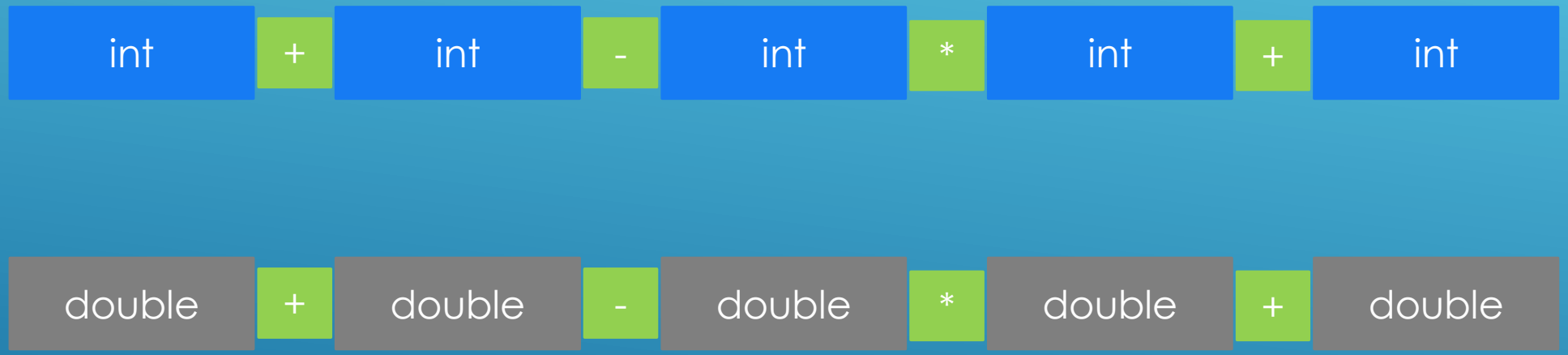
Slides

Section :Data conversions

Slide intentionally left empty

Data Conversions, Overflow and underflow

Technically , all variables in an expression should be of the same type


int + int - int * int + int
double + double - double * double + double


int
+
double
-
int
*
double
+
int

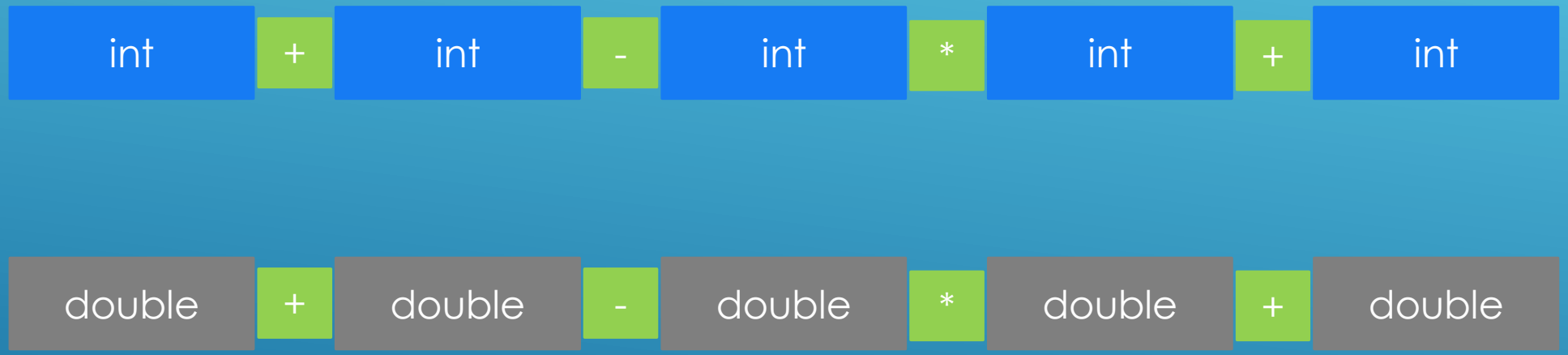
Overflow and Underflow

```
//Overvlow and overflow.

unsigned char char_var {55};
char_var = 261; // Store in more than can fit in memory
char_var = -1;  // Store in a negative number
```

Slide intentionally left empty

Implicit Data Conversions

Technically , all variables in an expression should be of the same type

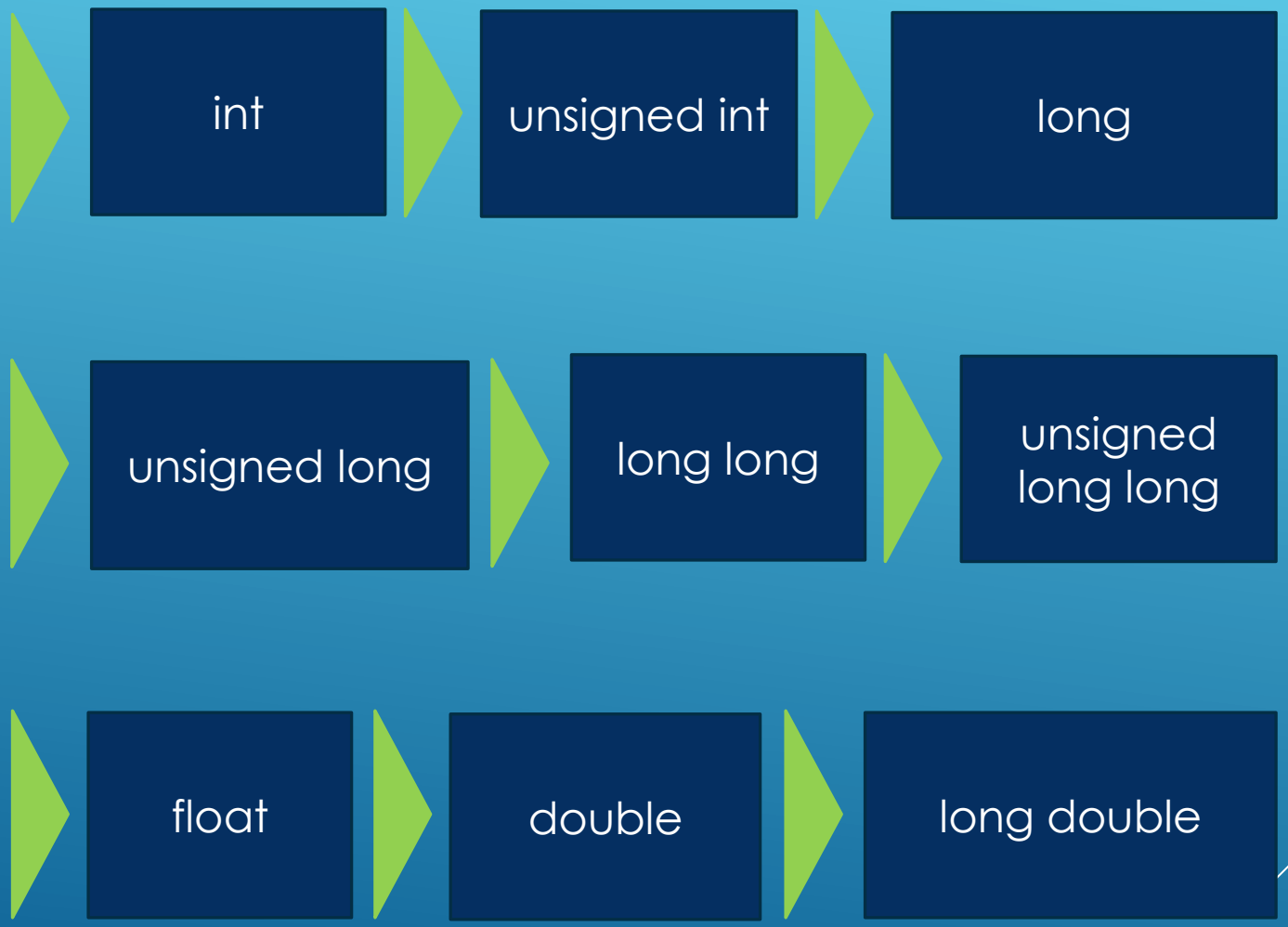

int + int - int * int + int
double + double - double * double + double


int
+
double
-
int
*
double
+
int

Conversions done by the compiler without your involvement

```
//      . The compiler applies implicit conversions
//          when types are different in
//          an expression
//      . Conversions are always done from the smallest
//          to the largest type in this case int is
//          transformed to double before the expression
//          is evaluated.Unless we are doing an assignment

double price { 45.6 };
int units {10};

double total_price = price * units;

std::cout << "Total price : " << total_price << std::endl;
std::cout << "sizeof total_price : " << sizeof(total_price) << std::endl;
```

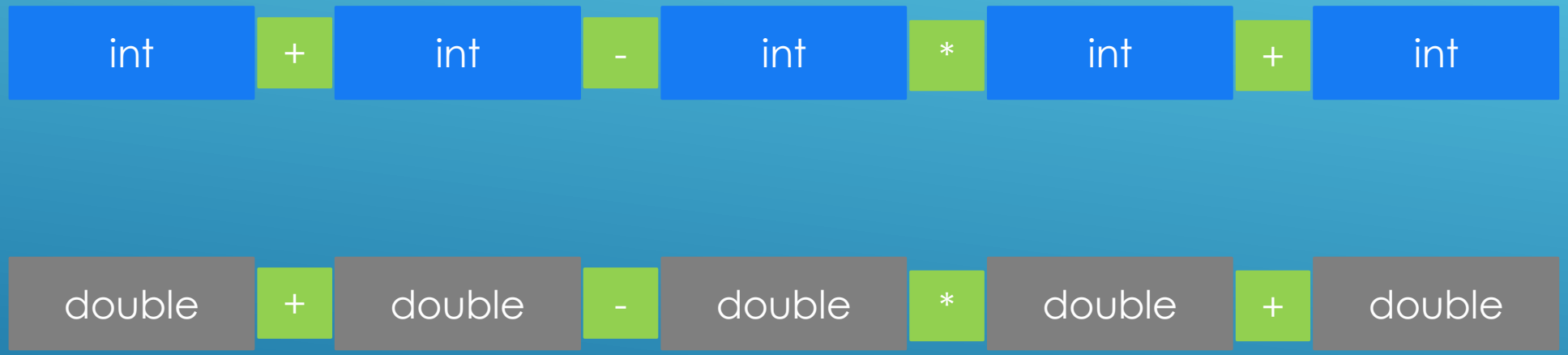
Conversion Guideline



```cpp
//Implicit conversions in assignments

// The assignment operation is going to cause an implicit
// narrowing conversion , y is converted to int before assignment
int x;
double y {45.44};
x = y;
std::cout << "The value of x is : " << x << std::endl;
```

Slide intentionally left empty

Explicit Data Conversions


int + int - int * int + int
double + double - double * double + double


int
+
double
-
int
*
double
+
int

Conversions are actively initiated by YOU! The developer.

Implicit cast

```
//Implicit cast will add up the doubles,
//then turn result into int for assignment
double x { 12.5 };
double y { 34.6};

int sum = x + y;

std::cout << "The sum  is : " << sum << std::endl;
```

static_cast<>()

```
double x { 12.5 };
double y { 34.6};

//Explicity cast : cast then sum up
sum = static_cast<int>(x) + static_cast<int>(y) ;

std::cout << "Cast then sum, result : " << sum << std::endl;
```

static_cast<>()

```
double x { 12.5 };
double y { 34.6};

int sum ;

//Explicit cast : sum up then cast, same thing as implicit cast
sum =  static_cast<int> (x + y);
std::cout << "Sum up then cast, result : " << sum << std::endl;
```

Old style C-Cast

```
//Old style C-cast
double PI {3.14};

int int_pi = (int)(PI);
std::cout << "PI : " << PI << std::endl;
```

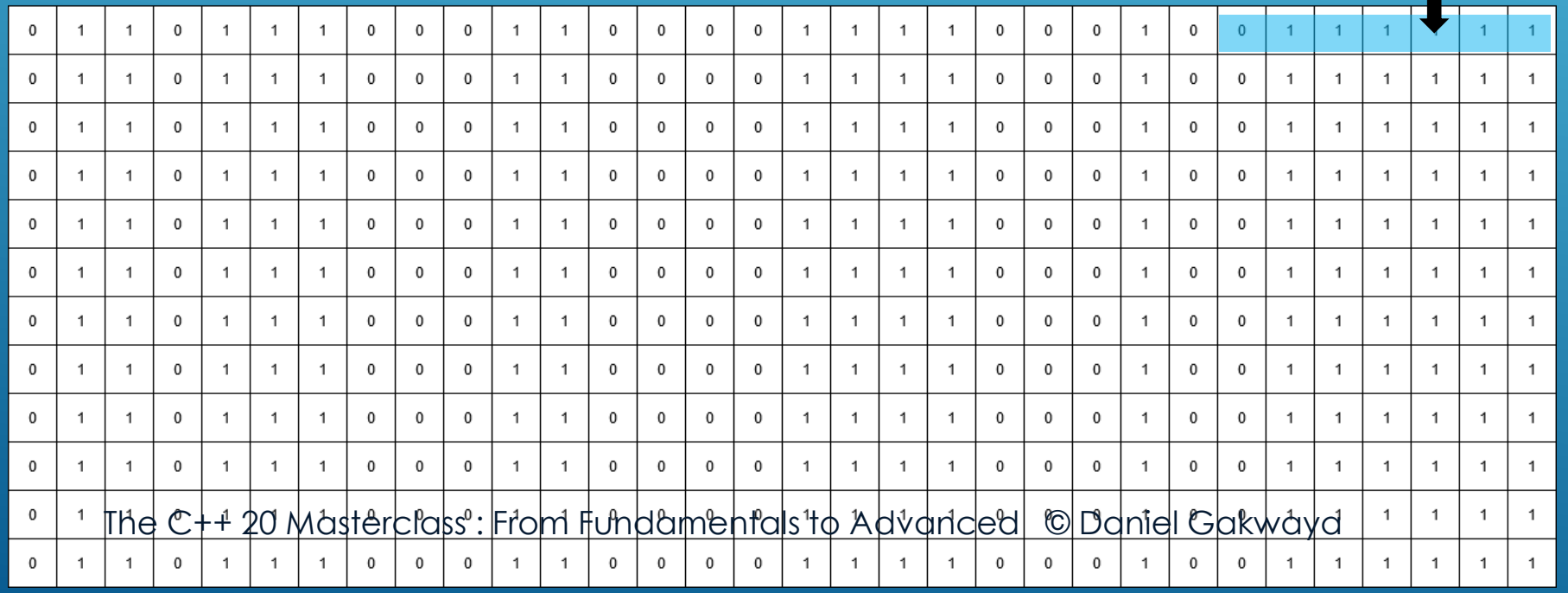
Prefer C++ casts over C- style casts

- C++ casts , static_cast<> is jut one of them, make your intent very clear
- They are easy to search for in code
- Static_cast<>() is checked by the compiler, and if the types are not compatible, you'll get a nice compiler error

Slide intentionally left empty

Overflow and Underflow

```
unsigned char char_var {55};
```




1byte
0 ~ 255

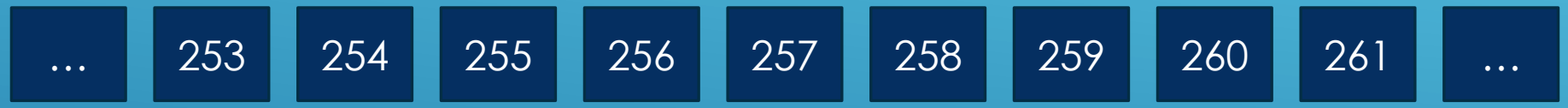
Overflow

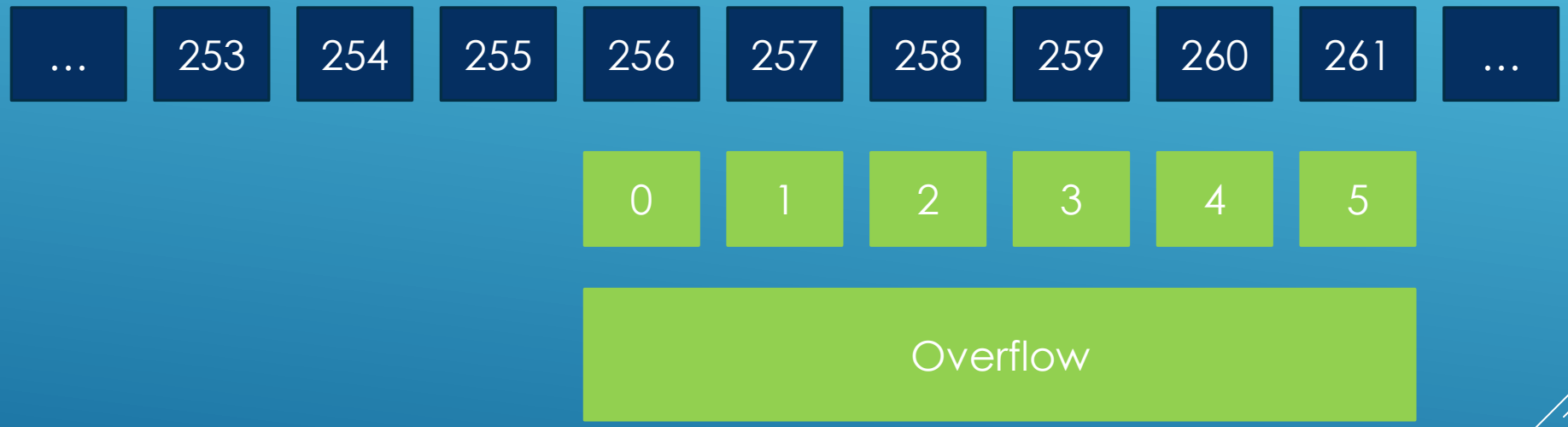
```
unsigned char char_var {55};

unsigned char val1 {130};
unsigned char val2 {131};

char_var = val1 + val2;

std::cout << "char_var : " << static_cast<int>(char_var) << std::endl;
```

Overflow

...
11111101 (253 dec)
11111110 (254 dec)
11111111 (255 dec)

Reset all bits
00000000 (000 dec)
00000001 (001 dec)

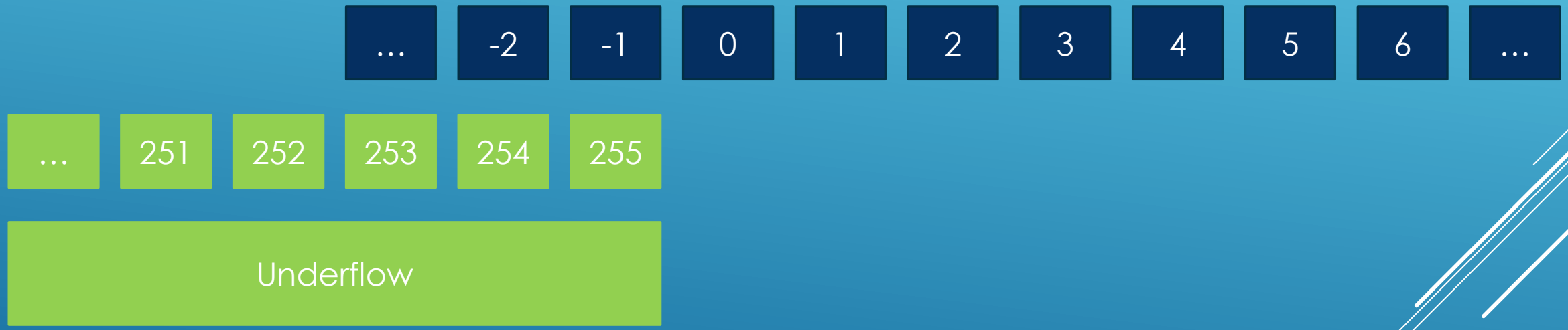

...
253
254
255
256
257
258
259
260
261
...


...
253
254
255
256
257
258
259
260
261
...
0
1
2
3
4
5
Overflow

Underflow

```
unsigned char char_var {55};

unsigned char val1 {130};
unsigned char val2 {131};

char_var = val1 - val2; // Underflow
std::cout << "char_var (exp -1) : " << static_cast<int>(char_var) << std::endl;
```


...
-2
-1
0
1
2
3
4
5
6
...


...
-2
-1
0
1
2
3
4
5
6
...
...
251
252
253
254
255
Underflow

```
unsigned char char_var {55};
char_var = 261; // Store in more than can fit in memory : Overflow
char_var = -1;  // Store in a negative number : Underflow
```

General Thoughts

- Overflow and underflow are not only limited to char or integral types in general.
- It happens even on floating types, it was just easier to demonstrate it using the smallest possible integral type we can get our hands on : char. We just made it unsigned to , again, make it easier to demonstrate underflow
- As a general guideline, always be mindful of the valid range for the values you assign to the fundamental types we've learnt about. If in doubt, the <numeric> header has utilities you can use to query the limits for your types
- The compiler will sometimes throw warnings about overflows and underflows in your code. Keep an eye out for that

Slide intentionally left empty