.

- Let's try to return an array in a function.

```
int* fv() {
  int t[] = {1, 2, 5};
  return t;
}
int main(void) {
  int *a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

- This leads to the `Segmentation Fault` error.
- The array **t** is destroyed once the **fv** function ends, like all other local variables.

Returning an array

- Let's try to return an array in a function.

```
int* fv() {
  int t[] = {1, 2, 5};
  return t;
}
int main(void) {
  int *a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

- This leads to the `Segmentation Fault` error.
- The array **t** is destroyed once the **fv** function ends, like all other local variables.
- The true return value of the function is a copy of **t** as a pointer.

Returning an array 2

- Everything dynamically allocated will still exist even outside its scope.

Returning an array 2

- Everything dynamically allocated will still exist even outside its scope.
- This means the following works:

```
int* fv() {
  int *t = new int[3];
  t[0] = 1; t[1] = 2; t[2] = 5;
  return t;
}
int main(void) {
  int *a;
  a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

Returning an array 2

- Everything dynamically allocated will still exist even outside its scope.
- This means the following works:

```
int* fv() {
  int *t = new int[3];
  t[0] = 1; t[1] = 2; t[2] = 5;
  return t;
}
int main(void) {
  int *a;
  a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

- In truth, if something is dynamically allocated it can exist for as long as the program runs.

Returning an array 3

- What's the issue here?

```
int* fv() {
  int *t = new int[3];
  t[0] = 1; t[1] = 2; t[2] = 5;
  int *t_sqr = new int[3];
  for(int i = 0; i < 3; i++) {
    t_sqr[i] = t[i] * t[i];
  }
  return t_sqr;
}
```

Returning an array 3

- What's the issue here?

```
int* fv() {
  int *t = new int[3];
  t[0] = 1; t[1] = 2; t[2] = 5;
  int *t_sqr = new int[3];
  for(int i = 0; i < 3; i++) {
    t_sqr[i] = t[i] * t[i];
  }
  return t_sqr;
}
```

- The dynamically allocated **t** array stays in the memory after the function ends. But we won't have any way to access it.

Returning an array 3

- What's the issue here?

```
int* fv() {
  int *t = new int[3];
  t[0] = 1; t[1] = 2; t[2] = 5;
  int *t_sqr = new int[3];
  for(int i = 0; i < 3; i++) {
    t_sqr[i] = t[i] * t[i];
  }
  return t_sqr;
}
```

- The dynamically allocated **t** array stays in the memory after the function ends. But we won't have any way to access it.
- If we execute the function 100 times, then it will be present in the memory 100 separate times uselessly.

Freeing memory

- Dynamically allocated things have to be manually freed (removed from the memory).

Freeing memory

- Dynamically allocated things have to be manually freed (removed from the memory).
- That's what the **delete** keyword is for:

```
int p* = new int;
*p = 5;
delete p;
```

Freeing memory

- Dynamically allocated things have to be manually freed (removed from the memory).
- That's what the **delete** keyword is for:

```
int p* = new int;
*p = 5;
delete p;
```

- And for dynamically allocated arrays the **delete[]** keyword:

```
int* fv() {
  int *t = new int[3];
  t[0] = 1; t[1] = 2; t[2] = 5;
  int *t_sqr = new int[3];
  for(int i = 0; i < 3; i++) {
    t_sqr[i] = t[i] * t[i];
  }
  delete[] t;
  return t_sqr;
}
```

Freeing memory 2

- Since the function **fv** returns with a dynamically allocated array, it is the responsibility of the caller to free this array (the caller is the **main** fucntion here):

```
int main(void) {
  int *a;
  a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  delete[] a;
  return 0;
}
```

Freeing memory 2

- Since the function **fv** returns with a dynamically allocated array, it is the responsibility of the caller to free this array (the caller is the **main** fucntion here):

```
int main(void) {
  int *a;
  a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  delete[] a;
  return 0;
}
```

- The way we should think about this is that every dynamical allocation has to have its freeing pair.

Advantages of dynamic allocation

- Why are we putting so much effort into understanding the inner workings of the memory, pointers and such?

Advantages of dynamic allocation

- Why are we putting so much effort into understanding the inner workings of the memory, pointers and such?
- What we did so far:

Advantages of dynamic allocation

- Why are we putting so much effort into understanding the inner workings of the memory, pointers and such?
- What we did so far:
  - Using pointers we returned more than 1 value with a function.

Advantages of dynamic allocation

- Why are we putting so much effort into understanding the inner workings of the memory, pointers and such?
- What we did so far:
  - Using pointers we returned more than 1 value with a function.
  - We could handle arrays as pointers (though it's not that useful).

Advantages of dynamic allocation

- Why are we putting so much effort into understanding the inner workings of the memory, pointers and such?
- What we did so far:
  - Using pointers we returned more than 1 value with a function.
  - We could handle arrays as pointers (though it's not that useful).
  - We can use pointers to store the pointer of the dynamically allocated memory.

Advantages of dynamic allocation

- Why are we putting so much effort into understanding the inner workings of the memory, pointers and such?
- What we did so far:
  - Using pointers we returned more than 1 value with a function.
  - We could handle arrays as pointers (though it's not that useful).
  - We can use pointers to store the pointer of the dynamically allocated memory.
  - We can directly return an array with a function.

Advantages of dynamic allocation

- Why are we putting so much effort into understanding the inner workings of the memory, pointers and such?
- What we did so far:
  - Using pointers we returned more than 1 value with a function.
  - We could handle arrays as pointers (though it's not that useful).
  - We can use pointers to store the pointer of the dynamically allocated memory.
  - We can directly return an array with a function.
- Now we'll use pointers to create a new data structure.

The problem

- Develop a data structure the can hold as many elements as we want (or as much as our computer's memory permits).

The problem

- Develop a data structure the can hold as many elements as we want (or as much as our computer's memory permits).
- What we want is a similar data structure as the list in python. The following or something similar should work:

The problem

- Develop a data structure the can hold as many elements as we want (or as much as our computer's memory permits).
- What we want is a similar data structure as the list in python. The following or something similar should work:

```
int x;          // auxiliary variable
list l;         // our new data type
append(l, 5) // we append 5 at the end of ;
append(l, 4) // now 4
cin >> x;
while(x != 0) { // while we do not get a 0
  append(l, x)  // add elements to l
  cin >> x;   // read the next element
}
```

Array expansion idea

First idea:

- Let's use a dynamically allocated array.
- Store the current length of it in a variable
- Should we need more space we can just create a bigger array and copy all elements to this new and larger array. Then we can free the original array.

Array expansion idea

First idea:

- Let's use a dynamically allocated array.
- Store the current length of it in a variable
- Should we need more space we can just create a bigger array and copy all elements to this new and larger array. Then we can free the original array.

Issues:

- It's still limited in size. That's because **int** has a limit, a so called "int max". There would be no way to index the array past this number.
- During every expansion we'll need to do a lot of copying. At one point this will simply take too long.
- At this point we could just as well create an int array the size of "int max" and we would be in the same boat.

Minimalist array expansion implementation (just for show)

(The **x->y** command is equivalent to **(*x).y**. We'll talk about it later.)

```
struct list {
  int *a;
  int n;
  int max;
};

void append(struct list *l, int e) {
  if(l->n >= l->max) {
    int *t = new int[l->max + 100];
    for(int i = 0; i < l->n; i++) {
      t[i] = l->a[i];
    }
    delete[] l->a;
    l->a = t;
    l->max = l->max + 100;
  }
  l->a[l->n] = e;
  l->n++;
}
```

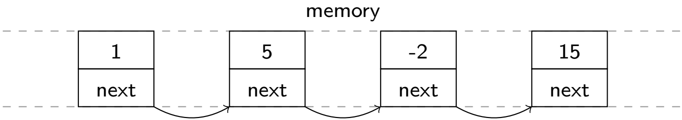
```
int main(void) {
  struct list l;
  l.a = new int[1];
  l.n = 0;
  l.max = 1;
  append(&l, 1);
  append(&l, 5);
  append(&l, -2);
  for(int i = 0; i < l.n; i++) {
    cout << l.a[i] << endl;
  }
  delete[] l.a;
  return 0;
}
```

Linked list idea

- What if every element of our list structure would store the next element's pointer?

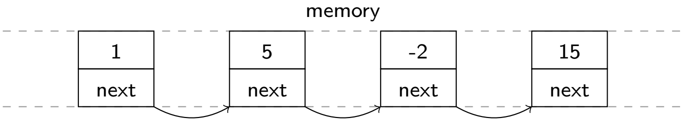
Linked list idea

- What if every element of our list structure would store the next element's pointer?
- Something like this:



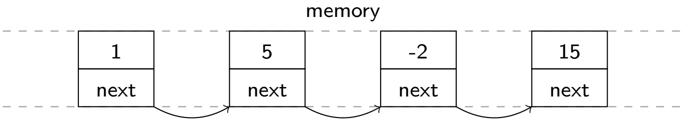
Linked list idea

- What if every element of our list structure would store the next element's pointer?
- Something like this:



- This would store the 1, 5, -2, 15 elements. We would only need to store the pointer to the first element and we could access them all.

- What if every element of our list structure would store the next element's pointer?
- Something like this:

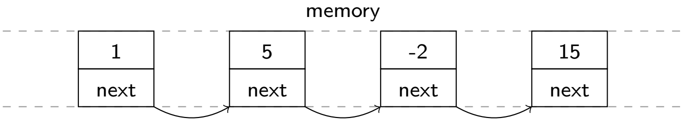


- This would store the 1, 5, -2, 15 elements. We would only need to store the pointer to the first element and we could access them all.
- This is how the elements would look like:

```
struct list_e {
  int num;
  struct list_e *next;
};
```

Linked list idea

- What if every element of our list structure would store the next element's pointer?
- Something like this:



- This would store the 1, 5, -2, 15 elements. We would only need to store the pointer to the first element and we could access them all.
- This is how the elements would look like:

```
struct list_e {
  int num;
  struct list_e *next;
};
```

- It stores a value **num** and the pointer to the next element in **next**.

Implementation of linked lists

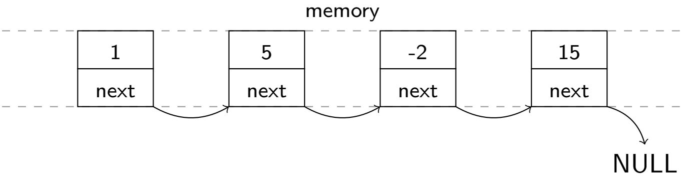
- Let's implement this. But we're instantly faced with a problem. What should the last element's **next** point to?

Implementation of linked lists

- Let's implement this. But we're instantly faced with a problem. What should the last element's **next** point to?
- Let's use the trick from C strings where we had the terminal zero character **'\0'**.

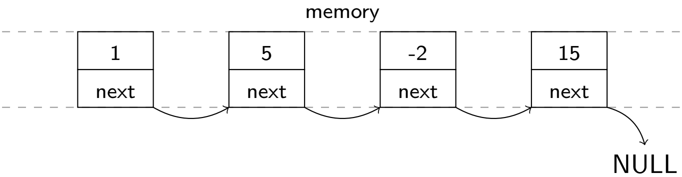
Implementation of linked lists

- Let's implement this. But we're instantly faced with a problem. What should the last element's **next** point to?
- Let's use the trick from C strings where we had the terminal zero character **'\0'**.
- **NULL** is a special pointer value. We can use this to express if something doesn't actually point anywhere:

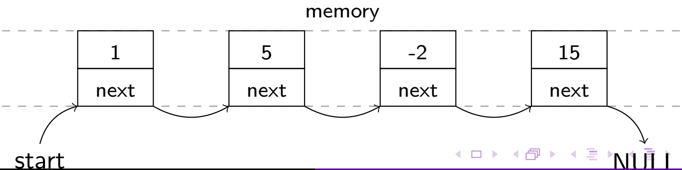


Implementation of linked lists

- Let's implement this. But we're instantly faced with a problem. What should the last element's **next** point to?
- Let's use the trick from C strings where we had the terminal zero character **'\0'**.
- **NULL** is a special pointer value. We can use this to express if something doesn't actually point anywhere:



- Now we just need to store the first element's pointer in a variable and we're basically done:



Implementation of linked lists

- Let's implement this in C++. We want to make this **main** function work:

```
int main(void) {
  struct list_e *start = NULL; // pointer of the first eleme
  append(start, 1);            // add the first element
  append(start, 5);            // then the second
  append(start, -2);           // and a third
  for(;;) {    // we'll have to iterate over this list someh
    // print the elements of the list here
  }
  return 0;
}
```

Implementation of linked lists

- Let's implement this in C++. We want to make this **main** function work:

```
int main(void) {
  struct list_e *start = NULL; // pointer of the first eleme
  append(start, 1);           // add the first element
  append(start, 5);          // then the second
  append(start, -2);        // and a third
  for(;;) {    // we'll have to iterate over this list someh
    // print the elements of the list here
  }
  return 0;
}
```

- Let's start writing the **append** function in small steps.

- Let's implement this in C++. We want to make this **main** function work:

```
int main(void) {
  struct list_e *start = NULL; // pointer of the first eleme
  append(start, 1);            // add the first element
  append(start, 5);           // then the second
  append(start, -2);         // and a third
  for(;;) {    // we'll have to iterate over this list someh
    // print the elements of the list here
  }
  return 0;
}
```

- Let's start writing the **append** function in small steps.
- Since we'll be going step by step there wll be moments when the code contains errors. I'll let you know each time.

Implementation of linked lists

- Let's implement this in C++. We want to make this **main** function work:

```
int main(void) {
  struct list_e *start = NULL; // pointer of the first eleme
  append(start, 1);            // add the first element
  append(start, 5);            // then the second
  append(start, -2);           // and a third
  for(;;) {    // we'll have to iterate over this list someh
    // print the elements of the list here
  }
  return 0;
}
```

- Let's start writing the **append** function in small steps.
- Since we'll be going step by step there wll be moments when the code contains errors. I'll let you know each time.
- Let's use the previously defined **struct list_e** type.

Implementation of linked lists

- The very first thing we need to do is to add the first element. At this point **start** is still **NULL**, since we're not storing anything yet.

```
void append(struct list_e *start, int n) {
  struct list_e *e = new struct list_e;
  (*e).num = n;
  (*e).next = NULL;
  start = e;
}
```

Implementation of linked lists

- The very first thing we need to do is to add the first element. At this point **start** is still **NULL**, since we're not storing anything yet.

```
void append(struct list_e *start, int n) {
  struct list_e *e = new struct list_e;
  (*e).num = n;
  (*e).next = NULL;
  start = e;
}
```

- Dynamically create the element, so that it remains after the function returns.

Implementation of linked lists

- The very first thing we need to do is to add the first element. At this point **start** is still **NULL**, since we're not storing anything yet.

```
void append(struct list_e *start, int n) {
  struct list_e *e = new struct list_e;
  (*e).num = n;
  (*e).next = NULL;
  start = e;
}
```

- Dynamically create the element, so that it remains after the function returns.
- If you try this with only 1 **append** call it will still not be okay

Implementation of linked lists

- The very first thing we need to do is to add the first element. At this point **start** is still **NULL**, since we're not storing anything yet.

```
void append(struct list_e *start, int n) {
  struct list_e *e = new struct list_e;
  (*e).num = n;
  (*e).next = NULL;
  start = e;
}
```

- Dynamically create the element, so that it remains after the function returns.
- If you try this with only 1 **append** call it will still not be okay
- Since we're trying to change the **start** pointer, not its stored stuff in the memory. So we'll have to use a pointer pointer.

Implementation of linked lists

- I only changed the pointer of a pointer part:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  (*e).num = n;
  (*e).next = NULL;
  *start = e; // now we need *start
}
```

Implementation of linked lists

- I only changed the pointer of a pointer part:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  (*e).num = n;
  (*e).next = NULL;
  *start = e; // now we need *start
}
```

- This works. We can add 1 element to the list.

Implementation of linked lists

- I only changed the pointer of a pointer part:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  (*e).num = n;
  (*e).next = NULL;
  *start = e; // now we need *start
}
```

- This works. We can add 1 element to the list.
- However the notation `(*x).y` would quickly become infuriating. Thankfully there's a solution. It means the same as the `x->y` expression.

Implementation of linked lists

- Only replaced with `x->y`

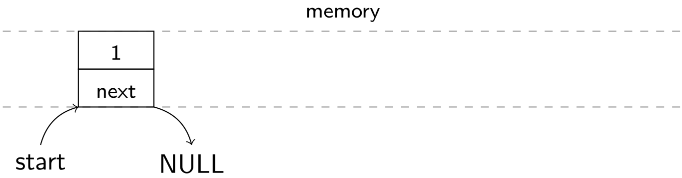
```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  *start = e; // now we need *start
}
```

Implementation of linked lists

- Only replaced with x->y

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  *start = e; // now we need *start
}
```

- This is how the memory looks now:

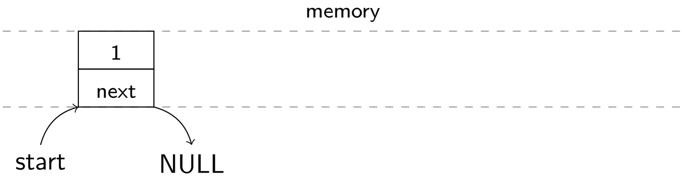


Implementation of linked lists

- Only replaced with `x->y`

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  *start = e; // now we need *start
}
```

- This is how the memory looks now:



- However if we were to call **append** again (to append an element) then it wouldn't work and it's easy to see why. We would only change the first element to this new element.

Implementation of linked lists

- Let's do a condition based on whether we're appending the first element or not:

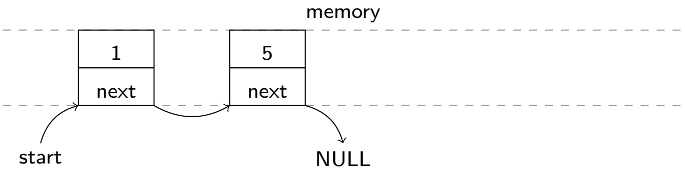
```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e;
  } else {
    (*start)->next = e;
  }
}
```

Implementation of linked lists

- Let's do a condition based on whether we're appending the first element or not:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e;
  } else {
    (*start)->next = e;
  }
}
```

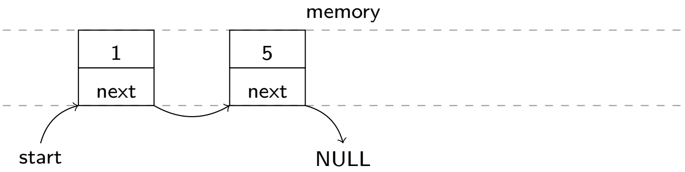
- Now the memory looks like this:



- Let's do a condition based on whether we're appending the first element or not:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e;
  } else {
    (*start)->next = e;
  }
}
```

- Now the memory looks like this:



- It's easy to see again that this still isn't right. When we use **append**-et, we always change the **next** of the **start** element.

Iterating over a linked list

- Let's stop for a moment and think about how we would navigate to the last element. Since that's the element whose **next** we need to set.

Iterating over a linked list

- Let's stop for a moment and think about how we would navigate to the last element. Since that's the element whose **next** we need to set.
- We can borrow from C strings again. We parsed a string there until we encountered the **'\0'** terminal zero. In a linked list the equivalent would be the **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Iterating over a linked list

- Let's stop for a moment and think about how we would navigate to the last element. Since that's the element whose **next** we need to set.
- We can borrow from C strings again. We parsed a string there until we encountered the **'\0'** terminal zero. In a linked list the equivalent would be the **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

- If you think about it this is just a regular **for** loop:

Iterating over a linked list

- Let's stop for a moment and think about how we would navigate to the last element. Since that's the element whose **next** we need to set.
- We can borrow from C strings again. We parsed a string there until we encountered the **'\0'** terminal zero. In a linked list the equivalent would be the **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

- If you think about it this is just a regular **for** loop:
  - We initialize the cycle variable (**e**) with the first element.

Iterating over a linked list

- Let's stop for a moment and think about how we would navigate to the last element. Since that's the element whose **next** we need to set.
- We can borrow from C strings again. We parsed a string there until we encountered the **'\0'** terminal zero. In a linked list the equivalent would be the **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

- If you think about it this is just a regular **for** loop:
  - We initialize the cycle variable (**e**) with the first element.
  - Stopping condition is almost the same as with C strings.

Iterating over a linked list

- Let's stop for a moment and think about how we would navigate to the last element. Since that's the element whose **next** we need to set.
- We can borrow from C strings again. We parsed a string there until we encountered the **'\0'** terminal zero. In a linked list the equivalent would be the **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

- If you think about it this is just a regular **for** loop:
  - We initialize the cycle variable (**e**) with the first element.
  - Stopping condition is almost the same as with C strings.
  - We step onto the next element.

Iterating over a linked list

- Let's stop for a moment and think about how we would navigate to the last element. Since that's the element whose **next** we need to set.
- We can borrow from C strings again. We parsed a string there until we encountered the **'\0'** terminal zero. In a linked list the equivalent would be the **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

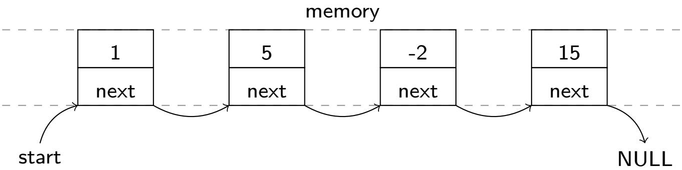
- If you think about it this is just a regular **for** loop:
  - We initialize the cycle variable (**e**) with the first element.
  - Stopping condition is almost the same as with C strings.
  - We step onto the next element.
- We can use this in the **main** function to print the linked list's elements.

Iterating over a linked list

Let's see how this iteration over the linked list would work in our (theoretical) complete implementation of a linked list.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Where we are: before the loop



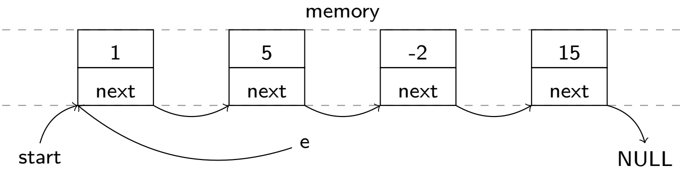
Output:

Iterating over a linked list

Let's see how this iteration over the linked list would work in our (theoretical) complete implementation of a linked list.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Where we are: initialization happened



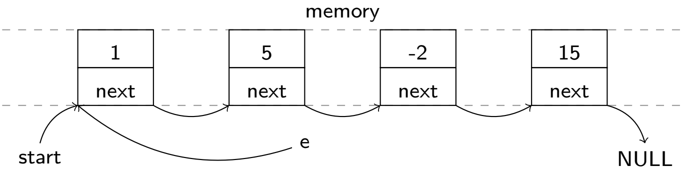
Output:

Iterating over a linked list

Let's see how this iteration over the linked list would work in our (theoretical) complete implementation of a linked list.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Where we are: end of the first loop



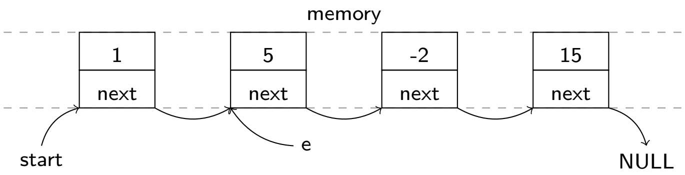
Output:
1

Iterating over a linked list

Let's see how this iteration over the linked list would work in our (theoretical) complete implementation of a linked list.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Where we are: beginning of the second loop



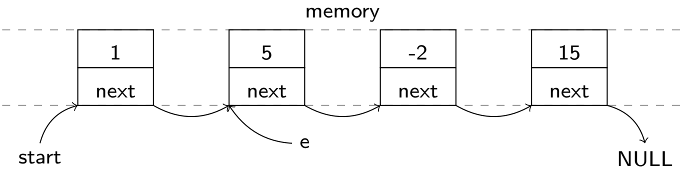
Output:
1

Iterating over a linked list

Let's see how this iteration over the linked list would work in our (theoretical) complete implementation of a linked list.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Where we are: end of the second loop



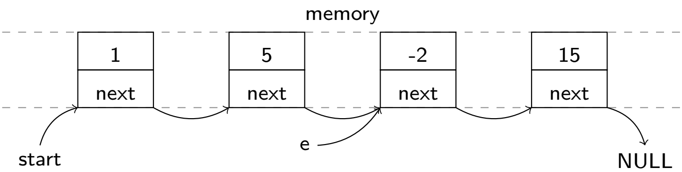
Output:
1
5

Iterating over a linked list

Let's see how this iteration over the linked list would work in our (theoretical) complete implementation of a linked list.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Where we are: end of the third loop



Output:

1

5

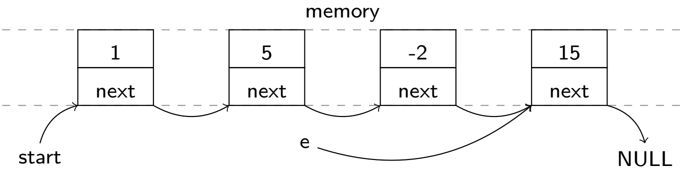
-2

Iterating over a linked list

Let's see how this iteration over the linked list would work in our (theoretical) complete implementation of a linked list.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Where we are: end of the forth loop



Output:

1
5
-2

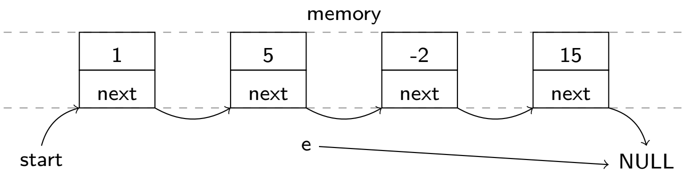
15

Iterating over a linked list

Let's see how this iteration over the linked list would work in our (theoretical) complete implementation of a linked list.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Where we are: end of the forth loop and the step also happened



Output:
1
5
-2

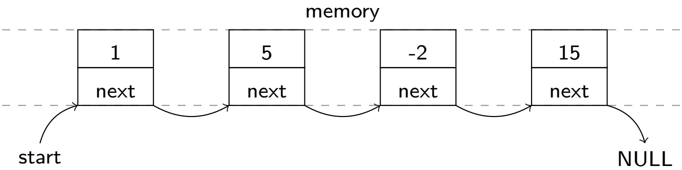
15

Iterating over a linked list

Let's see how this iteration over the linked list would work in our (theoretical) complete implementation of a linked list.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Where we are: end of the loop, the condition was false



Output:
1
5
-2

15

Implementation of linked lists

- Let's go back to the implementation and use what we learned:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e; // most már *start kell
  } else {
    struct list_e *p = NULL;
    for(p = *start; p->next != NULL; p = p->next){}
    p->next = e;
  }
}
```

Implementation of linked lists

- Let's go back to the implementation and use what we learned:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e; // most már *start kell
  } else {
    struct list_e *p = NULL;
    for(p = *start; p->next != NULL; p = p->next){}
    p->next = e;
  }
}
```

- The loop that we used is a bit different to the one we discussed previously.

Implementation of linked lists

- Let's go back to the implementation and use what we learned:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e; // most már *start kell
  } else {
    struct list_e *p = NULL;
    for(p = *start; p->next != NULL; p = p->next){}
    p->next = e;
  }
}
```

- The loop that we used is a bit different to the one we discussed previously.
- Here we want to stop when we've reached the last element.

Implementation of linked lists

- Let's go back to the implementation and use what we learned:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e; // most már *start kell
  } else {
    struct list_e *p = NULL;
    for(p = *start; p->next != NULL; p = p->next){}
    p->next = e;
  }
}
```

- The loop that we used is a bit different to the one we discussed previously.
- Here we want to stop when we've reached the last element.
- We know we're at the last element when the element's **next** is **NULL**.

Implementation of linked lists

- Let's go back to the implementation and use what we learned:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e; // most már *start kell
  } else {
    struct list_e *p = NULL;
    for(p = *start; p->next != NULL; p = p->next){}
    p->next = e;
  }
}
```

- The loop that we used is a bit different to the one we discussed previously.
- Here we want to stop when we've reached the last element.
- We know we're at the last element when the element's **next** is **NULL**.
- There is no command in the inner part of the loop. We only need to set the **p** pointer to the last element.

Implementation of linked lists

- Let's go back to the implementation and use what we learned:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e; // most már *start kell
  } else {
    struct list_e *p = NULL;
    for(p = *start; p->next != NULL; p = p->next){}
    p->next = e;
  }
}
```

- The loop that we used is a bit different to the one we discussed previously.
- Here we want to stop when we've reached the last element.
- We know we're at the last element when the element's **next** is **NULL**.
- There is no command in the inner part of the loop. We only need to set the **p** pointer to the last element.
- This truly implements a dynamically expanding data structure that might even fill the whole memory.

Implementation of linked lists (complete code)

```
#include <iostream>

using namespace std;

struct list_e {
  int num;
  struct list_e *next;
};

void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e;
  } else {
    struct list_e *p = NULL;
    for(p = *start; p->next != NULL; p = p->next){}
    p->next = e;
  }
}
```

```
int main(void) {
  struct list_e *start = NULL;
  append(&start, 1);
  append(&start, 5);
  append(&start, -2);
  append(&start, 15);
  for(struct list_e *e = start; e != NULL; e = e->next) {
    cout << e->num << endl;
  }
  return 0;
}
```

Where to now

Where to next?

Where to now

Where to next?

- We need a way to free the dynamically created linked list.

Where to now

Where to next?

- We need a way to free the dynamically created linked list.
- It would be confusing in the long run if we used **append** and similarly named functions (we would need an append_list, append_dictionary, append_set, etc.). The solution to this will be classes.

Where to now

Where to next?

- We need a way to free the dynamically created linked list.
- It would be confusing in the long run if we used **append** and similarly named functions (we would need an append_list, append_dictionary, append_set, etc.). The solution to this will be **class**es.
- Another issue is that the type of the stored element is set in stone in this implementation (we only stored 1 **int**). The solution to this will be **template**s.

Where to next?

- We need a way to free the dynamically created linked list.
- It would be confusing in the long run if we used **append** and similarly named functions (we would need an append_list, append_dictionary, append_set, etc.). The solution to this will be **class**es.
- Another issue is that the type of the stored element is set in stone in this implementation (we only stored 1 **int**). The solution to this will be **template**s.
- We can't store everything in one file. Once we start using more structures like this we'll need to start using more source files and **header** files.

Where to now

Where to next?

- We need a way to free the dynamically created linked list.
- It would be confusing in the long run if we used **append** and similarly named functions (we would need an append_list, append_dictionary, append_set, etc.). The solution to this will be **class**es.
- Another issue is that the type of the stored element is set in stone in this implementation (we only stored 1 **int**). The solution to this will be **template**s.
- We can't store everything in one file. Once we start using more structures like this we'll need to start using more source files and **header** files.
- We can't implement everything ourselves. We'll start using already implemented data structure from libraries at some point.

Pop quiz questions

- Draw a schematic representation of a linked list in memory.
- What is the **NULL** pointer?
- Show an example of dynamically creating an array.
- What is **delete** and **delete[]** used for?