DIGITAL ARTS & ENTERTAINM

Pointers and Arrays

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

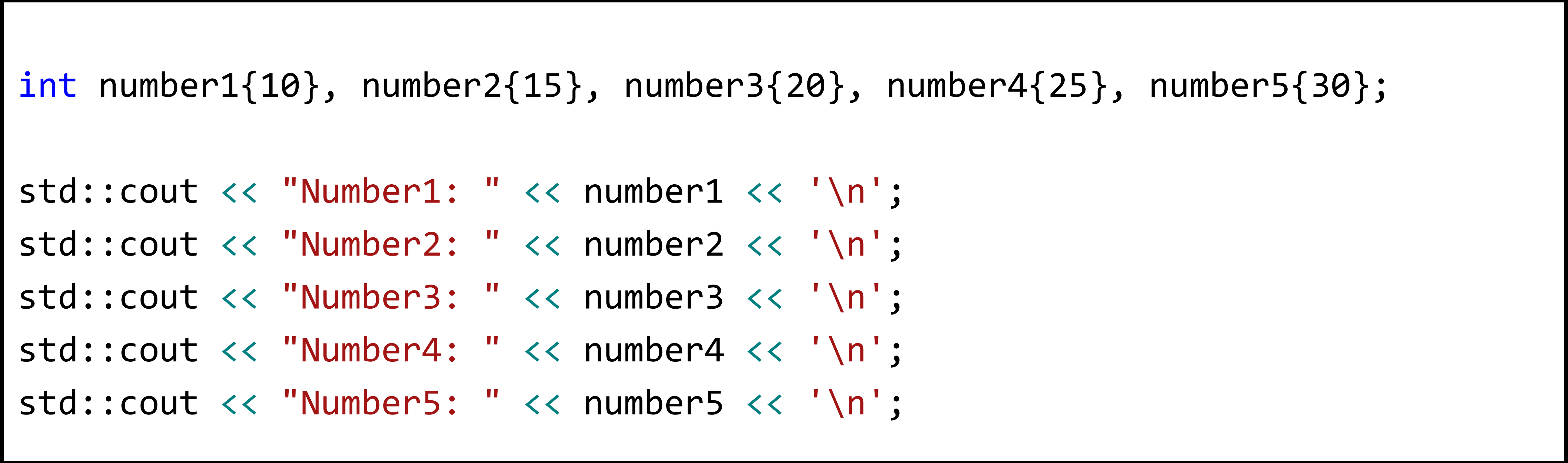
Overview course Programming 1

1. Intro + Variables 1
2. Variables 2
3. Variables 3
4. Iterations
5. Conditionals
6. Functions 1
7. Functions 2
8. Arrays
9. Strings – Game
10. Classes 1 – Encapsulation
11. Classes 2 – Static const

Content

- Array
- Pointer
- Arrays and pointers
- Functions, pointers and arrays

Consider this code snipped:



```cpp
int number1{10}, number2{15}, number3{20}, number4{25}, number5{30};

std::cout << "Number1: " << number1 << '\n';
std::cout << "Number2: " << number2 << '\n';
std::cout << "Number3: " << number3 << '\n';
std::cout << "Number4: " << number4 << '\n';
std::cout << "Number5: " << number5 << '\n';
```

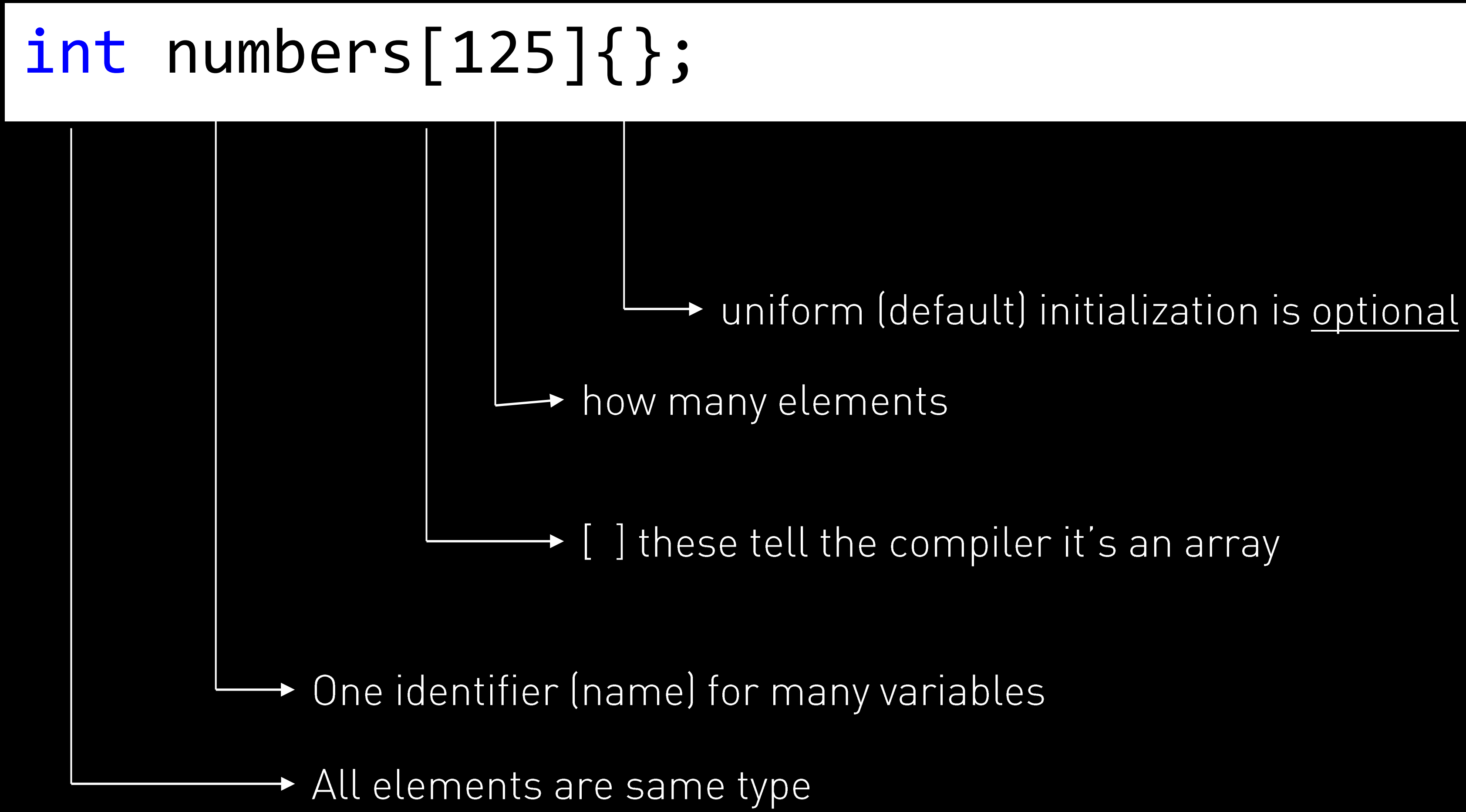
Would it not be nice to have another way to work with many similar variables?

Same, this time using an array

```
int numbers[5]{ 10, 15, 20, 25, 30 };

for (int i{ 0 }; i < 5; ++i)
{
  std::cout << "Number" << i << ": " << numbers[i] << '\n';
}
```

An array is a data type that lets us access many variables of the same type through a single identifier.

Fixed array declaration



```
int numbers[125]{};
```

uniform (default) initialization is <u>optional</u>

how many elements

[ ] these tell the compiler it’s an array

One identifier (name) for many variables

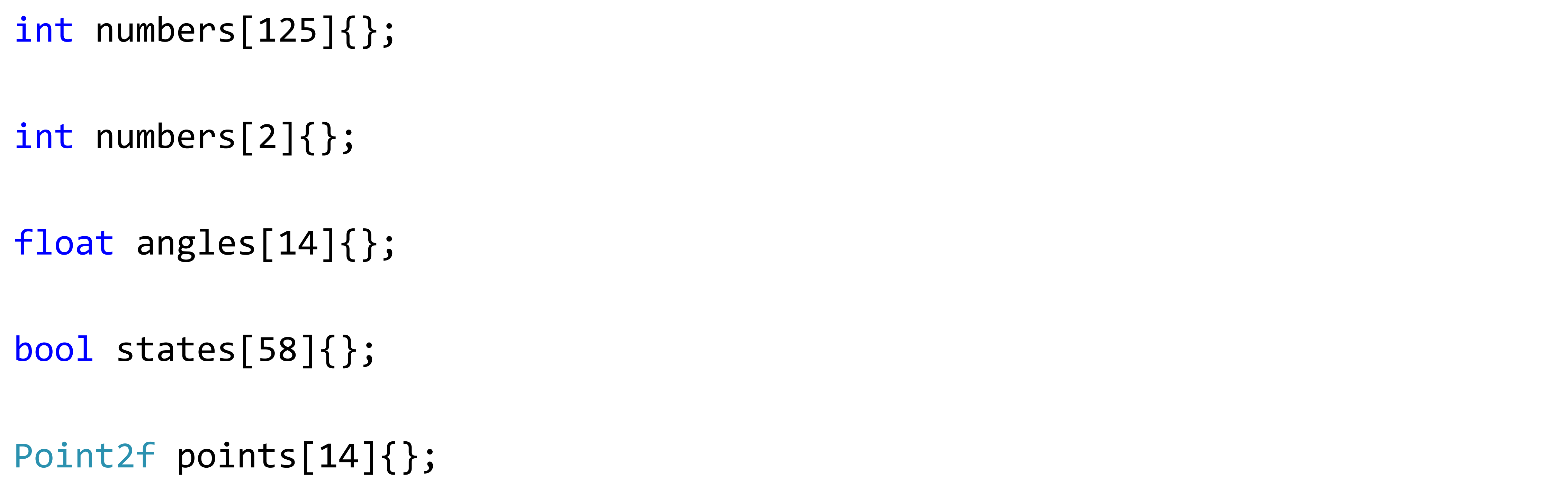
All elements are same type

Fixed array declaration

```
int numbers[125]{};
```

- A fixed length or sized array: length is known at compile time.
- The square brackets [ ] tell the compiler this variable is an array.
- Each of the variables in the array is called an element.
- Elements can be accessed through the subscript operator [ ]
- The index or subscript determines what element is accessed.
- The index starts at 0 (!)
- Arrays are not automatically initialized, need to use {}.

Fixed array declaration



```
int numbers[125]{};

int numbers[2]{};

float angles[14]{};

bool states[58]{};

Point2f points[14]{};
```

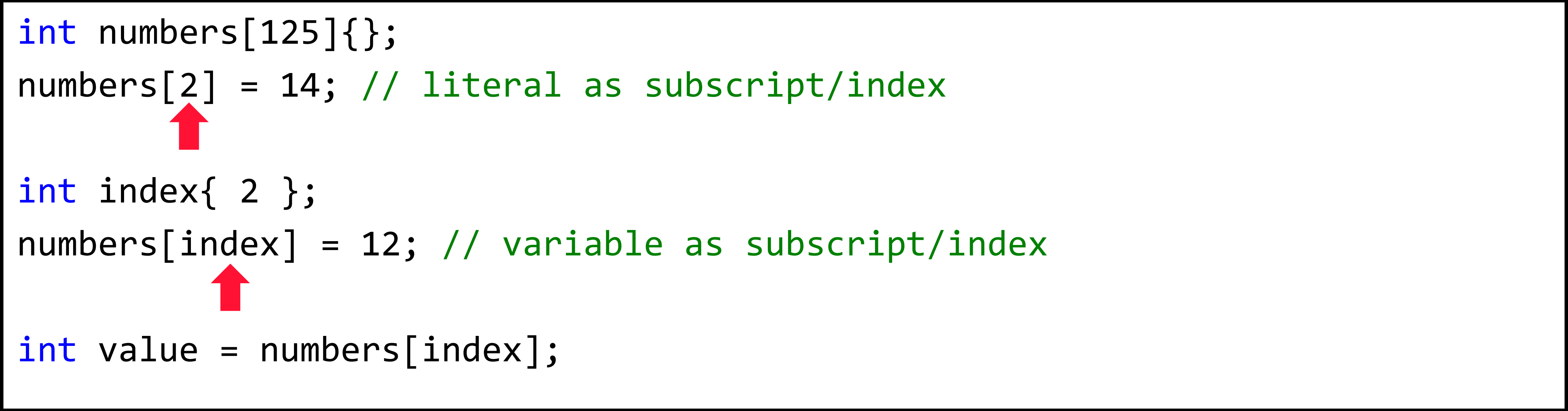
- Arrays can be made from any data type.

Fixed array declaration

```
int numbers[47]{}; // ok

int length{ 47 };
int numbers[length]{}; // NOT ok, length is not known at compile time

const int length{ 47 };
int numbers[length]{}; // ok, length is known at compile time
```

- Number of elements is defined by an integral type
- Length of an array must be a compile-time constant.
- literal or const int or if a member variable, a static const int

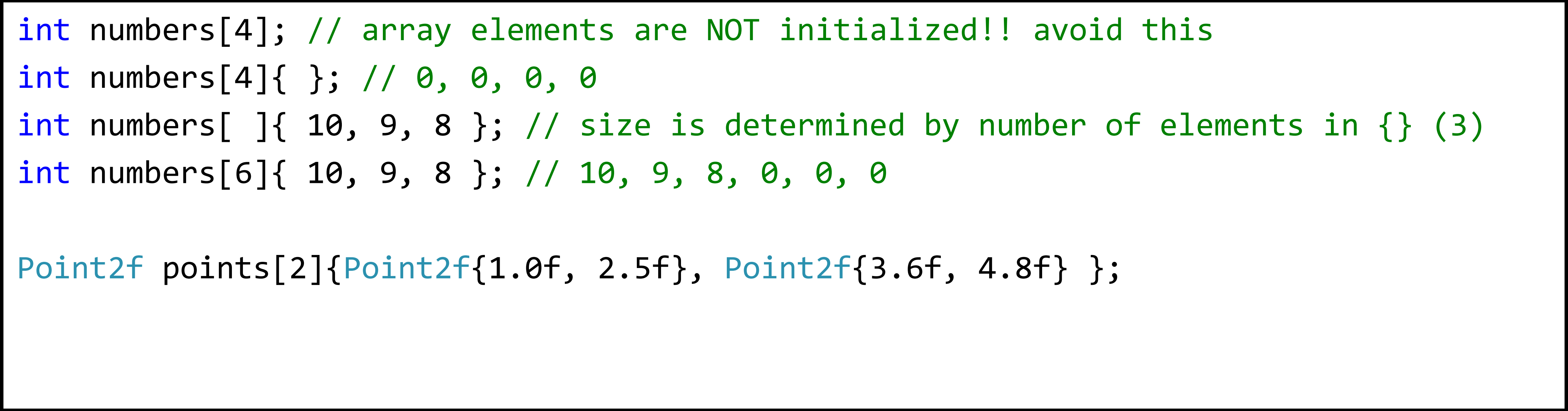
Array subscript operator [ ] and index



```
int numbers[125]{};
numbers[2] = 14; // literal as subscript/index

int index{ 2 };
numbers[index] = 12; // variable as subscript/index

int value = numbers[index];
```

- Must be an integral type. (char, int, unsigned int,…)
- Can be constant or non constant
- Literals or variables

Initializing arrays



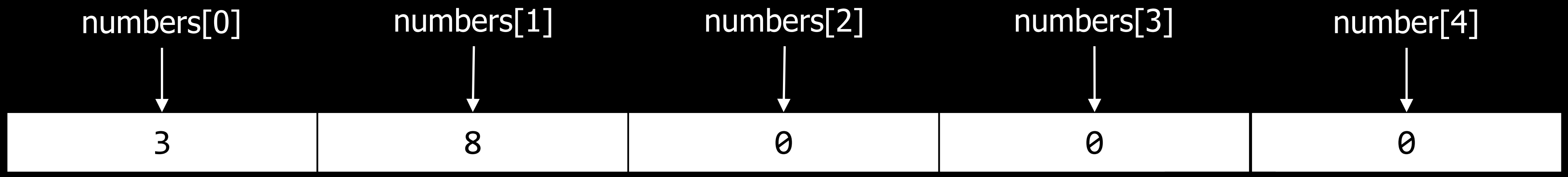
```
int numbers[4]; // array elements are NOT initialized!! avoid this
int numbers[4]{ }; // 0, 0, 0, 0
int numbers[ ]{ 10, 9, 8 }; // size is determined by number of elements in {} (3)
int numbers[6]{ 10, 9, 8 }; // 10, 9, 8, 0, 0, 0

Point2f points[2]{Point2f{1.0f, 2.5f}, Point2f{3.6f, 4.8f} };
```

- C++11: uniform initialization: { }
- Arrays are not automatically initialized
- Never leave an array uninitialized!

Array memory layout

```
int numbers[5]{3, 8};
```

- An array guarantees a contiguous block of memory

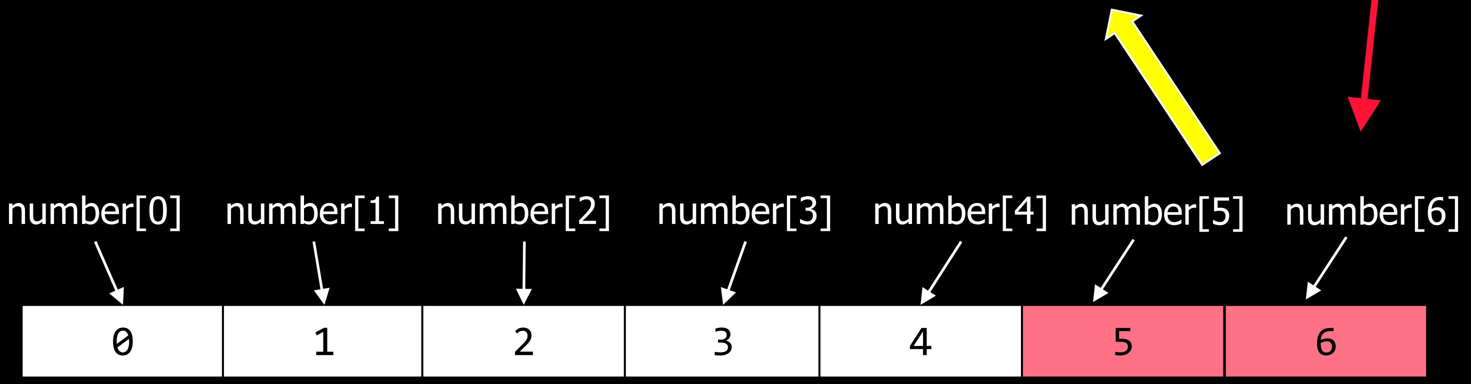


Array

Accessing an array using a for loop

```
int number[5]{};
for(int i = 0; i< 7 ; ++i)
{
  number[i] = i;
}
```

An array has no memory range protection



number[0]	number[1]	number[2]	number[3]	number[4]	number[5]	number[6]
0	1	2	3	4	5	6

Array

Accessing an array using a for loop

```
int number[5]{};
for(int i = 0; i< 7 ; ++i)
{
  number[i] = i;
}
```

An array has no memory range protection

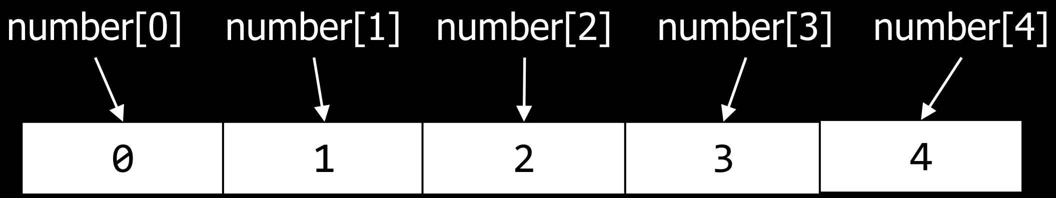
Resulting in undefined behavior: possible crashes, weird bugs, overwriting other variables

number[0]	number[1]	number[2]	number[3]	number[4]	number[5]	number[6]
0	1	2	3	4	5	6

Array

Always use a const int type to prevent agony and pain

```
const int size { 5 };
int number[size]{};
for(int i = 0; i< size ; ++i)
{
  number[i] = i;
}
```



number[0]	number[1]	number[2]	number[3]	number[4]
0	1	2	3	4

What have we learned?

- Never use literals to define array sizes.
  - When the size of the array is needed in more than one location, using literals is making the code vulnerable for bugs.

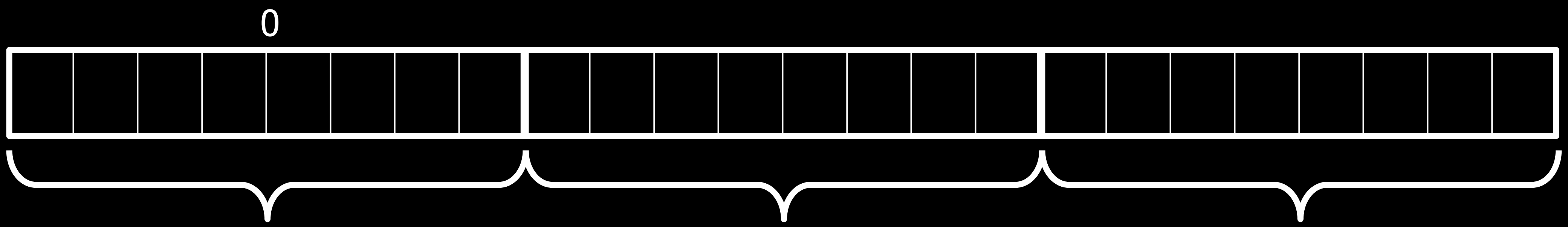
- Always use const variables to define the size of arrays.
  - Use the const variable to determine the max index.

- Index starts from 0 till the size of the array -1
- Never use an index that is larger than the size-1 or smaller than 0.
- Accessing elements that are out of bounds leads to undefined behaviour.

Content

- Array
- Pointer
  - Address of operator
  - Dereference operator
  - Pointer properties
- Arrays and pointers
- Functions

Computer memory



Memory cell 0

- 1 byte
- 8 bits

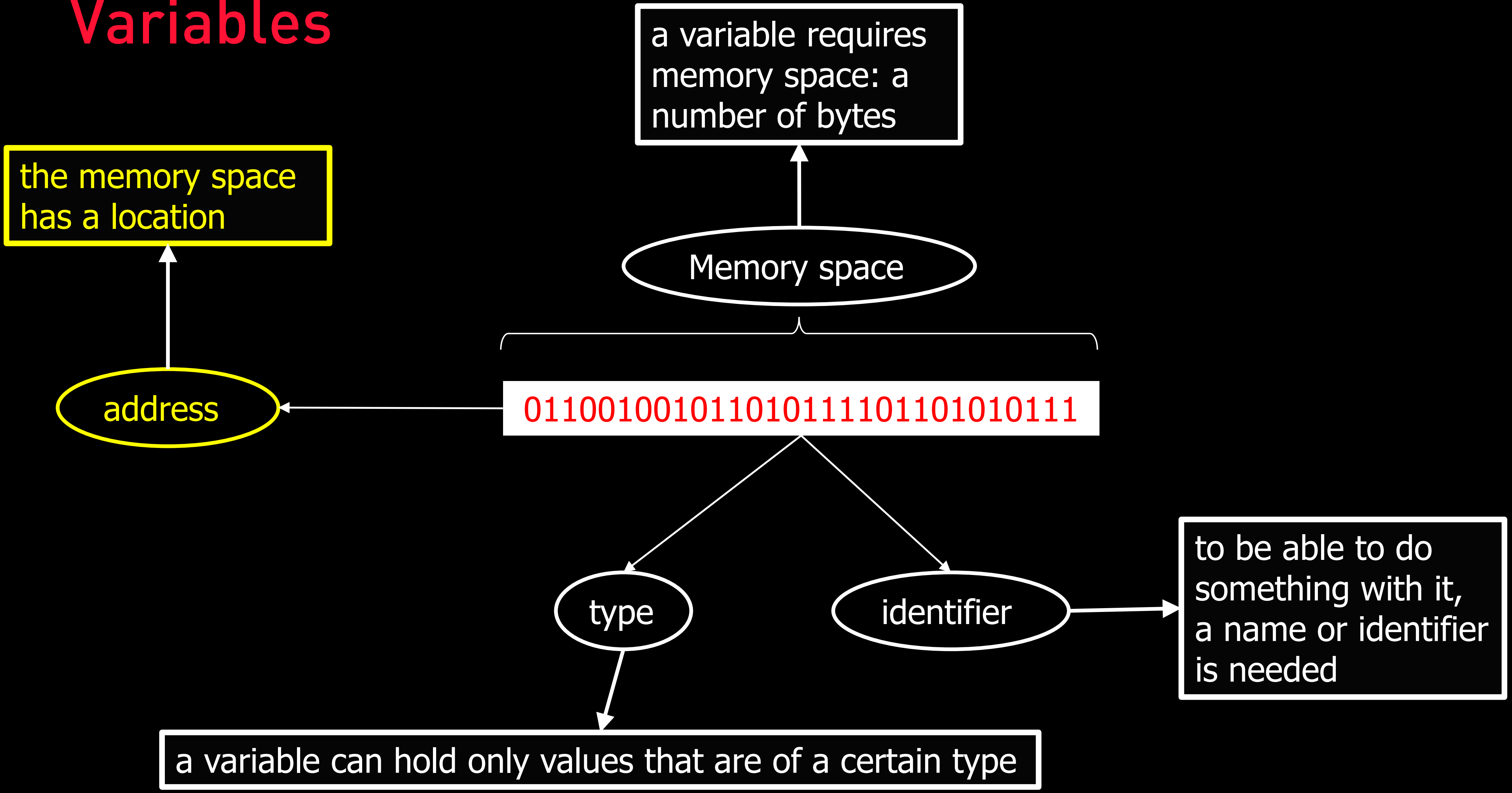
Memory cell 1

- 1 byte
- 8 bits

Memory cell 2

- 1 byte
- 8 bits

Variables



a variable requires memory space: a number of bytes

Memory space

the memory space has a location

address

011001001011010111101101010111

type

identifier

to be able to do something with it, a name or identifier is needed

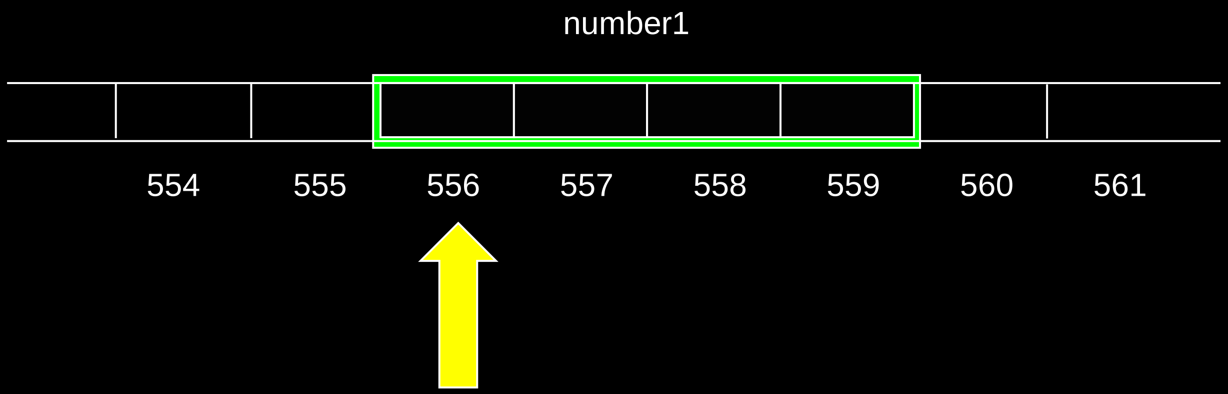
a variable can hold only values that are of a certain type

Memory address of a variable

- On declaration, memory cells are reserved
- Number of cells depends on the type
- The location or number of the first cell is “the memory address of the variable”
- This memory address can be stored in a pointer variable.

Memory address of a variable

- Name: number1
- Type: int
- Occupies memory cells 556, 557, 558, 559
- → the memory address of number1 is 556



Pointer as variable: "memory address variable"

- A pointer is a variable in which you can store a memory address.
- A *pointer* is a *memory address variable.*
- On declaration, use an asterisk between type and name:
- Example:

  ```
  int * pNumber{};
  ```

Naming convention: always use p as prefix for pointers

Content

- Array
- Pointer
  - Address of operator
  - Dereference operator
  - Pointer properties
- Arrays and pointers
- Functions

The “address of” operator: &

- Read as “The address of”
- &number1 → “The memory address of the number1 variable”
- The result can be stored in a pointer variable.
- Example:

```
int number1 { 42 };
int * pNumber1 { &number1 };
```

↓

pNumber1 contains the memory address of the variable number1

pNumber1 "points at" the address of the variable number1

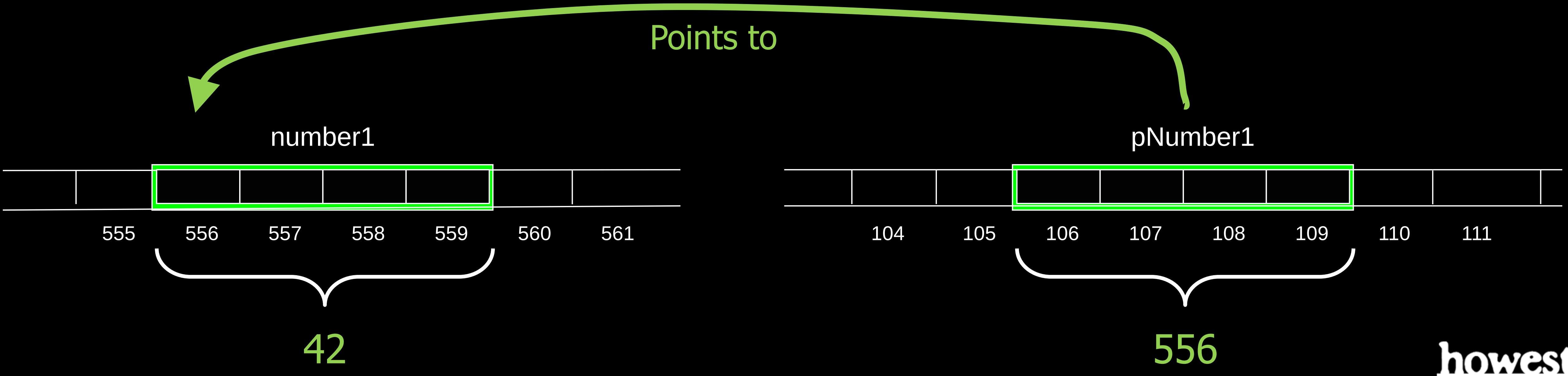
Example

```
int number1 { 42 };
```

- identifier: number1
- type : int
- value : 42
- address : 556

```
int *pNumber1 { &number1 };
```

- identifier: pNumber1
- type : int *
- value : address of number1



Content

- Array
- Pointer
  - Address of operator
  - Dereference operator
  - Pointer properties
- Arrays and pointers
- Functions

The dereference (contents of) operator: *

- Read as “the value of the variable to which the pointer refers”
- A dereferenced pointer evaluates to the contents of the address it is pointing to.

```
int number { 42 };
int *pSome = &number;
cout << number;
cout << pSome;
cout << *pSome;
```

The dereference (contents of) operator: *

- Read as "the value of the variable to which the pointer refers"
- A dereferenced pointer evaluates to the contents of the address it is pointing to.

```
int number { 42 };
int *pSome = &number;
cout << number; // prints 42
cout << pSome;
cout << *pSome;
```

The dereference (contents of) operator: *

- Read as “the value of the variable to which the pointer refers”
- A dereferenced pointer evaluates to the contents of the address it is pointing to.

```
int number { 42 };
int *pSome = &number;
cout << number; // prints 42
cout << pSome; // prints the address of number
cout << *pSome;
```

The dereference (contents of) operator: *

- Read as "the value of the variable to which the pointer refers"
- A dereferenced pointer evaluates to the contents of the address it is pointing to.

```
int number { 42 };
int *pSome = &number;
cout << number; // prints 42
cout << pSome; // prints the address of number
cout << *pSome; // prints the contents of the variable it
points to: 42
```

The dereference (contents of) operator: *

```
int number { 42 };
int *pSome { &number };
*pSome = 128;
cout << number; // prints ???
```

The dereference (contents of) operator: *

```
int number { 42 };
int *pSome { &number };
*pSome = 128;
cout << number; // prints 128

//128 is assigned to the variable pSome points to:
```

Content

- Array
- Pointer
  - Address of operator
  - Dereference operator
  - Pointer properties
- Arrays and pointers
- Functions

Pointer properties

- The type of the pointer must match the type of the variable being pointed to:

```
int number { 42 };
double pi { 3.1415926535 };
double *p1 { &number }; ->ERROR!!
double *p2 { &pi };     ->OK
```

- Why?

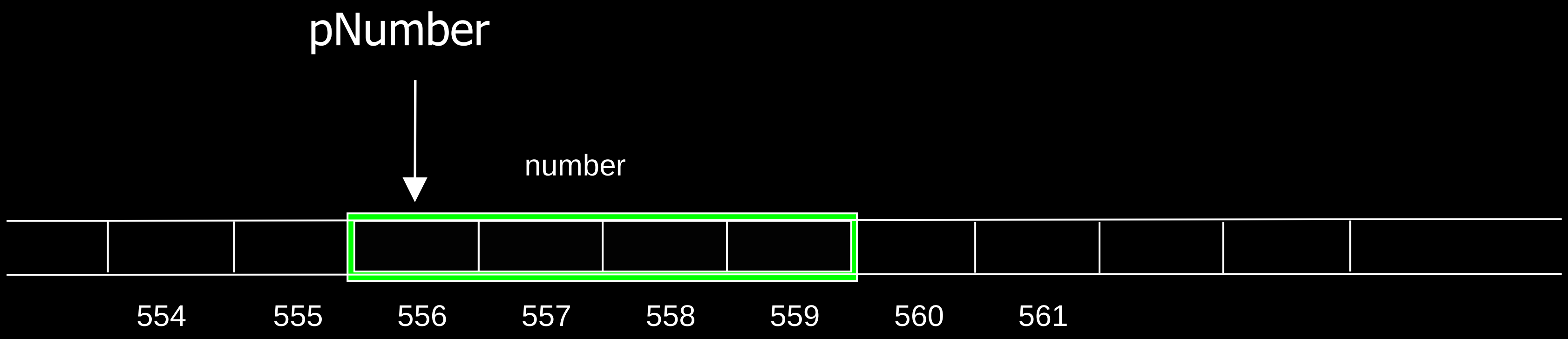
Pointer properties

- Remember:
- A pointer has two properties:
  1. The memory address of a variable.
  2. The type of the variable it refers to. → WHY?
- Why the type of the variable? → Pointer arithmetic:
  - Incrementing a pointer makes it point at the memory address right after the memory occupied by the variable

Incrementing a pointer

- Example:

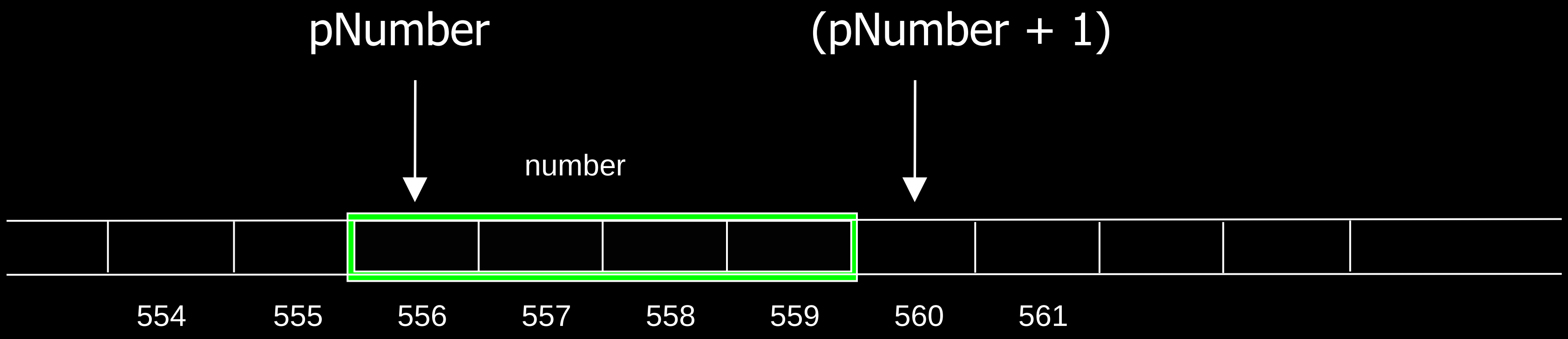
```
int number { 42 };
int *pNumber = &number;
```



Incrementing a pointer

- Example:

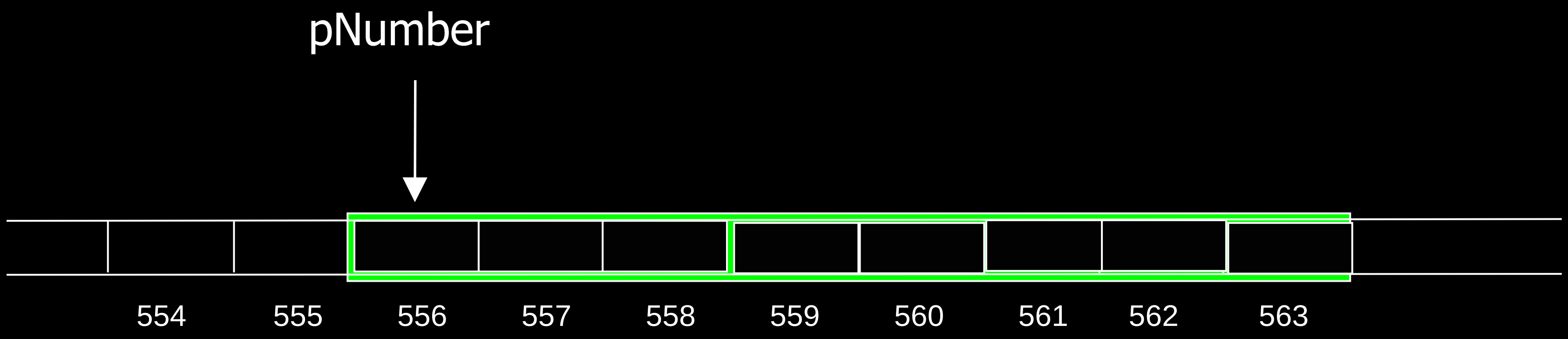
```
int number { 42 };
int *pNumber = &number;
++pNumber;
```



Incrementing a pointer

- Example:

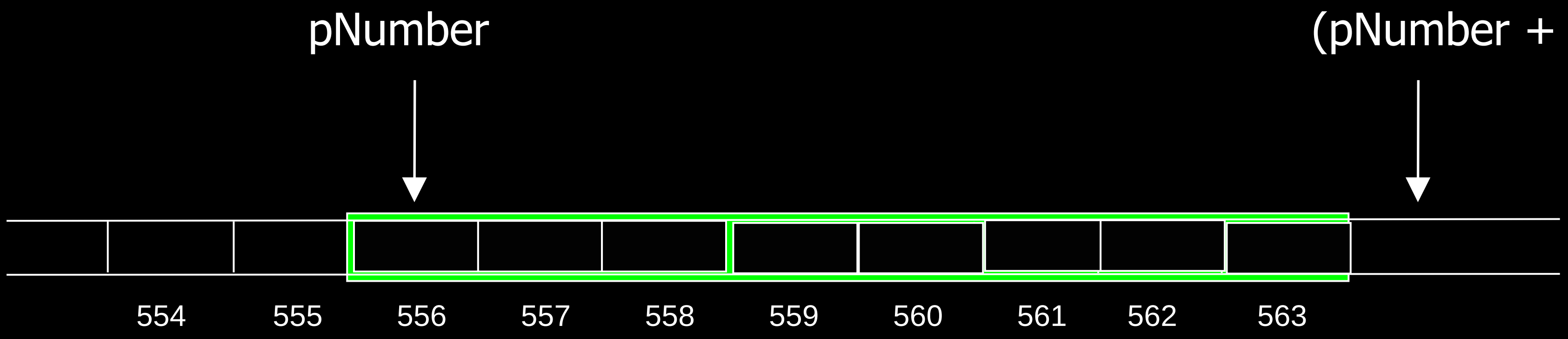
```
double number = 42;
double *pNumber = &number;
```



Incrementing a pointer

- Example:

```
double number = 42;
double *pNumber = &number;
++pNumber;
```

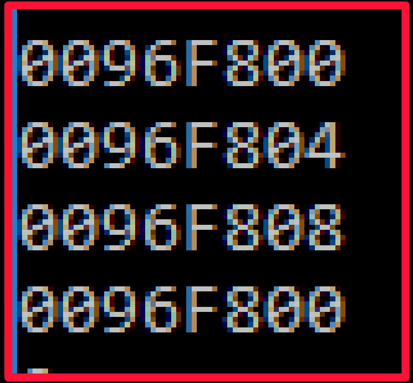


Content

- Array
- Pointer
- Arrays and pointers
- Functions

Addresses of elements of an array

```cpp
const int size{ 5 };
int numbers[size]{0, 1, 2, 3, 4};
std::cout << numbers[0] << '\n'; // prints the value at index 0
std::cout << &numbers[0] << '\n'; // address of first element
std::cout << &numbers[1] << '\n'; // address of second element
std::cout << &numbers[2] << '\n'; // address of third element
std::cout << numbers << '\n'; // prints the address of 1st el.
```



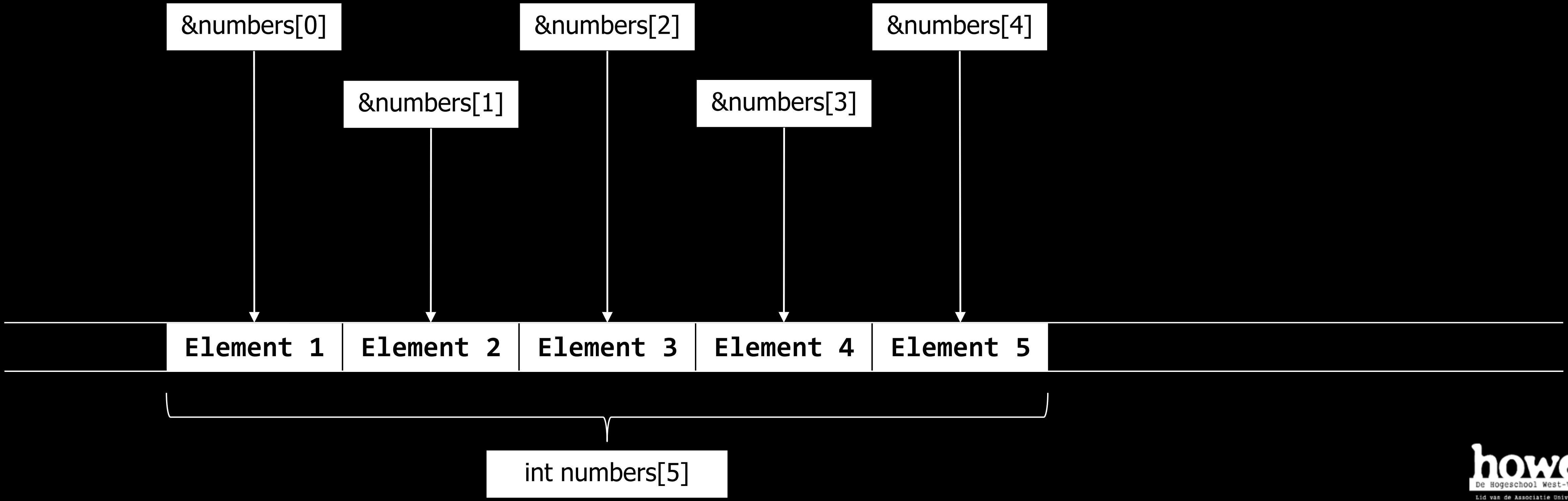
```
0096F800
0096F804
0096F808
0096F800
```

Conclusions:

- The addresses of the elements differ 4 bytes. This the size of the array element.
- In an array, the elements occupy adjacent memory space. → this is so by definition.
- The name of the array refers to the address of the first element of the array

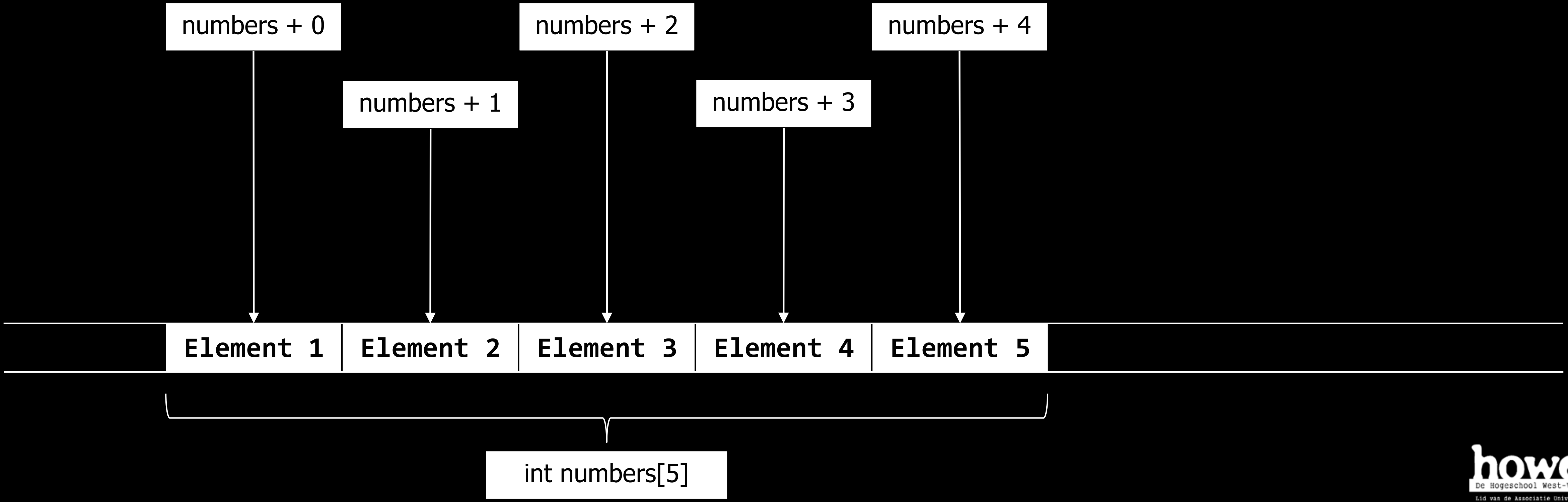
Addresses of elements of an array

- The addresses of the elements differ 4 bytes.



Addresses of elements of an array

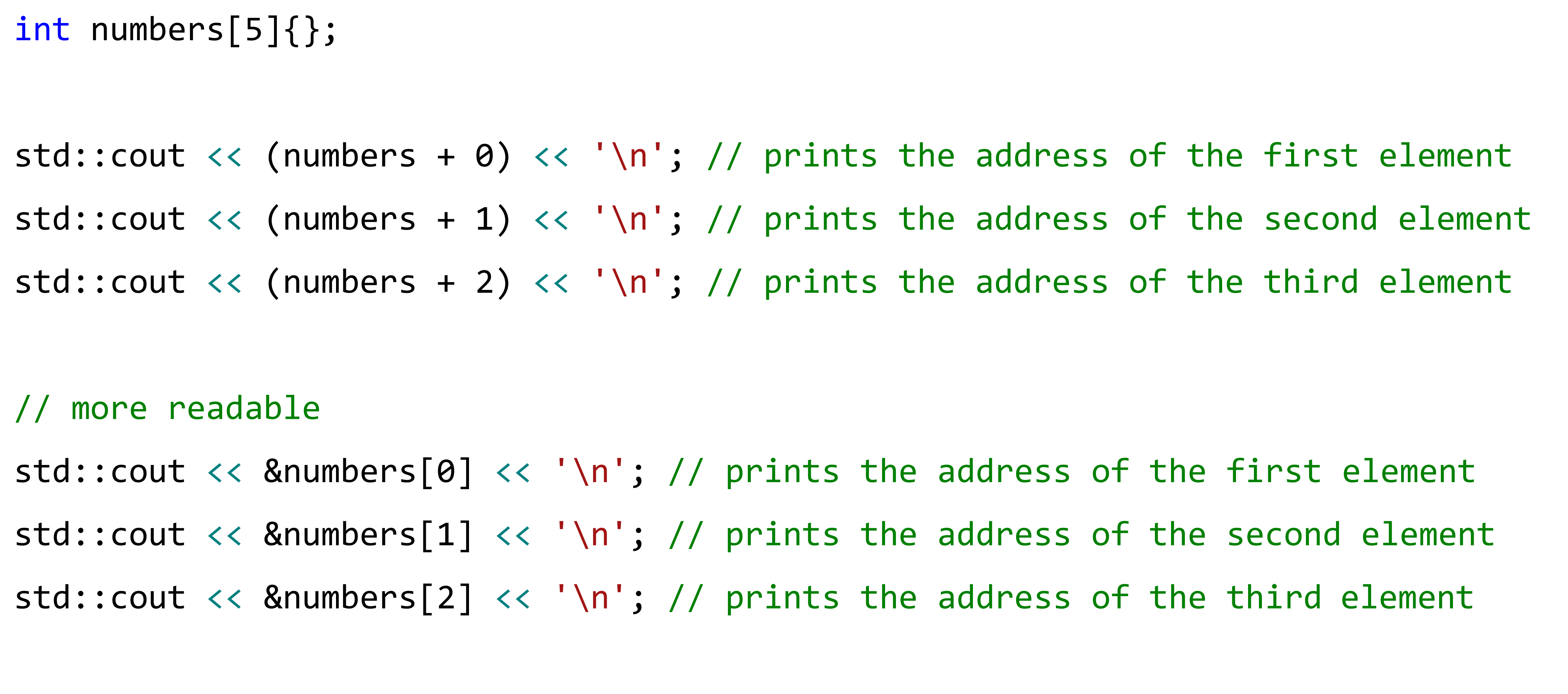
- The name of the array is the address of the first element of the array.
- Adding one to the array variable, adds the size of an element to that address.
- The index of an arrays starts with 0 because the index is actually the offset to the first element.



Values of elements in an array

```cpp
int numbers[5]{};

std::cout << *(numbers + 0) << '\n'; // prints the value of the first element
std::cout << *(numbers + 1) << '\n'; // prints the value of the second element
std::cout << *(numbers + 2) << '\n'; // prints the value of the third element

// more readable
std::cout << numbers[0] << '\n'; // prints the value of the first element
std::cout << numbers[1] << '\n'; // prints the value of the second element
std::cout << numbers[2] << '\n'; // prints the value of the third element
```

Addresses of elements in an array



```
int numbers[5]{};


std::cout << (numbers + 0) << '\n'; // prints the address of the first element
std::cout << (numbers + 1) << '\n'; // prints the address of the second element
std::cout << (numbers + 2) << '\n'; // prints the address of the third element


// more readable
std::cout << &numbers[0] << '\n'; // prints the address of the first element
std::cout << &numbers[1] << '\n'; // prints the address of the second element
std::cout << &numbers[2] << '\n'; // prints the address of the third element
```

pointer vs fixed array

```cpp
int numbers[5]{}, *pPointer{}, size{}, numElements{};
pPointer = numbers; // this copies the address of the first element in the pointer
size = sizeof(numbers);              // 20: size in bytes of the array
size = sizeof(int);                  // 4 : the size of the type of the elements
int numElements { sizeof(numbers) / sizeof(int) }; // 5
```

- An array variable knows its size in bytes
- This can be used to retrieve the number of elements in the array.
- The decayed array loses the information about its size!

Content

- Array
- Pointer
- Arrays and pointers
- Functions:
  - Passing an array as an argument
  - Passing a pointer by value

Passing arrays with functions

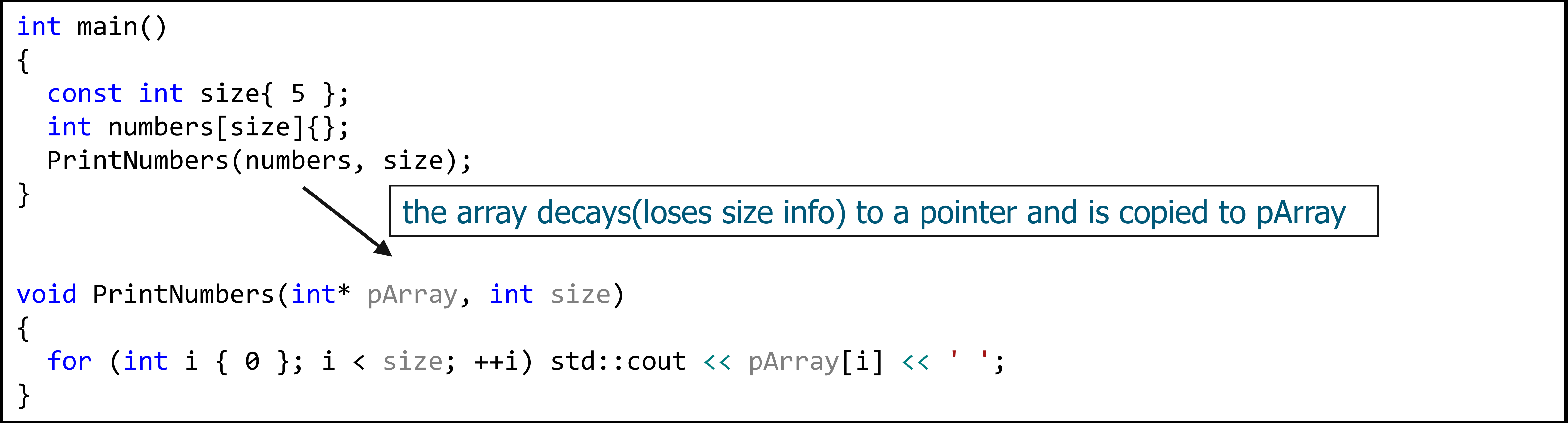
```
int main()
{
  const int size{ 5 };
  int numbers[size]{};
  PrintNumbers(numbers, size);
}
```

the array decays(loses size info) to a pointer and is copied to pArray

```
void PrintNumbers(int* pArray, int size)
{
  for (int i { 0 }; i < size; ++i) std::cout << pArray[i] << ' ';
}
```

- The array variable (numbers)contains the address of the first element of the array.
- The array decays (loses size info) to a pointer and when it is copied to pArray.

Passing arrays with functions

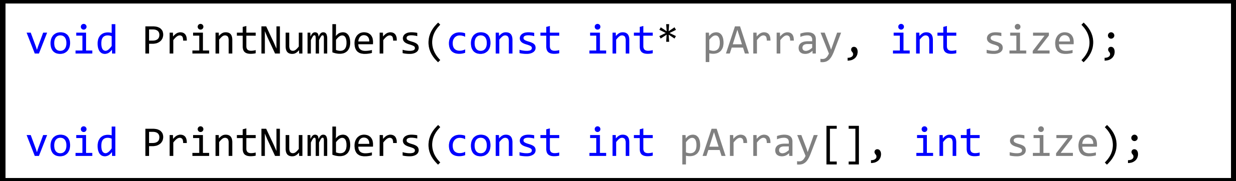


```
int main()
{
  const int size{ 5 };
  int numbers[size]{};
  PrintNumbers(numbers, size);
}


void PrintNumbers(int* pArray, int size)
{
  for (int i { 0 }; i < size; ++i) std::cout << pArray[i] << ' ';
}
```

the array decays(loses size info) to a pointer and is copied to pArray

- pArray is a plain pointer that points to the first element of the array.
- size is the only way to know the number of elements in the array in the function.
- The [ ] operator can be used to iterate over the different elements of the array.

Passing arrays with functions



```
void PrintNumbers(const int* pArray, int size);

void PrintNumbers(const int pArray[], int size);
```

- Two ways of declaring a pointer parameter:
  - As a pointer to the first element: *
  - As an array [ ]

- Compiler ignores the difference.

- A const qualifier can be used to protect the contents of the array from modifications.

Content

- Array
- Pointer
- Arrays and pointers
- Functions:
  - Passing an array as an argument
  - Passing a pointer by value

Passing pointers to functions

- Recap:
- Pass by value
- Pass by reference
- Pass by const reference

- Now: pass by pointer

Passing pointers to functions

```
void MyFunction(int * pValue);

int main()
{
  int score{ 10 };

  std::cout << "score: " << score << " address of score: " << &score << '\n'; // score is 10
  MyFunction(&score); // the memory address of score is passed to the function
  std::cout << "score: " << score << " address of score: " << &score << '\n'; // score is 20
}

void MyFunction(int * pValue)
{
  std::cout << "the value of the variable pValue refers to is: " << *pValue << '\n’;
  std::cout << "the address is: " << pValue << '\n’;
  *pValue = 20; // assign 20 to the variable pValue points at
}
```

Passing pointers to functions

- Passing a pointer to a function, enables us to modify the value of the variable it refers to.
- Pass by pointer was first used in C. It was the only way to modify a variable that was passed to a function.
- C++ added pass by reference to avoid pointer confusion and make the code easier to read.

- OpenGL and DirectX still use pass by pointer. (!)

Passing const pointers to functions

```
void MyFunction(const int * pValue);

int main()
{
  int score{ 10 };

  std::cout << "score: " << score << " address of score: " << &score << '\n'; // score is 10
  MyFunction(&score); // the memory address of score is passed to the function
  std::cout << "score: " << score << " address of score: " << &score << '\n'; // score is 20
}

void MyFunction(const int * pValue)
{
  std::cout << "the value of the variable pValue refers to is: " << *pValue << '\n’;
  std::cout << "the address is: " << pValue << '\n’;
  *pValue = 20; // error: the const modifier prevents modifying the variable pValue points at
}
```

Passing const pointers to functions

- Passing a const pointer to a function, is similar to pass by const reference.
- Same advantages as pass by const refence.
  - E.g. Avoid making a copy of the large object
- C++ added pass by const reference to avoid pointer confusion and make the code easier to read.
- Some api's that are C compatible such as OpenGL and DirectX still use pass by (const) pointer. (!)