Memory

Raul P. Pelaez

November 27, 2024

Contents

1 Memory, addresses, and pointers **1**
1.1 Key new concepts . . . 4

2 The null pointer **4**

3 Passing by value or by reference **5**
3.1 Key new concepts . . . 6

4 Const-ness **6**
4.1 Key new concepts . . . 8

5 The stack and the heap **8**
5.1 Key new concepts . . . 9

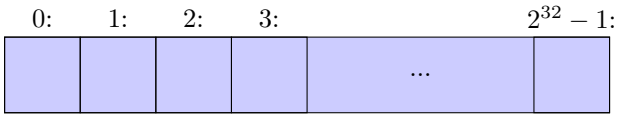
6 New and delete **9**
6.1 Key new concepts . . . 10

7 Casting **11**
7.1 Key new concepts . . . 11

8 Exercises **12**
8.1 Pointer shenanigans . . . 12
8.2 Doubly linked List . . . 16

1 Memory, addresses, and pointers

A computer's memory is a sequence of bytes. We can number the bytes from 0 to the last one (if your computer has 4GB of memory, the last byte will be number $2^{32} - 1$). We call each of these numbers an address. In C++ we have direct access to these addresses by requesting chunks of it from the operative system.
We can represent the memory as a sequence of boxes, each box representing a byte. For example, the memory of a computer with 4GB of memory would look like this:



Everything we put in memory has an address. For example:

Source code

```
int x = 5;
```

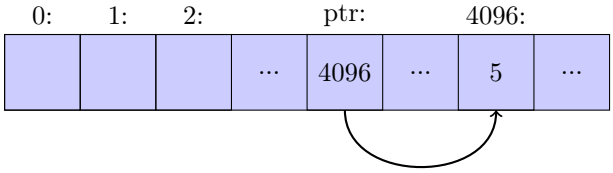
This will reserve an "int-sized" piece of memory for var somewhere and store the value 5 in it. We can access this memory by using the variable name x. We can also store and manipulate addresses. An object that holds an address is called a pointer. For example, the type needed to store the address of an

int is called a "pointer to int" and the notation is `int*`.

Source code

```
int* ptr = &x;
//int *ptr = &x; //This is equivalent, but you should think about the
↪  construction "int*" as a type name.
```

Prepending an object's name with `&` reads as "the address of this object".



`ptr` (which lives in some address in memory itself) stores the address of `x`. We can use this address (4096) to access the value stored in `x` (5).

We can access the value stored in an address by prepending the address with `*`. For example:

Source code

```
int y = *ptr;
```

This is called "dereferencing" an address/pointer.

Advice

As you see, the different operators/keywords of the language have wildly different meanings depending on context (which can be really subtle). This happens a lot in C++, so beware!

Finally, we can also store references to variables (another use of the "address of" `&` operator). Like this

Source code

```
int original = 5;
int &alias = original; // Reads as "alias is a reference to original"
alias = 10;
std::cout << original << std::endl; // This will print 10
```

Info

`auto` can be used alongside references and pointers. For instance, `auto*` is a pointer to whatever is needed, and `auto&` is a reference to whatever is needed.
Why would you need to write `auto* ptr = &x;` instead of `auto ptr = &x;`? Well, it might make little sense here, but see this example in a range-based loop:

Source code

```
std::vector<int> vec = {1, 2, 3, 4, 5};
// We can modify the elements of vec by decorating auto with a reference
↪  here
// Without & auto will be deduced to just "int" and we will be modifying
↪  a copy of the elements, thus the original vector would remain
↪  unchanged
for (auto& x : vec) {
  x *= 2;
 }
```

Info

Dereferencing and the `[]` operator
If you have an array, such as the one we created with `int arr[10];` or the underlying memory of a vector, you can access the elements of the array by using the `[]` operator.

Source code

```
int arr[10];
arr[7] = 5;
std::cout << *(&arr[0]+7) << std::endl; // This will print 5
// This is equivalent to arr[7]
```

Advanced

Does pointer arithmetics imply that we can access and modify the value stored in any address of the RAM?
Yes! unless that address is protected by the operative system, which includes sensitive regions of the memory (like the kernel's memory) and memory reserved by other programs.
Tricking the OS into letting you inspect forbidden addresses is a common way to exploit vulnerabilities in software.

Calling methods on pointers
Remember in C++ types (classes) can have methods, such as `std::vector<int>::push_back` or `std::string::size`. We normally use the `.` operator to call these methods. But what if we have a pointer to an object? We can still call methods on it by using the `->` operator. For example:

Source code

```
void print_string_size(const std::string* str) {
  std::cout << str->size() << std::endl;
  // This is equivalent to (*str).size()
}
```

1.1 Key new concepts

Info

The memory model
The computer memory is a sequence of bytes, each byte has an address.

Info

Pointers
A pointer is a variable that stores an address.

Info

References
A reference is an alias to a variable.

Info

The `&` and `*` operators
The `&` symbol returns the address of a variable (a pointer to it) or denotes a reference when decorating a type.
The `*` symbol returns the value stored in an address (derreferences) or denotes a pointer when decorating a type.

Info

The `->` operator
This operator is used to call methods on pointers to types.
Writing `ptr->method()` is equivalent to writing `(*ptr).method()`.

2 The null pointer

There is a special address reserved for the meaning "no address". This is called the null pointer. In C we would call this `NULL`, but in C++ we should use `nullptr`. For example:

Source code

```
void foo(int* ptr) {
  // It is a good practice to check for nullptr when a function receives a
  ↪  pointer
  if (ptr == nullptr) {
  //if(!ptr){ // nullptr casts to false automatically
    std::cout << "The pointer is null" << std::endl;
    exit(1);
  }
}
```

Advice

Checking for the `nullptr` is one reason why you should always prefer passing by reference over passing by pointer in a function.

3 Passing by value or by reference

This is such an important concept I am giving it a separate section.
See these three versions of a function:

Source code

```
void increment_val(int x/*This is called passing by value*/) {
  x++;
}
void increment_ref(int& x/*This is called passing by reference*/) {
  x++;
}
void increment_ptr(int* x) {
  // This version is more bug prone and less readable than the previous one
  // Always prefer passing by reference over passing by pointer
  if(x == nullptr) {
    std::cout << "The pointer is null" << std::endl;
    exit(1);
  }
  (*x)++;
}
int main(){
      int x = 5;
      increment_val(x);
      std::cout << x << std::endl; // This will print 5, we incremented a copy
      ↪  of x
      increment_ref(x);  // Note that there is an "automatic conversion" here
      ↪  from the value x to "a reference to x"
      std::cout << x << std::endl; // This will print 6, we incremented the
      ↪  original x by sending a reference to it
      // The following has the same effect as the previous function, but with a
      ↪  noisier and unsafer syntax
      increment_ptr(&x);
      std::cout << x << std::endl; // This will print 7, we incremented the
      ↪  original x by sending a pointer to it

}
```

We can have a function that receives a copy of the variable (passing by value), a function that receives a reference to the variable (passing by reference), and a function that receives a pointer to the variable. The last two versions can be used to modify the original variable. The first version can't.
One has to be careful when designing an interface that receives a reference or pointer with the intention of modifying the original variable. The fact that a variable will be modified might not be obvious to the reader.

Advice

When to pass by value, reference or pointer?
It depends, but as a general rule:

- Prefer passing-by-value for "tiny" objects (like ints, floats, chars, etc).
- For functions where "no object" (represented by `nullptr`) is a valid input, prefer passing by pointer.
- For everything else, prefer passing by reference.

3.1 Key new concepts

Info

Passing by value
The function receives a copy of the variable.

Info

Passing by reference
The function receives a reference to the variable.

4 Const-ness

There are two levels of const-ness in C++: `const` and `constexpr`. The first one means that the value or object can't be modified. The second one means that the value or object is constant at compile time.
Const
Sometimes we want a value or object that is constant, meaning that we can't modify it. We can declare a variable as constant by using the `const` keyword.
Example

Source code

```
const int x = 5;
//x = 10; // This would give a compilation error
int y = 10;
const int* ptr_y = &y; // This is a pointer to a constant int
// *ptr_y = 5; // This would give a compilation error
const int& ref_y = y; // This is a reference to a constant int
// ref_y = 5; // This would give a compilation error
```

Advice

Why would you use a const reference?
Passing an integer by value to a function is not a problem, but what if a function needs to process a vector with 10 million elements? Passing by value would require copying this vector, which is really expensive (and maybe we do not even have enough RAM).
OTOH we might want to make sure that the function does not modify the vector. We can pass a constant reference to a vector to the function, and the function will not be able to modify it:

Source code

```
void process(const std::vector<int>& vec) {
  // vec.push_back(5); // This would give a compilation error
  // vec[0] = 5; // This would also give a compilation error
  for(auto x : vec) {
    do_something_with(x);
  }
}
```

Constexpr

The `constexpr` keyword is used to declare that a value is constant at compile time. The consequences of this are quite profound. For instance, if we pass a `constexpr` variable to a `constexpr` function, the function will be evaluated at compile time, costing 0 runtime. For example:

Source code

```
#include <iostream>
constexpr int factorial(int n) {
  return n == 0 ? 1 : n * factorial(n-1);
}
int main() {
  constexpr int x = 5;
  constexpr int y = factorial(x); // This will be evaluated at compile time
  std::cout<<y;
}
```

Advanced

Try to compile this code using Godbolt and see the assembly code generated. You will see that the factorial function is not even present in the assembly code, nor the variables, for that matter. The program simply compiles to "print 120".

Info

`constexpr` is a safer alternative to a preprocessor macro (`#define`) and is recommended in modern C++.

Source code

```
#define PI 3.14159265359
constexpr double pi = 3.14159265359; //Always prefer this
```

`constexpr` allows us to use the full power of the C++ language to define constants (type safety, scope,...) , while preprocessor macros are just text substitutions.

4.1 Key new concepts

Info

The `const` **keyword**
A variable declared as `const` can't be modified. This includes references, pointers, etc.
Always mark variables as `const` if they are not meant to be modified.

Info

The `constexpr` keyword
`constexpr` roughly means "to be evaluated at compile time". It can be used to define constants and functions. Always prefer this to preprocessor macros.
Always mark variables as `constexpr` when possible.

5 The stack and the heap

The most relevant memory regions in a C++ program are the stack and the heap.

- The stack is a region set aside by the compiler to store things that are "known at compile time", for instance the local variables of a function. The stack is fast and has a limited size.

- The heap (also known as free store or dynamic memory) consists of the rest of the memory that the OS gives us access to (the RAM). We can request memory from the heap at runtime using the `new` keyword. The heap is slower than the stack and has a much larger size.

  Allocating on the stack: `std::array`
  Besides local variables, we can also store things in the stack by using the `std::array` class. This is a fixed-size array that is allocated in the stack.
  Allocating on the heap: `std::vector`
  We request memory from the heap at runtime using the `std::vector` class. This is a dynamic array that can be resized at runtime.
  Example

Source code

```
void foo(){
  int stack_var; // This is a local variable, it is stored in the stack
  std::array<int, 10> cpp_arr; // This is an array of 10 ints, it is stored in
  ↪  the stack
  //cpp_arr cannot be resized, and its size must be known at compile time
  cpp_arr[0] = 5;
  for(auto x : cpp_arr) {
    std::cout << x << std::endl;
  }
  //std::array is a safer version of the C-style array:
  //int c_arr[10]; // This is an array of 10 ints, it is stored in the stack
  // Regarding the stack, there is no difference between std::array and C-style
  ↪  arrays
  std::vector<int> vec; // This is a dynamic array, it is stored in the heap
  vec.resize(100); // We can resize it at runtime
}
```

Advice

Always prefer using `std::array` over C-style arrays. It stores the size of the array and does not cast to a pointer automatically, which prevents a lot of bugs and makes the code more readable.

5.1 Key new concepts

Info

The stack
A region of memory set aside by the compiler to store local variables, which have size known at compile-time.
The stack is fast and has a limited size.
`std::array` is a fixed-size array that is allocated in the stack.

Info

The heap
A region of memory that we can request from the operative system at runtime.
The heap is slower than the stack and has a much larger size.
`std::vector` is a dynamic array that is allocated in the heap.

6 New and delete

In C++ we can request heap memory from the operative system by using the `new` keyword. This will return a pointer to the memory granted to us by the system. For example:

Source code

```
int* ptr = new int;
```

This will request an int-sized piece of memory and return a pointer to it. We can also request arrays of memory like this:

Source code

```
int* arr = new int[10];
```

This will request 10 int-sized pieces of memory and return a pointer to the first one. We can access the elements of the array by using the `[]` operator. For example:

Source code

```
arr[0] = 5;
arr[1] = 10;
std::cout << arr[0] << std::endl; // This will print 5
```

When requesting memory using `new` we must remember to release it when we are done using it. We do this by using the `delete` keyword. For example:

Source code

```
delete ptr; // Frees the memory for a single value
delete[] arr; // Frees the memory for an array
```

Internally, `std::vector` uses `new` and `delete` to manage memory.

Warning

You should never use malloc and free in C++ code. Always use new and delete if you really have to.
We will see much better ways to handle memory in the future such that we will never have to use new and delete either.
In C++ we have tools that will handle memory allocation and deallocation for us, like smart pointers, containers and RAII in custom classes. We will learn about them in the future.

Warning

You should almost never use new and delete. They are error-prone and can lead to memory leaks and segmentation faults. Always prefer using containers like `std::vector` or `std::array` over raw pointers.

6.1 Key new concepts

Info

The `new` and `delete` keywords
The `new` keyword requests heap memory and returns a pointer to it.
The `delete` keyword releases memory back to the operative system.
You should never use `malloc` and `free` in C++ code.
You should almost never use `new` and `delete` in C++ code. Prefer `std::vector` instead.

7 Casting

Sometimes we need to convert between types. For example, we might need to convert an int to a float. This is called casting. There are several types of casting in C++:

`static_cast`

Used to convert between types that are "compatible" (like int to float, or int to double) in a safe way.

Example

Source code

```
int x = 5;
float y = static_cast<float>(x);
// This is equivalent to float y = (float)x;
// But safer, because static_cast will give a compilation error if the
↪  conversion is not safe
```

`reinterpret_cast`

In some very specific and rare circumstances we might need to convert between types that are not compatible. This is called reinterpret casting. An integer is represented by 32 bits of information, same as a float. Given that we have full control over the memory, we can reinterpret an integer as a float. This is a dangerous cast and should be avoided.

Example

Source code

```
int a = 1065353216;
float *a_as_float_ptr = reinterpret_cast<float*>(&a);
float a_as_float = *a_as_float_ptr;
std::cout<<"a: "<< a << std::endl; // This will print 1065353216
std::cout << "a interpreted as float:
↪  "<<std::fixed<<std::setprecision(3)<<a_as_float<<std::endl; // This will
↪  print 1.000
```

Advanced

`dynamic_cast`

This is used to convert between types in a polymorphic class hierarchy. We will come back to it when we learn about classes.

7.1 Key new concepts

Info

`static_cast`

This is the most common cast in C++. It is used to convert between types that are "compatible" (like int to float, or int to double) in a safe way.

Info

`reinterpret_cast`

This is used to convert between types that are not compatible. This is a dangerous cast and should be avoided.

8 Exercises

8.1 Pointer shenanigans

Goal

Lets do some exercises to understand the memory layout of C++ programs and the use of pointers.
These examples show the importance of understanding how pointers work and the use of abstractions such as classes and vectors.
In general, we should try to use the highest level abstraction available to us (like using `std::vector` instead of raw pointers) to avoid bugs and make our code more readable.
One cornerstone of C++ is "zero cost abstractions", which means that using a higher level abstraction should not have a performance penalty.

Milestone

Memory layouts
Use the `sizeof` operator to determine the size of some types.
What happens when you print the `sizeof` of a vector?

Source code

```
std::vector<int> v(1000); // A vector with 1000 elements
std::cout<<sizeof(v)<<std::endl;
```

Do the same for an `std::array` with 1000 elements.
Can you explain what you see?

Milestone

Pointer arithmetic 1/2
Write three versions of a function that will sum the elements in a vector:

1. A function that takes a vector by value.
2. A function that takes a vector by reference.
3. A function that takes a pointer to the first element of the vector.

What happens if I call these functions like this:

Source code

```
constexpr int size = 1000;
std::vector<int> v(size); // A vector with 1000 elements
std::iota(v.begin(), v.end(), 0); // Fill the vector with 0, 1, 2, 3, ...
std::cout<<sum_val(v)<<std::endl;
std::cout<<sum_ref(v)<<std::endl;
auto* ptr = v.data();
// auto* ptr = &v[0]; // This is equivalent
v.clear();
std::cout<<sum_val(v)<<std::endl;
std::cout<<sum_ptr(ptr, size)<<std::endl;
```

Can you explain what happens?

Milestone

Pointer arithmetic 2/2
Change the order of these lines in the previous milestone:

Source code

```
auto* ptr = v.data();
v.clear();
```

Does your `sum_ptr` function still work? Modify the function so that it does.

Hint

- Check the documentation for the `std::vector::clear` function.
- Taking pointers as arguments requires to painstakingly check for `nullptr`.

Milestone

Casting
Define an integer variable.
Convert this variable to `float` using:

1. A C-style cast.
2. A `static_cast`.
3. A `reinterpret_cast`.

Whats the difference between these casts?
Make it so that the resulting `float` stores the value "2.0" when reinterpreting the integer.

Hint

- `reinterpret_cast` is used to convert pointers to unrelated types.
- `static_cast` is used to convert between types that are "convertible".

Advanced milestone

Casting
What happens if you `reinterpret_cast` a pointer to an integer to a pointer to a double?
Try to print the value of the double, run the code several times in a row.
Can you explain what happens?

Advanced milestone

Privilege escalation
While only being allowed to add code inside the inject function, make it so that the root message is printed

Source code

```
#include <iostream>
#include <string>

void root_privileged_print(std::string message, auto user){
  if (user==0xbada55){
    std::cout<<"Root: "<<message<<std::endl;
  }
  if(user == 0xdeadbeef){
    std::cout<<"Users do not have the rights to print"<<std::endl;
  }
}

void injection(){

}

int main(){
  auto user = 0xdeadbeef;
  injection();
  root_privileged_print("ALL YOUR BASE ARE BELONG TO US", user);
  return 0;
}
```

This is a hard, lateral-thinking, problem and you might not even be able to get it working in your machine due to the compiler optimizations.
The resulting code is absolutely terrible and should never be used in a real program. It will unavoidably have to break the rules of the language and rely on undefined behavior. That being an option is both the power and the danger of C++.
Many times, even with the intended solution, the program will crash.

Can you explain why it crashes sometimes?

Hint

- Disable compiler optimizations by using the -O0 flag.
- You will need to abuse undefined behavior and the stack layout to solve it.
- The final program will work sometimes even with optimizations enabled if you decorate `user` with the `volatile` keyword.

8.2 Doubly linked List

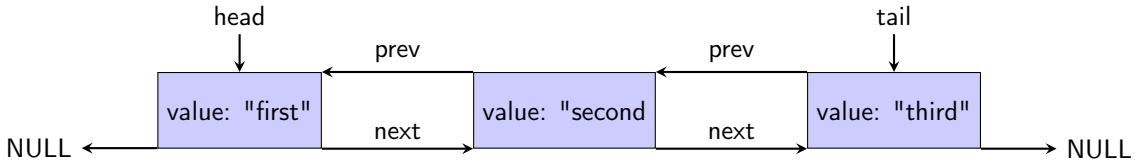
Goal

Implement a doubly linked list in C++
A doubly linked list is a data structure that consists of a sequence of elements where each element contains a data field and two link fields: one that points to the next element in the sequence (next pointer) and another that points to the previous element (previous pointer). This bidirectional linking allows for efficient traversal in both directions.
It is a very important data structure in programming, and it is used in many algorithms and libraries. For instance the C++ standard library uses doubly linked lists to implement the list container, a memory allocator (like the logic behind malloc) could also use a doubly linked list to keep track of free memory blocks.

Warning

This is a very common example to showcase pointers, LLMs will be very familiar with it, but they might lie to you in subtle ways because of the myriad of implementations with slightly different requirements they have seen.
Even so, the good ones can probably just write this exercise really easily. Just a word of advise that you might want to write this one yourself entirely.

We can visualize it like this:



Milestone

Create a copy of the templated project we used the previous week with CMake and GTest, or keep using the one you wrote if you prefer.
Add your code for the next milestones in homework/doubly_linked_list.h (there are no binaries for this file).
Add your tests to test/test_doubly_linked_list.cpp, make sure to include the necessary headers and modify the CMakeLists.txt file accordingly.

Milestone

Create a structure that represents a node in a doubly linked list. It should have a value field and two pointers, one to the next node and one to the previous node.
Make the value a string.
Write a function that creates a new node with a given value.

Hint

- The previous and next pointers should be of the same type as the structure itself.
- You can use the `new` operator to allocate memory for a node.
- Remember to always initialize variables.

Advice

Writing a class is a much cleaner and safer way to implement a container like this, but we have not learned about classes yet, so for the moment we can implement this using free standing functions.
When we learn about classes, we will refactor this code to use them, which will help us understand the benefits of classes.

Milestone

Write a function that inserts a new node before a given node. Its signature should be:

Source code

```
/**
* @brief Insert a node before a given node. If the given node is nullptr,
↪  insert at the beginning.
* @param p The node before which to insert
* @param n The node to insert
* @return The inserted node
*/
Node* insert(Node* node, Node* new_node);
```

Make sure your code works correctly when either p is null (making n the first and only node) or when n is null (returning).

Milestone

Write a test for insert using GTest.

Hint

You can transform this code into GTest:

Source code

```
void test_insert(){
  NodePtr node = createNode("Apple");
  node = insert(node, createNode("Banana"));
  node = insert(node, createNode("Cherry"));
  NodePtr current = node;
  assert(current->data == "Cherry");
  assert(current->next->data == "Banana");
  assert(current->prev == nullptr);
  current = current->next;
  assert(current->next->data == "Apple");
  assert(current->prev->data == "Cherry");
  current = current->next;
  assert(current->prev->data == "Banana");
  assert(current->next == nullptr);
  // Free all
  while (node != nullptr) {
    NodePtr temp = node;
    node = node->next;
    delete temp;
  }
}
```

You can also compare with `std::list`, the standard doubly linked list implementation:

Source code

```
void test_insert(){
  std::list<std::string> l = {"Apple", "Banana", "Cherry"};
  NodePtr node = createNode("Apple");
  node = insert(node, createNode("Banana"));
  node = insert(node, createNode("Cherry"));
  auto node_l = l.begin(); // Iterator (a fancy pointer) to the beginning
  ↪  of the list
  assert(node->data == *node_l);
  node = node->next;
  assert(node->data == *(node_l++));
  node = node->next;
  assert(node->data == *(node_l++));
  // Free all
  while (node != nullptr) {
    NodePtr temp = node;
    node = node->prev;
    delete temp;
  }
}
```

Milestone

Write a function that deletes a node from the list. Its signature should be:

Source code

```
 /**
* @brief Delete a node from the list
* @param n The node to delete
* @return The next node
*/
 Node* erase(Node* node);
```

Also write a GTest for it.

Milestone

Add a function that adds a node **after** a given one. Its signature should be:

Source code

```
/**
 * @brief Add a node after a given node.
 * @param p The node after which to add
 * @param n The node to add
 * @return The added node
 */
Node* add(Node* p, Node* n);
```

Also add a test for it.

Advanced milestone

Modify the documentation scripts and sources to include the doubly linked list implementation somewhere.

Advanced milestone

Use the tool valgrind to make sure your tests do not have memory leaks. Memory leaks consist of memory that is allocated but never deallocated. Using correct memory management via classes (RAII) and the standard containers when available (like the `std::list` doubly linked list implementation in this case) are the best ways to avoid memory leaks.

Hint

- You can use valgrind like this: valgrind –leak-check=full ./your_program.
  A correct execution will look like this:

Source code

```
==250816== Memcheck, a memory error detector
==250816== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward
↪  et al.
==250816== Using Valgrind-3.23.0 and LibVEX; rerun with -h for
↪  copyright info
==250816== Command: ./a.out
==250816==
==250816==
==250816== HEAP SUMMARY:
==250816==     in use at exit: 0 bytes in 0 blocks
==250816==   total heap usage: 4 allocs, 4 frees, 73,872 bytes
↪  allocated
==250816==
==250816== All heap blocks were freed -- no leaks are possible
==250816==
==250816== For lists of detected and suppressed errors, rerun with:
↪  -s
==250816== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
↪  from 0)
```