Lecture 10: Addressing References

Bart Iver van Blokland

(Rune Sætre)

Last week

- Insperaøving

Two weeks ago

- Inheritance
- Virtual methods
- Operator overloading
- The friend keyword

Inheritance

- Inheritance lets you reuse parts of classes, and change other parts
- A parent class is «copied and pasted» into the child class by the compiler.

```
struct ParentClass {
    void aMethod() {}
    int aField = 10;
};

class ChildClass : public ParentClass {
};

int main() {
    ChildClass child;
    std::cout << child.aField << std::endl;
}
```

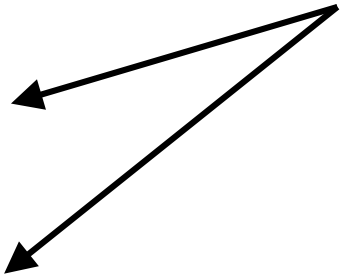
The parent class to inherit from is declared after the class name

ChildClass has inherited the aField field from ParentClass.

Inheritance: Overriding methods

- Declaring the exact same method in the child class will overwrite the one inherited from the parent class
- Useful for changing portions of the class's behaviour

```
class Lamp {
    bool isLit = false;
    bool isLightOn() {
        return isLit;
    }
};

class BrokenLamp : public Lamp {
    bool isLightOn() {
        return false;
    }
};
```

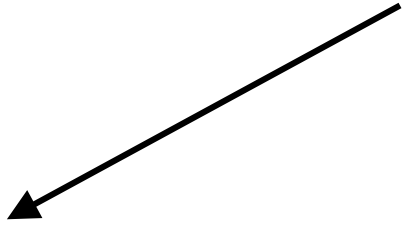


The overriding method must have the exact same name, parameters, and return type

When creating an instance of `BrokenLamp` and calling `isLightOn()` will run the overridden version!

Virtual methods

```
struct Lamp {
    virtual void shine() {}
};

class BlueLamp : public Lamp {
    void shine() {
        std::cout << "[blue light]" << std::endl;
    }
};

int main() {
    BlueLamp blueLamp;
    Lamp& childReference = blueLamp;
    childReference.shine(); // prints [blue light]
    return 0;
}
```



Because shine() is marked as virtual, the BlueLamp version of shine() will be executed, even though the reference is of type Lamp

Operators are functions

- Operators are normal functions, except:
  - The name must be `operator` with the desired operator appended to it (e.g. `operator*` or `operator!=`)
  - They are called by using the operator

```
std::string operator+(std::string text, int number) {
    return text + std::to_string(number);
}

int main() {
    std::string message = "The number is: ";
    std::string combined = message + 5;
    return 0;
}
```

Using the operator calls the operator function!

Using friend: Functions

- Primary use: giving access to a specific function

```
class House {
    bool plantsHaveBeenWatered = false;
    void openFrontDoor();

    friend void useHouseKey(House &house);
};

void useHouseKey(House &house) {
    house.openFrontDoor();
    house.plantsHaveBeenWatered = true;
}
```

Simply copy and paste the function declaration after the friend keyword

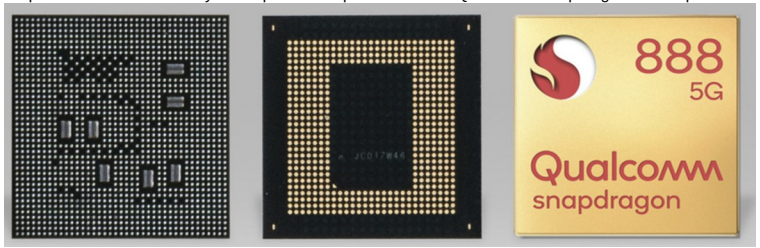
The function can now use private fields and methods!

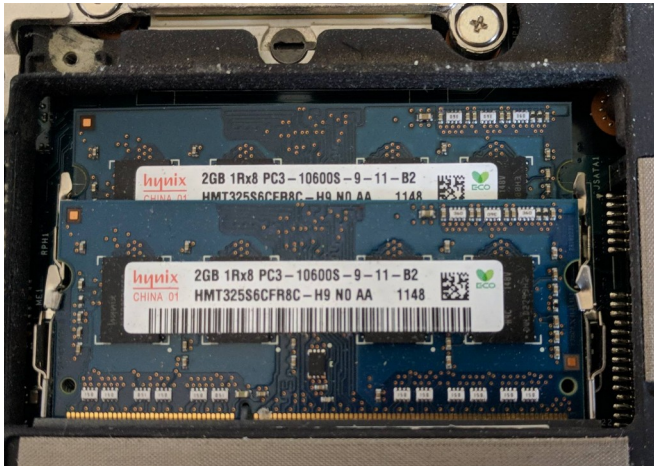
Today (and next week)

Memory

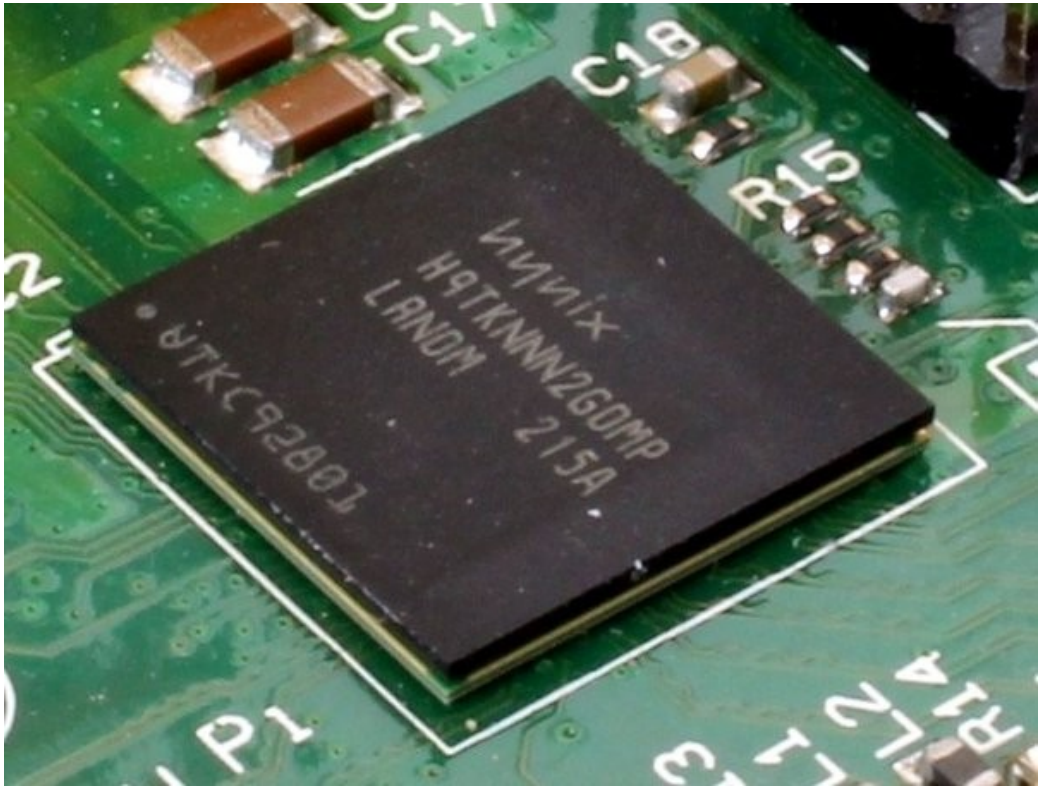
Memory

https://www.androidauthority.com/wp-content/uploads/2020/11/Qualcomm-Snapdragon-888-chip-front-and-back.jpg

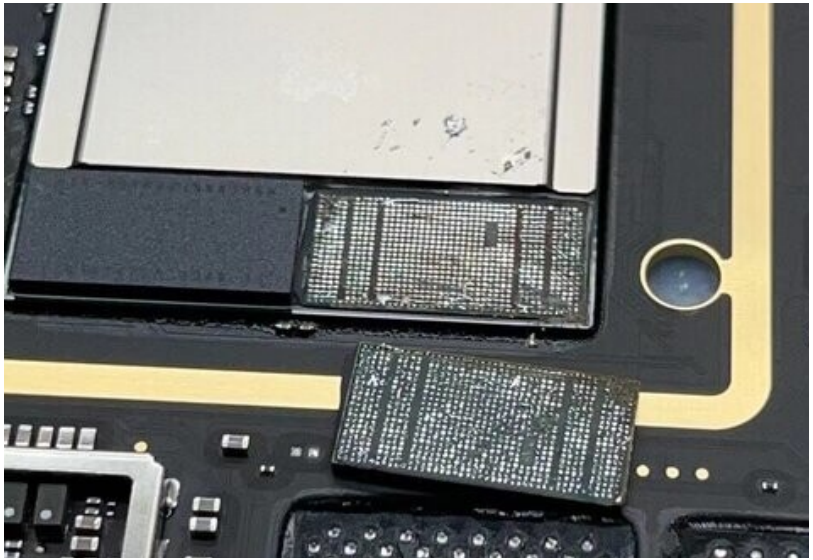




https://superuser.com/questions/1280480/laptop-ram-compatibility



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



https://www.techpowerup.com/img/wPXfXNdx31ZD8ceT.jpg

Why is understanding memory important?

- Pointers are highly popular in C++ libraries, and using such libraries often requires them
- Programs use memory all the time
- Understanding how a tool works makes you better at using that tool
- One of the main places where C++ gives you extremely tight control
  (and where other programming languages do not, with few exceptions)

Today

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

Random Access Memory (RAM)

RAM is used by any program on your computer, such as:

- A web browser
- VS Code
- A text editor
- (usually a lot more)

RAM is used for storing all data used by a program. For example:

- An image on a webpage
- A sound file that your music player is playing
- Some text shown in an error message
- Any variable or data that your program might want to store, such as:

```
std::vector<int> lotsOfNumbers((1024 * 1024 * 1024) / 4);
```

Question: How is this RAM divided between applications?

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

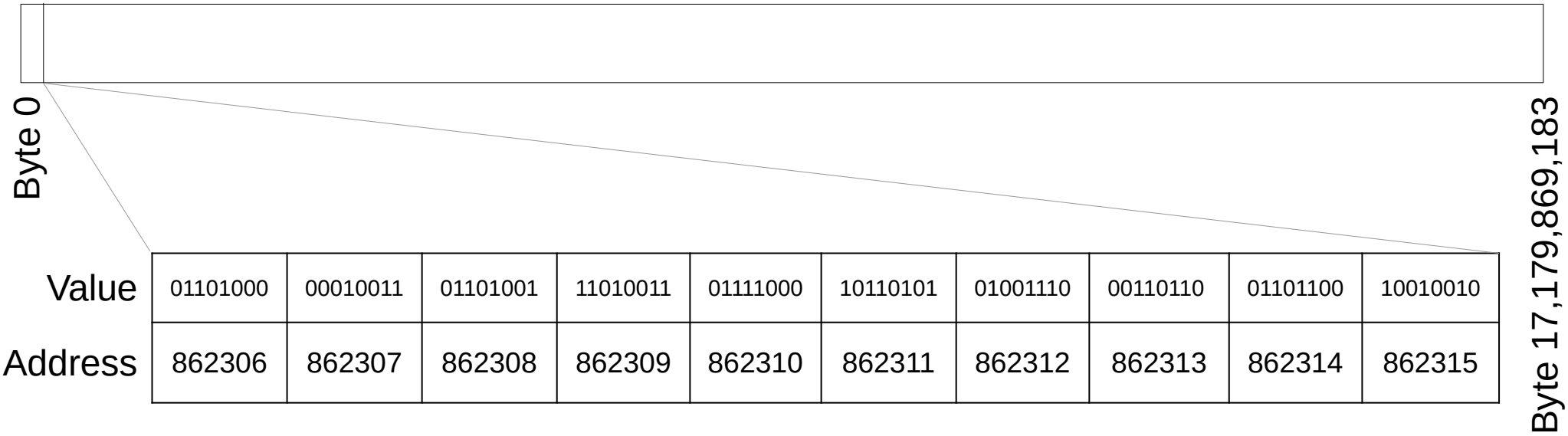
4. program notifies the OS that it is done using the memory
I no longer need those 48,879 bytes. Another program can use them :)

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

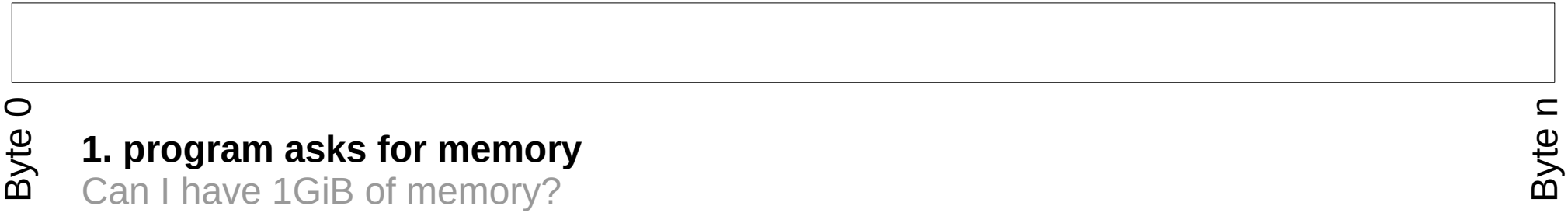
A memory address is the «index» of a byte in memory, where the first byte has the address of 0



Value	01101000	00010011	01101001	11010011	01111000	10110101	01001110	00110110	01101100	10010010
Address	862306	862307	862308	862309	862310	862311	862312	862313	862314	862315

RAM Allocation

What allocation looks like:

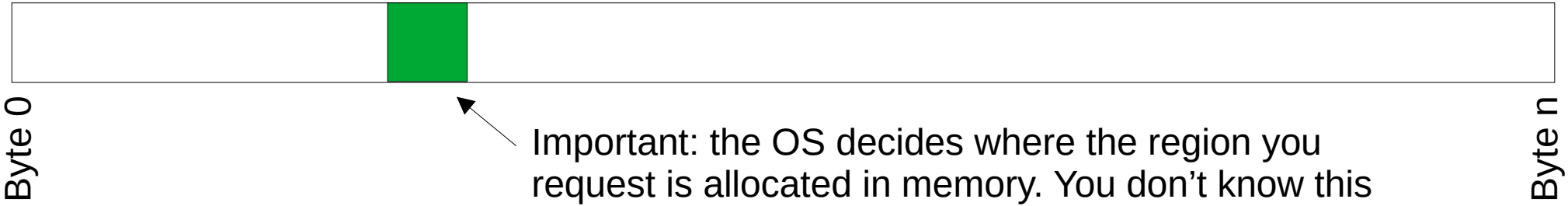


1. program asks for memory

Can I have 1GiB of memory?

2. OS reserves a memory region of the requested size

Sure! The region from address 4,311,744,512 to 5,385,486,335 is now assigned to you



Important: the OS decides where the region you request is allocated in memory. You don't know this location in advance, and it will vary between runs.

RAM Deallocation

What deallocation looks like:

Byte 0 Byte n

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

I no longer need the region from address 4,311,744,512 to 5,385,486,335.

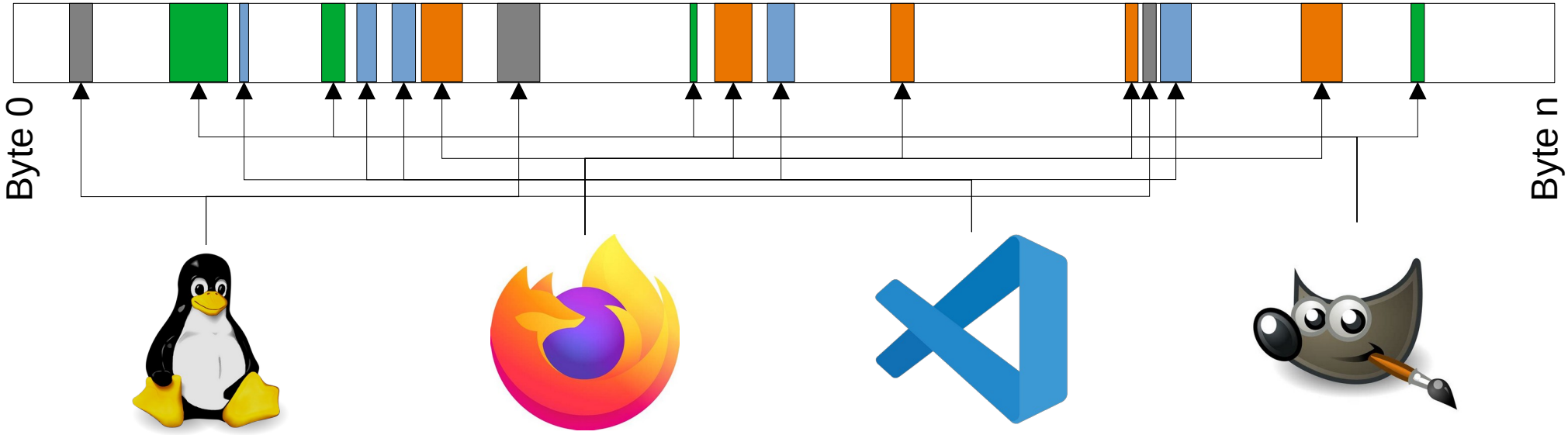
5. OS marks the memory region as available

Cool! That region is no longer assigned to you.

Byte 0 Byte n

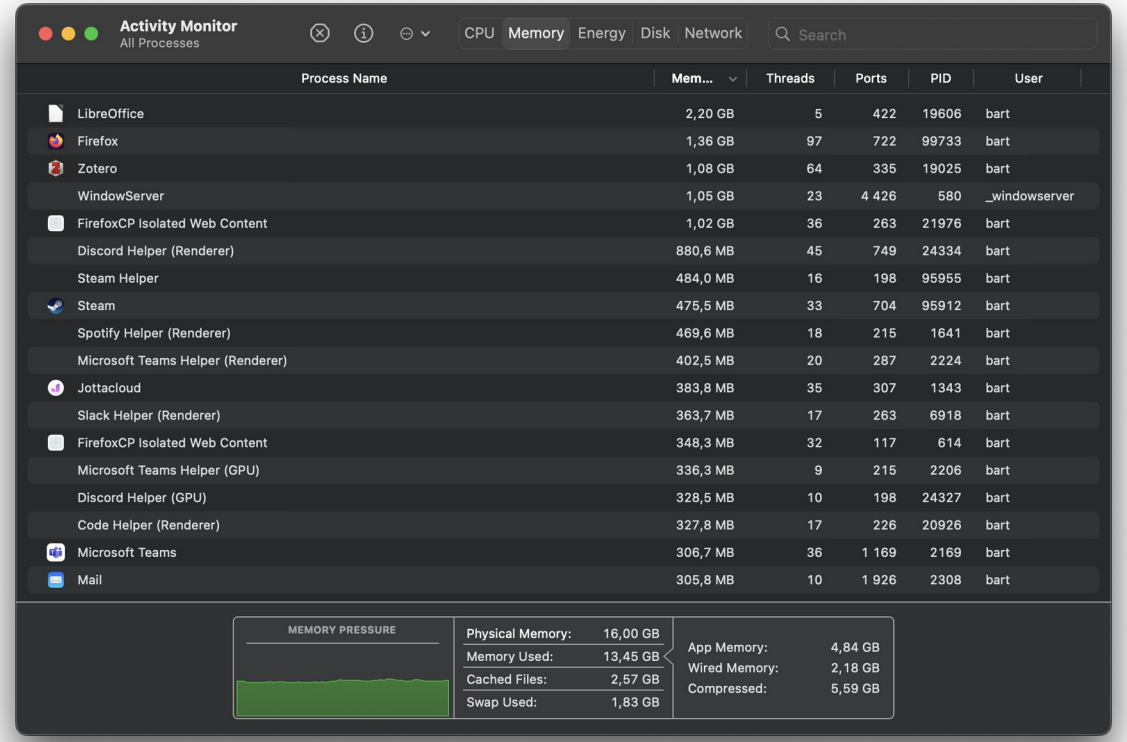
RAM is shared!

- Most programs have loads of regions allocated to them
- Each program can only use the regions allocated to it
- Even the OS has its own memory regions



RAM is shared!

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



Summary: Memory Management

- Memory is shared between programs
- Regions of memory are allocated to specific programs upon request
- A program can allocate as many regions and as much memory as it needs, provided there is enough space available
- Programs are only allowed to use memory regions assigned to them
    - If you try to access any other part of memory, the OS intervenes, and you get an “Access Violation” or “Segmentation Fault” error
- **Programs are themselves responsible to notifying the OS when they no longer need a memory assigned to them**

Today

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

Pointer

- Definition: A variable which contains a memory address
- Syntax:

```
int* pointerToNumber = nullptr;
```

`int` can be replaced with any data type. Meaning: «if you look at the address stored in this pointer, you will find an `int`»

The * indicates this variable is a pointer to `int`

Variable name

The default value of a pointer should always be **`nullptr`**, meaning «pointer to nowhere»

Example 1

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;
```

Example 1

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
```

Example 1

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!

Example 2

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 <u>after</u> 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
```

Example 1

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;
```

Example 1

Today

- Memory Management
- Pointers
- **Scope**
- Memory Allocation / Deallocation
- DESTRUCTORS
- 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!
```

Example 3

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**
- DESTRUCTORS
- Copy Constructor

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() {
    int x = 5;
    int y = x;
    return 0;
}
```

memory to store x and y is allocated here

x is defined here

x can be used any place it is defined

x and y are automatically deallocated here

Example 4

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` is able to store its data on the stack)

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

- Most memory of a program is allocated dynamically as regions of memory.

  As a refresher:
  1) Program asks the OS for an amount of memory
  2) OS allocates a region and assigns it to the program
  3) Program uses the memory
  4) Program notifies the OS it is done using the memory
  5) OS deallocates the region

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

Example 5

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.

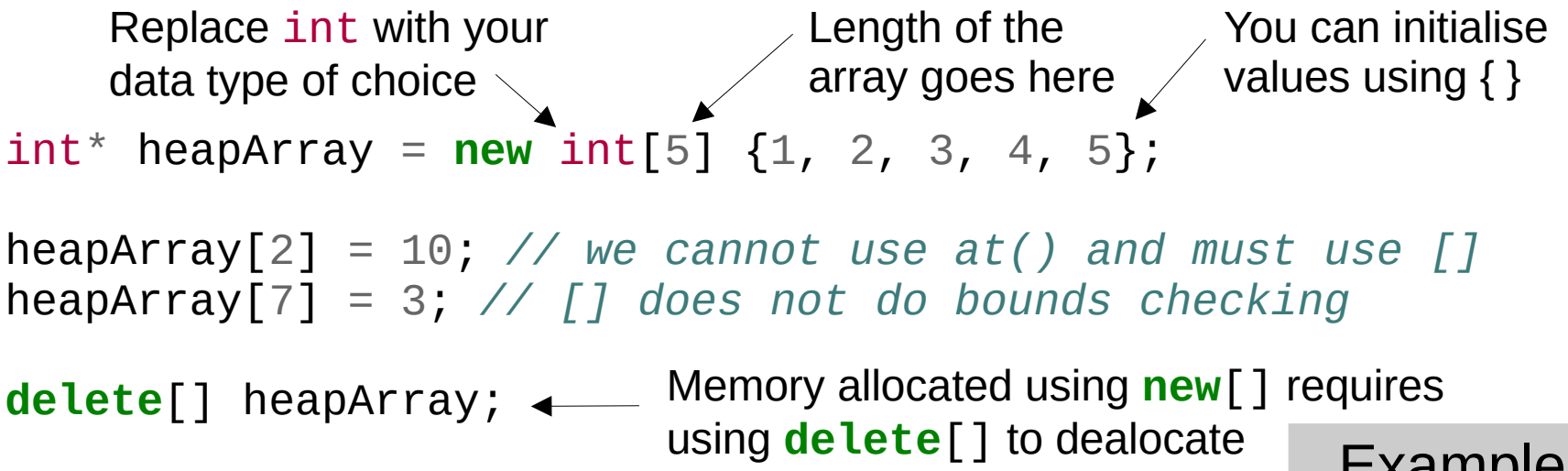
```
int* pointerOnTheHeap = new int {0};

delete pointerOnTheHeap;
```

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

Example 5

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!



Example 5

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

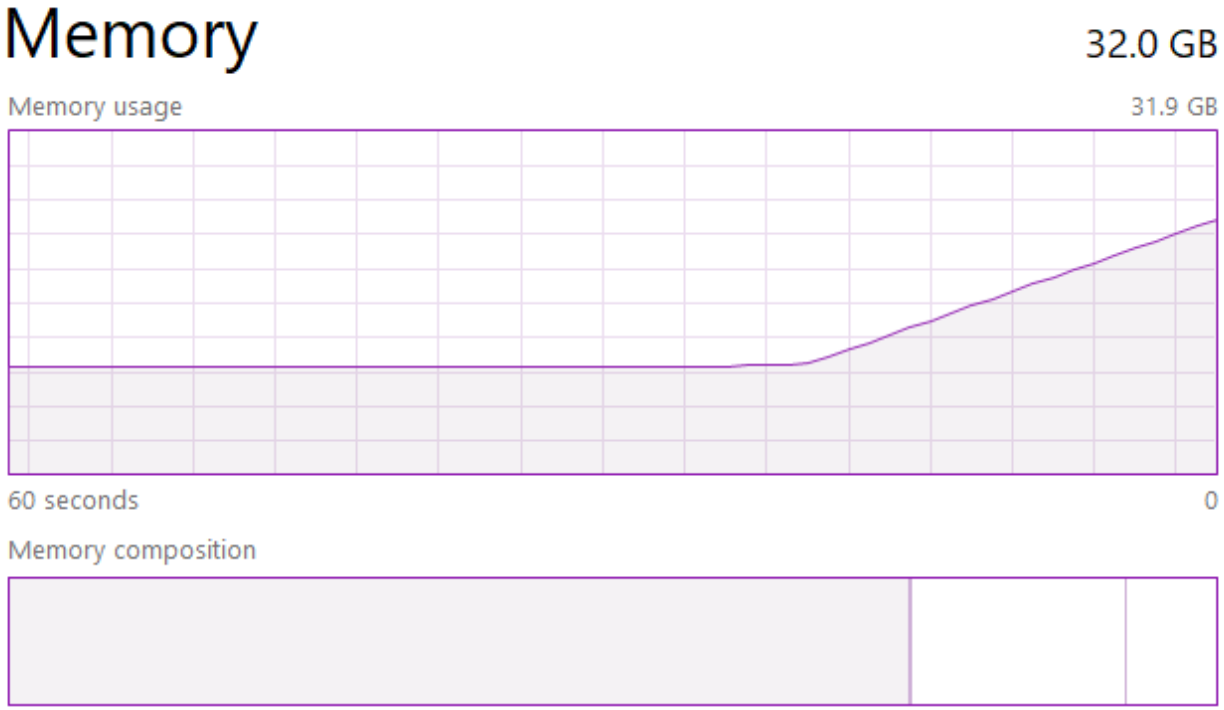
- Memory leak:
  The pointer to heap allocated memory is lost before the heap memory is deallocated

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

Example 6

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
```

Example 6

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;
}
```

Example 6

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

We have a problem..

```
class Houston {
    Satellite* satellite;
public:
    Houston() {
        satellite = new Satellite("sputnik");
    }
};
```

We can’t ever free the memory of satellite, because it is private to the class Houston

What do we do?


DESTRUCTOR

Destructors

- Destructors are functions that are automatically run when an object is deleted

- Syntax is the same as constructors, except the name has ~ in front, and it cannot have parameters.

```
class Houston {
    Satellite* satellite;
public:
    Houston() {
        satellite = new Satellite("sputnik");
    }
    ~Houston() {
        delete satellite;
    }
};
```

The desctructor is executed automatically when an instance of Houston is deleted

Example 7

Today

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

There is another problem..

```
class Houston {
    Satellite* satellite;
public:
    Houston() {
        satellite = new Satellite("sputnik");
    }
    ~Houston() {
        delete satellite;
    }
};

int main() {
    Houston houston1;
    Houston houston2 = houston1;
}
```

```
malloc: Double free of object 0x1002041e0
```

Example 8

Copy Constructor

- A special constructor that is called when copying an object (assigning one instance to another).
- If an object allocates its own memory, you must define both a destructor and copy constructor

Copy Constructor: Syntax

```
class Houston {
    Satellite* satellite;
public:
    Houston() {
        satellite = new Satellite("sputnik");
    }
    ~Houston() {
        delete satellite;
    }
    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

Example 9

Today

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

Next week

- Automatic deletion using unique_ptr and shared_ptr
- Graphical User Interfaces (GUI)