# - Examples

Let's look at a couple of pointers examples.

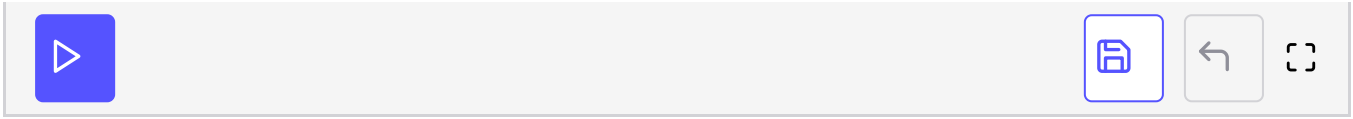
WE'LL COVER THE FOLLOWING



## Basic pointers #

```
#include <iostream>

int main(){

  std::cout << std::endl;

  int i{2011};
  int* iptr= &i;

  std::cout << i << std::endl;

  std::cout << "iptr: " << iptr << std::endl;
  std::cout << "*iptr: " << *iptr << std::endl;

  std::cout << std::endl;

  int * jptr = iptr;
  *jptr = 2014;
```

```
  std::cout << "*iptr: " << *iptr << std::endl;

  std::cout << "jptr: " << jptr << std::endl;
  std::cout << "*jptr: " << *jptr << std::endl;

}
```

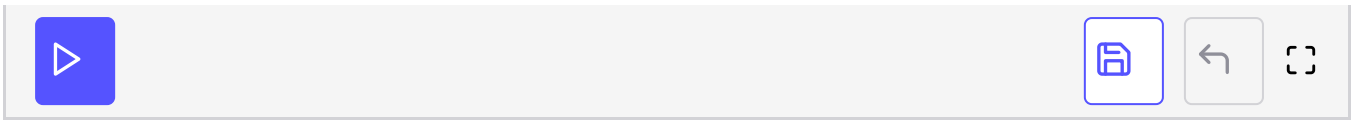


## Explanation #

- This example shows an instance of two pointers pointing to the same object.
- Since both `iptr` and `jptr` point to `i`, changing the dereferenced value of `jptr` in line 18 changes the values of `i` and `iptr` as well.

# Pointer arithmetic #

```
#include <iostream>

int main(){

  int intArray[] = {1, 2, 3, 4, 5};
  if (intArray[3] == *(intArray + 3)) std::cout << "Pointer arithmetic works" << std::endl;
}
```



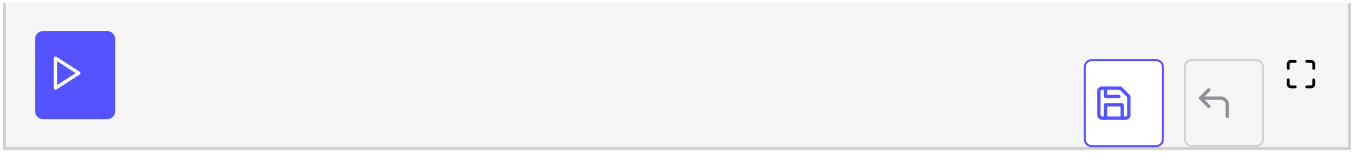
## Explanation #

- As we discussed earlier, arrays use pointer arithmetic.
- This can be seen in line 6 where both syntaxes return the same value.

## `nullptr` #

```
#include <iostream>
using namespace std;

int main() {
  int* pi = nullptr;       // OK
  std::cout << pi << std::endl;

  // int i = nullptr;      // ERROR

  bool b{nullptr};         // OK
  std::cout << b << std::endl;
```

```
}
```

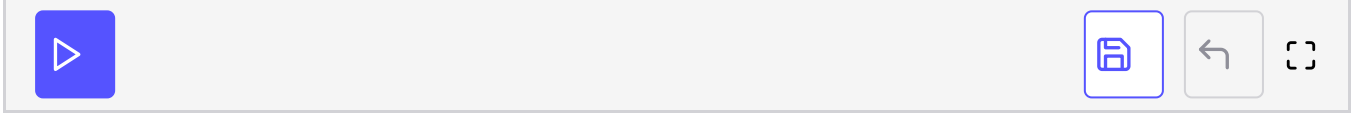


## Explanation

- The `nullptr` can be assigned to any arbitrary pointer, as seen in line 5.
- The `nullptr` cannot be assigned to any arbitrary variable except a `bool`. This will only work when creating a bool through uniform initialization, as seen in line 10.

# Function pointers

```
#include <iostream>

void addOne(int& x){
  x += 1;
}

int main(){

  void (*inc1)(int&)= addOne;
  auto inc2 = addOne;

  int a{10};

  addOne(a);
  std::cout << "after addOne(a): " << a << std::endl;
  inc1(a);
  std::cout << "after inc1(a): "  << a << std::endl;
  inc2(a);
  std::cout << "after inc2(a): "  << a << std::endl;

  std::cout << std::endl;
}
```



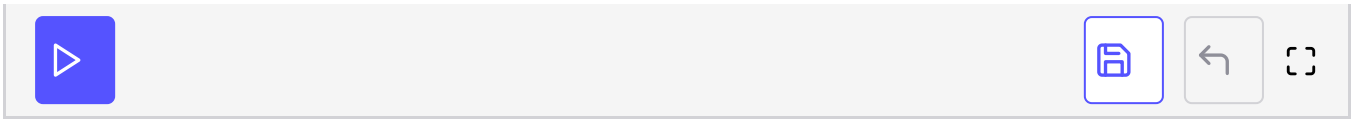
## Explanation

- In `addOne`, an integer is passed by reference. Hence, calling the function will change the actual value of the integer.
- `inc1` points to the `addOne` function. Hence, it will have the same functionality.

- We do not need to explicitly define the type of the `inc2` function pointer. This is because we use the `auto` keyword. We'll study this in more detail in the near future.

## Pointer to member #

```
#include <iostream>

struct X{
  int data;
};

int main(){

  std::cout << std::endl;

  int X:: * p = &X::data;
  X object;
  object.data = 2011;
  X* objptr = new X;
  objptr->data = 2014;


  int k = object.*p;
  int l = objptr->*p;

  std::cout << "k: " << k << std::endl;
  std::cout << "l: " << l << std::endl;

  std::cout << std::endl;
}
```



## Explanation #

- This code is an example of how we can make pointers to the members of a struct or class.
- The pointer for the `data` member is made in line 11 using the following syntax:

```
pointerType structName :: pointerName = &structName :: dataMember
```

- The `pointerType` must match the type of the `dataMember`.
- The pointer is then dereferenced in lines 18 and 19.

With that, we'll end our discussion on pointers. In the next lesson, we'll explore **references**.