

Lecture 10: Addressing References

Bart Iver van Blokland

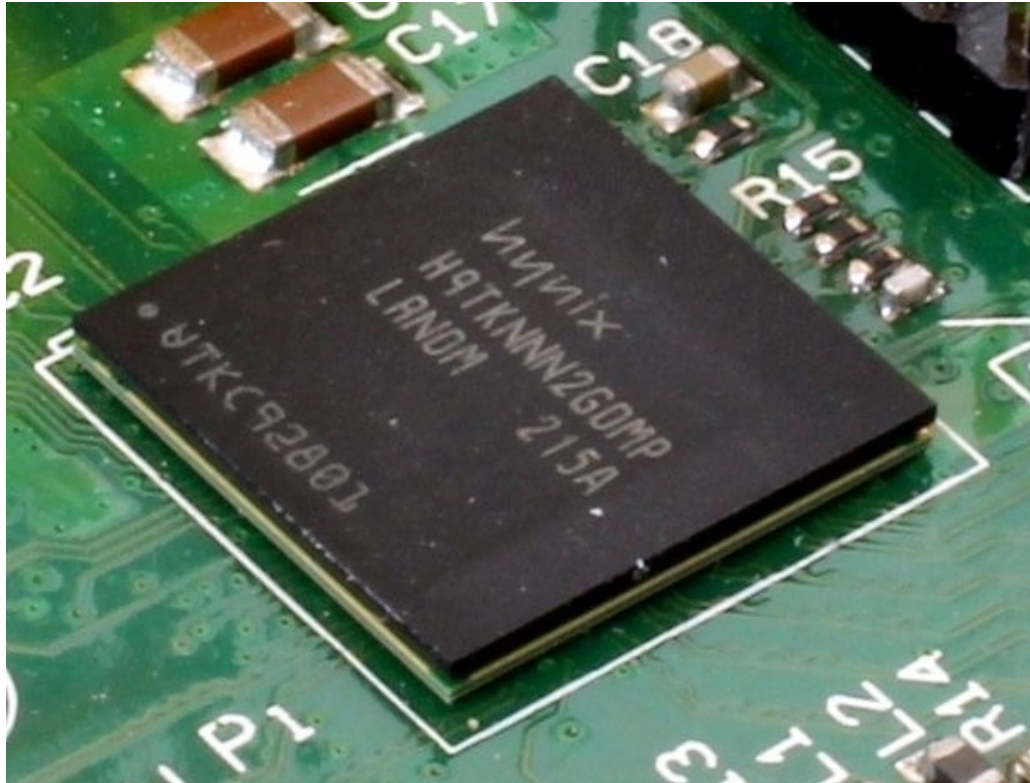
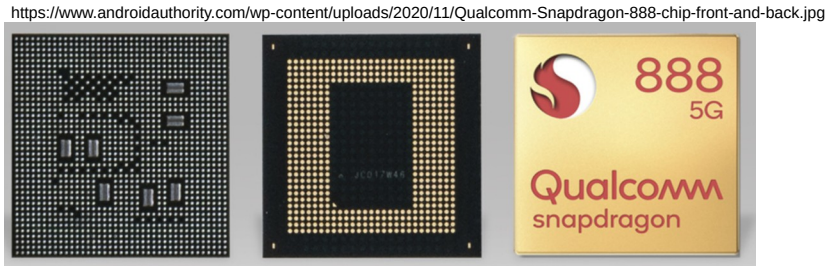
Do you remember?

- What are the main steps for reading a file?
- What is a stringstream, and where can it be useful?
- When overloading an output operator (<<), why should the stream object (std::ostream) be returned from the function?
- When overloading an input operator (>>), why should the value being read in be passed as a reference?

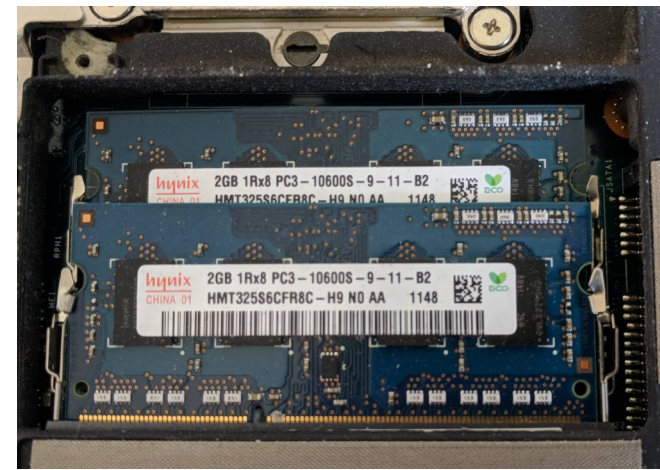
Today (and next week)

Memory

Memory



<https://www.quora.com/Are-RAM-and-storage-physically-located-on-a-smartphones-SoC>



<https://superuser.com/questions/1280480/laptop-ram-compatibility>



<https://www.techpowerup.com/img/wPXFxNdx31ZD8ceT.jpg>

Today

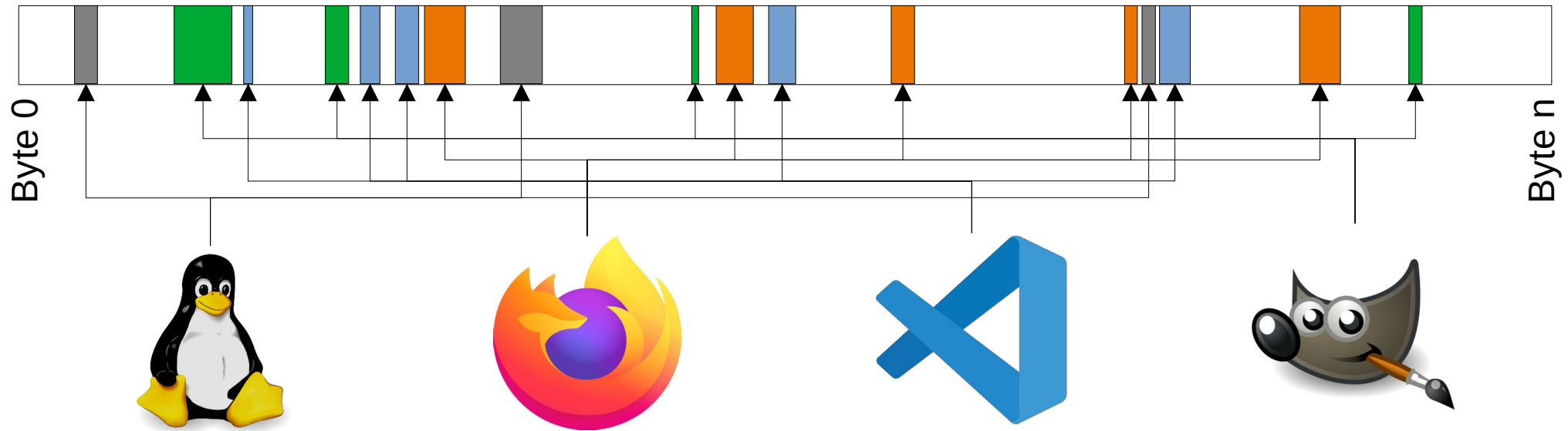
- **Memory Management**
- Pointers
- Scope
- Memory Allocation / Deallocation
- Copy Constructor

Random Access Memory (RAM)

- Programs running on your computer store their variables in RAM
- Inherently temporary storage.
 - Only available as long as program is running
 - Use files for persistent storage

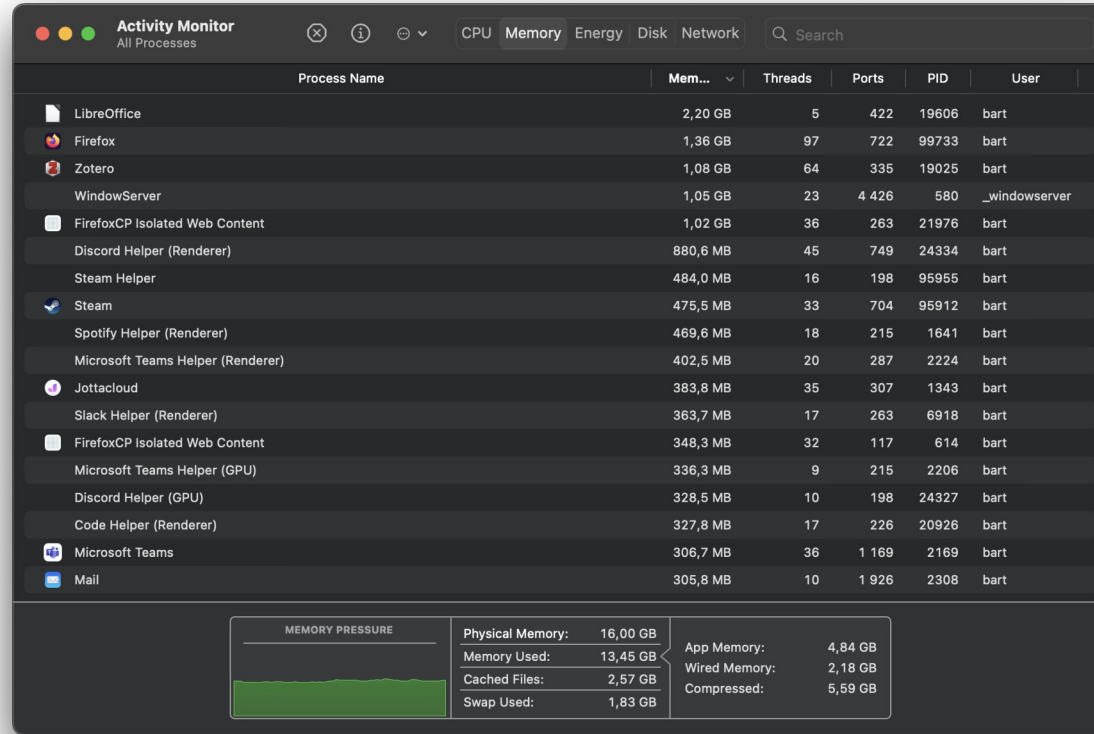
RAM is shared!

- Your computer has a limited amount of memory that must be shared between programs
- The operating system assigns portions of memory to programs as needed



RAM is shared!

You can monitor RAM usage in Task Manager (Windows) or Activity Monitor (MacOS)



Random Access Memory (RAM)

- Allocation: reserve a region of memory to store data (variables)
- Deallocation: return a reserved region of memory to the operating system such that other programs can use it

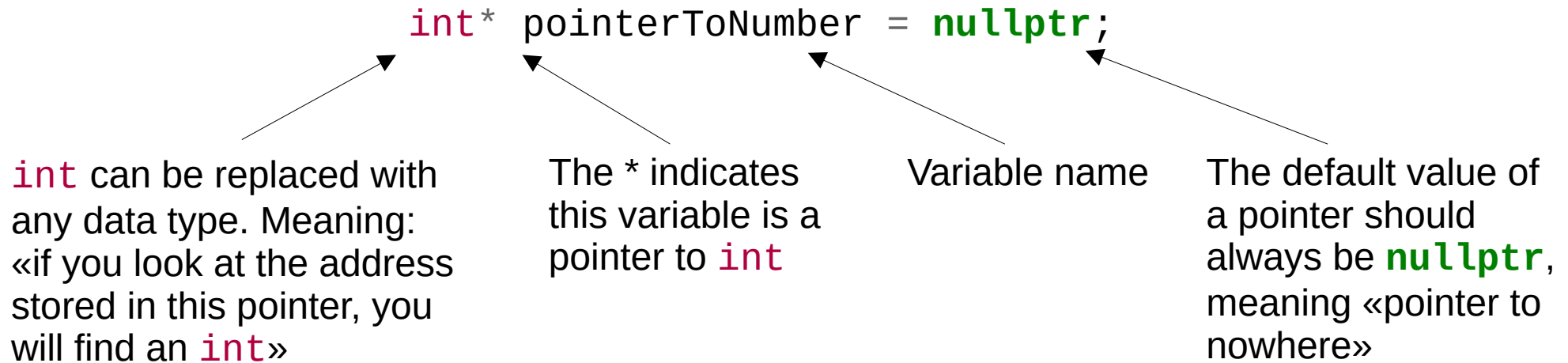
Today

- Memory Management
- **Pointers**
- Scope
- Memory Allocation / Deallocation
- Copy Constructor

Demonstration: Pointers

Pointer

- Definition: A variable which contains the location (memory address) of another variable
- Syntax:



Why pointers?

- Avoid making copies of objects, just like references
- References cannot be stored as a variable in an object or `std::vector`, pointers can
 - Very easy to share references to commonly used objects
- Can create `std::vectors` or `std::arrays` containing different types of objects (lecture 12)
- Required for storing large amounts of data
- Allows for some very efficient algorithms
(we will implement one in the assignment =))

Pointers

- You can obtain the address of any variable with the & operator:

```
int number = 5;  
int* pointerToNumber = &number;
```

- Fetching the value at the address stored in the pointer is done using the * operator:

```
std::cout << *pointerToNumber << std::endl; // prints 5  
int copyOfNumber = *pointerToNumber; // copyOfNumber is now 5
```

- The address the pointer references can be printed using std::cout:

```
// prints a memory address, for example: 0x16fdff308  
std::cout << pointerToNumber << std::endl;
```

Pointers

- We can modify the value the pointer references, but always need to dereference it first:

```
int number = 5;
int* pointerToNumber = &number;

*pointerToNumber += 5;
std::cout << number << std::endl; // prints 10

number = 25;
std::cout << *pointerToNumber << std::endl; // prints 25
```

Pointers are (almost) references!

	Pointers	References
References another value	Yes	Yes
Can reference a value that does not exist	Yes	Yes, but easier to do with a pointer
Can be set to nullptr	If not const	No
Can modify the address being referenced	If not const	No
Possible to create a vector or array containing these	Yes	No
Need to dereference the reference explicitly	Yes	No

Best practice: use references when you have a choice!

Task: pointers

- What do each of these programs print out to the terminal?

```
int a = 10;
int* ptr1 = &a;
int b = 20;
int* ptr2 = &b;
ptr1 = ptr2;
*ptr1 += *ptr2;
std::cout << a << std::endl;
std::cout << b << std::endl;
```

```
float x = 2.5;
float* ptr1 = &x;
float y = 7.0;
float* ptr2 = &y;
float** ptrCeption = &ptr1;
float* ptr3 = ptr2;
ptr1 = ptr2;
ptrCeption = &ptr3;
std::cout << **ptrCeption << std::endl;
```

Pointers to objects

- Using the members of objects referenced by a pointer requires the `->` operator, or first dereferencing the pointer (both methods are entirely equivalent)

```
std::vector<int> integers(10);  
std::vector<int>* pointerToIntegers = &integers;
```

```
pointerToIntegers->at(3) = 8;
```

// These two lines are equivalent:

```
std::cout << (*pointerToIntegers).at(3) << std::endl;  
std::cout << pointerToIntegers->at(3) << std::endl;
```

Today

- Memory Management
- Pointers
- **Scope**
- Memory Allocation / Deallocation
- Copy Constructor

Scope

- Determines where *names* (e.g. variables, classes, or functions) exist, and where they cease to exist

For example:

```
int a = 2;
```

```
if(a == 3) {  
    int b = a;  
}
```

```
int c = a; // no problem!
```

```
int d = b; // error: b does not exist!
```

Scope

- In general:
 - Scopes are denoted using { } braces
 - Names exist from where they are declared within a scope to the end of that scope
 - Names can be used in the scope where they exist


```
1           // i does not exist here
2 {         // The scope of i begins here
3           // i does not exist here
4     int i = 5; // i now exists
5     {
6         // i exists here
7     }
8         // i exists here
9 }         // The scope of i ends here, i ceases to exist
10          // i does not exist here
```

Block Scope

```
if(condition) {  
    // this is a scope  
}  
  
for(int i = 0; i < 10; i++) {  
    // i is part of this scope  
}  
  
while(condition) {  
    // this is a scope  
}  
  
try {  
    // this is a scope  
} catch(std::exception& e) {  
    // this is a separate scope  
}
```

```
void doStuff(int x) {  
    // this is a scope  
    // parameters such as x  
    // are also part of  
    // this scope  
}
```

```
switch(condition) {  
    // this is a scope  
}
```



While switch statements have their own scope, declaring variables within them is not permitted.

Other Scopes

```
namespace {  
    // this is a scope  
}
```

```
class Object {  
    // this is a scope  
};
```

```
struct AlsoObject {  
    // this is a scope  
};
```

```
enum class ScopeTypes {  
    // this is a scope  
};
```

For the sake of completion, these constructs also have their own scopes.

Like block scopes, names declared within them only exist within the bounds of the { } braces.

For accessing some members in these scopes you can use the scope operator :: (for example: std::cout)

Today

- Memory Management
- Pointers
- Scope
- **Memory Allocation / Deallocation**
- Copy Constructor

Memory Allocation & Deallocation

- Two main types of memory regions:
 - **The stack**
 - All function variables and parameters
 - Allocation and deallocation is automatic
 - Deallocation happens when the variable goes out of scope
 - Intended for small amounts of memory
 - The heap

Memory Allocation & Deallocation

- Two main types of memory regions:
 - The stack
 - **The heap**
 - Done explicitly using new/delete
 - Allocation and deallocation are manual
 - Deallocation happens manually when you say that you are done using the memory
 - Intended for memory amounts of any size

Memory Allocation & Deallocation

- **Allocating on the Stack**
- Allocating on the Heap (everyone calls this the heap, but C++ insists on calling it the free store)
- The risks of managing memory yourself

Allocating on the Stack

- Variables declared in block scopes (for loops, functions, if statements, etc) are allocated at the start of the scope, and deleted automatically at the end of the scope in which they are defined

```
int main() {    ←—— memory to store x and y is allocated here
    int x = 5;  ←—— x is defined here
    int y = x;  ←—— x can be used any place it is defined
    return 0;
}              ←—— x and y are automatically deallocated here
```

Demonstration – stack allocation

Task: stack issues

Find the problem with this program.

```
struct Zombie {  
    int health = 0;  
    // Things related to braaaaaaiinnss go here  
};  
  
Zombie* spawnZombie() {  
    Zombie zomb;  
    zomb.health = 100;  
    return &zomb;  
}  
  
int main() {  
    Zombie* zomber = spawnZombie();  
    Zombie* moreZomber = zomber;  
    Zombie zombest = *moreZomber;  
    moreZomber->health = 200;  
    std::cout << zomber->health << std::endl;  
    return 0;  
}
```

Memory issue: dangling pointer

- Dangling reference:
A pointer that references memory that has already been deallocated

Allocating on the Stack

- Two minor issues:
 - The stack has limited space
Usually around 1 to 8 MiB depending on compiler and OS
This program allocates more space than is available:

```
#include <array>

int main() {
    std::array<int, 1024 * 1024 * 1024> giantArray;
    return 0;
}
```

- All sizes for stack variables must be known at compile time
(which is why `std::array` by default stores its data on the stack,
but `std::vector` does not)

Memory Allocation & Deallocation

- Allocating on the Stack
- **Allocating on the Heap** (everyone calls this the heap, but C++ insists on calling it the free store)
- The risks of managing memory yourself

Allocating on the Heap

- Allocating on the heap is done using the **new** operator.
 - C++ will ask the OS for enough memory to store the data type you specify
 - The **new** operator returns a pointer to where in memory your data has been allocated by the OS

```
int* pointerOnTheHeap = new int {0};
```

```
*pointerOnTheHeap = 10;
```

```
std::cout << *pointerOnTheHeap << std::endl;
```

Replace **int** with any data type you wish to allocate

The braces initialise the allocated **int** to 0. We have already seen how to use pointers

Deallocating data on the Heap

- Data allocated on the heap is not deleted automatically!
- We need to explicitly tell C++ that we are done with each value we allocate by using the **delete** operator.

```
int* pointerOnTheHeap = new int {0};
```

```
delete pointerOnTheHeap;
```



For every **new** there must be
a corresponding **delete**

Allocating an array of values

- It is possible to allocate an array on the heap using the **new**[] operator
 - Recommendation: a `std::vector` does basically the same thing. Use it as much as you can!

Replace **int** with your data type of choice Length of the array goes here You can initialise values using { }

```
int* heapArray = new int[5] {1, 2, 3, 4, 5};
```

```
heapArray[2] = 10; // we cannot use at() and must use []  
heapArray[7] = 3; // [] does not do bounds checking
```

```
delete[] heapArray;      ← Memory allocated using new[] must  
use delete[] to dealocate
```

Demonstration – heap allocation

RAM Allocation

The Operating System (OS) is responsible for allocating RAM to running programs

Memory allocation

1. program asks for memory

Can I have 48,879 bytes of memory?

2. OS reserves a memory region of the requested size

Sure! I assigned a region of 48,879 bytes to you

3. program uses the memory

reading and writing intensifies

Memory deallocation

4. program notifies the OS that it is done using the memory

I no longer need those 48,879 bytes. Another program can use them :)

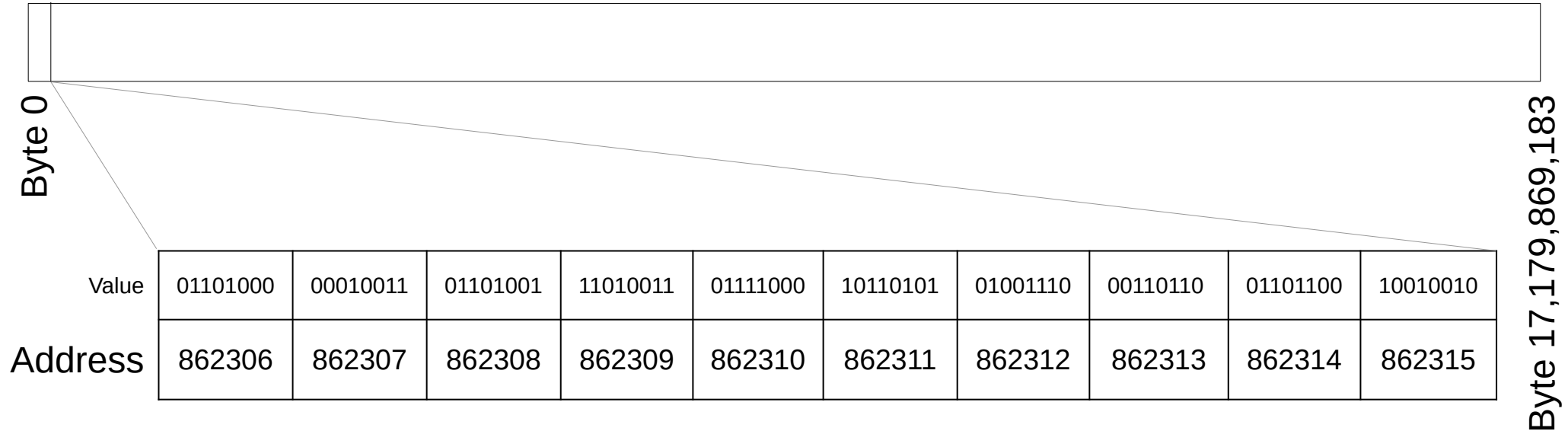
5. OS marks the memory region as available

Cool! Those 48,879 bytes are no longer assigned to you.

Memory Addresses

Programs and the OS specify where something resides in memory using addresses

A memory address is the «index» of a byte in memory, where the first byte has the address of 0



Memory Allocation & Deallocation

- Allocating on the Stack
- Allocating on the Heap (everyone calls this the heap, but C++ insists on calling it the free store)
- **The risks of managing memory yourself**

Memory Management Risks

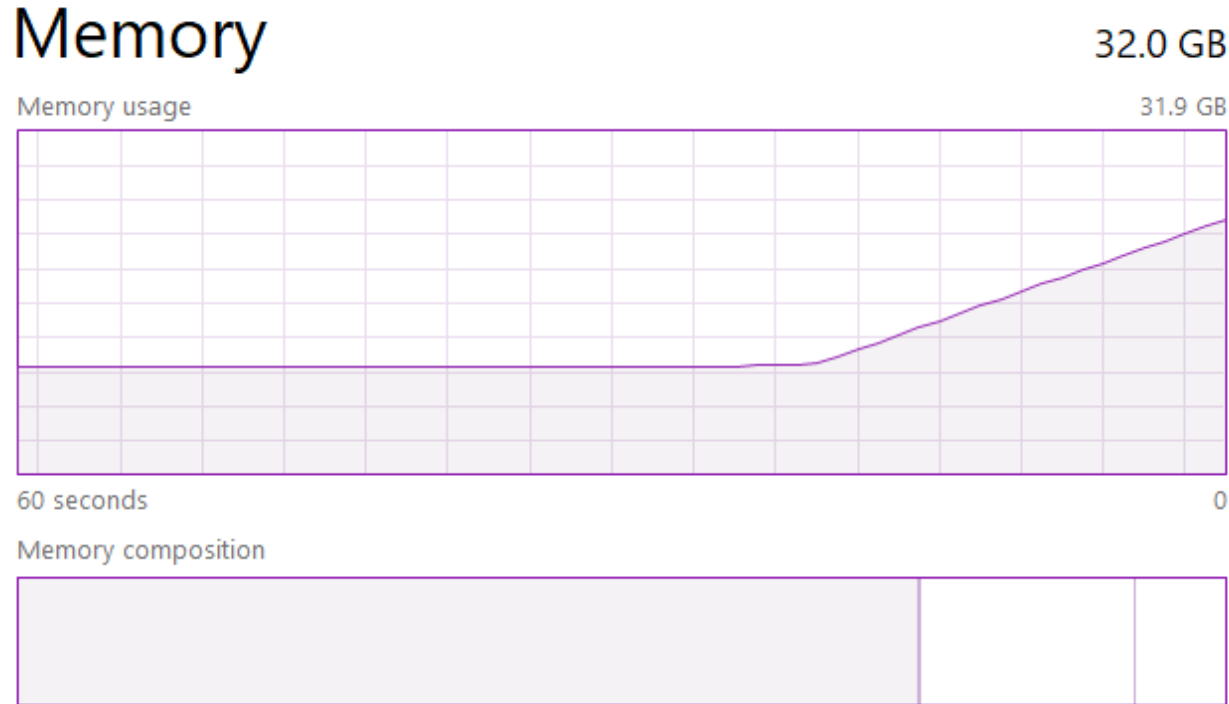
- Memory leak:
The pointer to heap allocated memory is lost before the heap memory is deallocated.

Delete can only be used with a pointer to allocated heap memory, so when this pointer is lost, calling delete becomes impossible.

```
for(int i = 0; i < 10; i++) {  
    std::string* helpMessage = new std::string("Oh no!");  
    std::cout << *helpMessage << std::endl;  
}
```

Memory Management Risks

- Memory leak:
The worst ones are quite easy to detect



Memory Management Risks

- Dangling reference:
A pointer that references memory that has already been deleted

```
int* heapArray = new int[5] {1, 2, 3, 4, 5};  
delete[] heapArray;
```

```
// heapArray is now a dangling reference  
// it could still be used, but no longer can be used  
// heapArray[3] = 32; <- this is for example not allowed
```

Memory Management Risks

- Double free:
Attempting to delete heap memory that has already been deleted

```
int* heapArray = new int[5] {1, 2, 3, 4, 5};  
  
for(int i = 0; i < 10; i++) {  
    delete[] heapArray;  
}
```

Memory Allocation & Deallocation

- Allocating on the Stack
- Allocating on the Heap (everyone calls this the heap, but C++ insists on calling it the free store)
- The risks of managing memory yourself

Miscellaneous: modifying pointers

- Using arithmetic operators directly changes the address, not the value it references:

```
int number = 5;
int* pointerToNumber = &number;

pointerToNumber++;
// no idea what this prints, but it's not the value of number
// because pointerToNumber now references another address
std::cout << *pointerToNumber << std::endl;
```

- Recommendation: put const after the pointer type if you don't intend to change the address. This ensures you can't modify the address referenced by the pointer.

```
int* const pointerToNumber = &number;
pointerToNumber++; // error! You're modifying the address
*pointerToNumber++; // all good
```

Miscellaneous: C strings

- C strings are arrays of characters, called as such because `std::string` is not a thing in C

```
char* usefulText = "The more you drink, the more WC!";
```

Stack variable

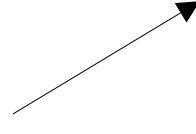
```
char* usefulText;
```

Somewhere in memory

'T'	'h'	'e'	' '	'm'	'o'	'r'	'e'	' '	'y'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Read characters using the `[]` operator, just like all arrays:

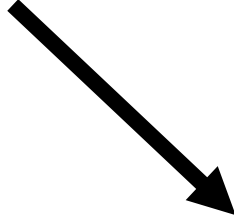
`usefulText[4]`



Miscellaneous: C strings

- C strings are arrays of characters, called as such because `std::string` is not a thing in C
- They were designed in a time when memory was expensive, so rather than store a length (like `std::string`'s `size()`), a byte set to 0 is used instead to denote the end of the string

Somewhere in memory



'e'	' '	'm'	'o'	'r'	'e'	' '	'W'	'C'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	------

```
char* usefulText = "The more you drink, the more WC!";
```


Today

- Memory Management
- Pointers
- Scope
- Memory Allocation / Deallocation
- **Copy Constructor**

Copy Constructor

- A special constructor that is called when copying an object.
 - Happens when assigning a variable to another, or when using pass-by-value for a function parameter
- If your object allocates heap memory in its constructor, your copy constructor should ensure that memory is duplicated in a separate region of memory

Copy Constructor: Syntax

```
class Houston {  
    Satellite* satellite;  
public:  
    Houston() {  
        satellite = new Satellite("sputnik");  
    }  
    Houston(const Houston& other) {  
        satellite = new Satellite(other.satellite->name);  
    }  
};
```

The copy constructor is another constructor which takes a **const** reference to an instance of the same object as its only parameter

Today

- Memory Management
- Pointers
- Scope
- Memory Allocation / Deallocation
- Copy Constructor

Next week

- DESTRUCTORS
- Automatic deletion using `unique_ptr` and `shared_ptr`
- Graphical User Interfaces (GUI)
- `std::unordered_map`