Hello, World!

Stroustrup 2.2.1

First Program

```
// smallest C++ program
int main() {
}
```

https://godbolt.org/z/db9oP8EM1

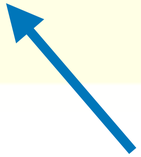
Every C++ program must contain exactly one function named main.

This is where execution of the program starts.

First Program

```
// smallest C++ program
int main() {
}
```

the return value



this main takes no arguments

Better First Program

```
// second smallest C++ program
int main() {
    return 0;
}
```

Now we actually return an integer.

A return value of 0 indicates success.

Nonzero return values can encode different types of errors.

It is a good habit to always include the return 0 statement.

Hello World

```cpp
#include <iostream>

// Hello World
int main() {
    std::cout << "Hello World\n";
    return 0;
}
```

https://godbolt.org/z/1nn6G5dar

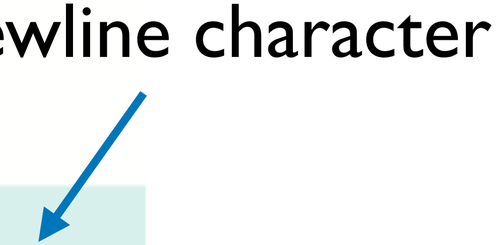
We can use a `cout` statement to write to the console.

`cout` is the standard output stream and is defined in the `iostream` library.

Hello World

```
#include <iostream>

// Hello World
int main() {
    std::cout << "Hello World\n";
    return 0;
}
```



https://godbolt.org/z/1nn6G5dar

We can specify a string between double quotes.

The << is the insertion operator. It looks like a mouth shouting the string.

Hello Again

```
// Hello Again
int main() {
    std::cout << "Hello" << " World\n";
    return 0;
}
```

https://godbolt.org/z/1s46Tzedz

We can concatenate outputs with the insertion operator.

Single characters can also be used, between single quotes.

```
std::cout << "Hello" << ' ' << "World\n";
```

Built-In Types

Stroustrup 2.2.2

Types

C++ offers some basic built-in types.

`bool`	a boolean value, `true` or `false`.
`char`	a character, for example ‘a’, ‘3’ or ‘\n’.
`int`	an integer like 123 or -4000.
`double`	a double precision floating point number like 2.71828.
`float`	single precision floating point number.

Size

How much memory a type uses can vary from machine to machine.

Typically:

`bool` and `char` use 8 bits or 1 byte.

`int` uses 4 bytes. This means an `int` can be in the range $[-2^{31}, 2^{31} - 1]$

`double` uses 8 bytes.

`float` uses 4 bytes.

You can check this with the `sizeof` operator.

```
std::cout << sizeof(double);
```

The answer is given as a multiple of the size of the `char` type.

Initializing Variables

Stroustrup 2.2.2

Variables

```
int main() {
    int x = 5;
    std::cout << "the value of x is " << x << '\n';
    return 0;
}
```

https://godbolt.org/z/bjeM6M15M

This prints out: the value of x is 5

The line `int x = 5` does two things:

- declares the name `x` to stand for an object of type `int` .
- initializes `x` with value 5.

Watch Out! UB

```
int main() {
    int x;
    // Don't do this!  The next line has undefined behavior.
    std::cout << "the value of x is " << x << '\n';
    return 0;
}
```

Here we declare the name `x` to stand for an `int` but we do not initialize it.

The compiler does not automatically initialize it for us. The value of `x` is whatever was already written to its spot in memory.

The result is undefined behavior (UB).

Watch Out! UB

```
int main() {
    int x;
    // Don't do this!  The next line has undefined behavior.
    std::cout << "the value of x is " << x << '\n';
    return 0;
}
```

The compiler does not automatically initialize it for us. The value of `x` is whatever was already written to its spot in memory.

This is done for performance reasons.

Best Practice: “Don’t introduce a name until you have a value for it”.

—Stroustrup

Boolean Example

```
int main() {
    bool b = true;
    // std::cout << std::boolalpha;
    std::cout << "This course is great: " << b << '\n';
    return 0;
}
```

https://godbolt.org/z/PnrzMP7rx

Mild surprise, this prints out: This course is great: 1

If you want to print out true/false instead of 0/1, use the commented out line.

Statically Typed

C++ is a statically typed language.

```
// Error! This code does not compile
int x = 5;
x = "hello";
```

Once `x` is declared to be an `int`, we cannot assign it to a value that can't be interpreted as an `int` .

Implicit Conversions

If types don't match, the compiler may do an implicit conversion in order to make it work.

```
int main() {
    int x = 5;
    // this code compiles with no complaints
    x = 3.14;
    std::cout << "the value of x is " << x << '\n';
    return 0;
}
```

https://godbolt.org/z/sY53jMr7n

In this case, an implicit conversion is done with no warning!

The value of `x` becomes 3.

Terminology

```
int main() {
    int x = 5;
    x = 3;
    return 0;
}
```

The first line initializes `x` with the value 5.

The second line assigns `x` the value 3.

Implicit Conversions

If types don't match, the compiler may do an implicit conversion in order to make it work.

```
int main() {
    bool b = 53;
    int x = 3.14;
    std::cout << std::boolalpha;
    std::cout << "b is " << b << " and x is " << x << '\n';
    return 0;
}
```

https://godbolt.org/z/6o5jqvoxT

This prints: `b is true and x is 3`

Implicit Conversions

`bool`

Zero is interpreted as `false`. Any non-zero numeric value is interpreted as `true`.

`int`

A floating point number is truncated towards zero.

Example:

```
int x = 3.99;    // x is 3
int y = -3.99;   // y is -3
```

Explicit Conversions

Best practice: if you want to use a conversion make it explicit.

We can do this with a cast.

```
int main() {
    bool b = static_cast<bool>(53);
    int x = static_cast<int>(3.14);
    std::cout << std::boolalpha;
    std::cout << "b is " << b << " and x is " << x << '\n';
    return 0;
}
```

https://godbolt.org/z/1dPezGdGT

This tells people "I intend to do this conversion."

Brace Initialization

Initialization in C++ is surprisingly complicated.

Brace initialization was introduced in C++11 as a uniform initialization method.

```
int main() {
    bool b {true};
    int x {3};
    std::cout << std::boolalpha;
    std::cout << "b is " << b << " and x is " << x << '\n';
    return 0;
}
```

https://godbolt.org/z/Kd6coTf6q

We put the value used to initialize the variable between braces.

Brace Initialization: default

Empty brackets initialize to a default value.

```
int main() {
    bool b {};
    int x {};
    std::cout << std::boolalpha;
    std::cout << "b is " << b << " and x is " << x << '\n';
    return 0;
}
```

https://godbolt.org/z/onETrP6Pb

A `bool` defaults to false.

An `int` defaults to zero.

Brace Initialization: conversion

A nice property of brace initialization is that it does not allow conversions.

```
bool b {53}
```

this does not compile.

error: narrowing conversion of 53 from int to bool

```
int x {3.14}
```

this does not compile.

error: narrowing conversion of 3.14 from double to int

Best Practice: Use brace initialization whenever possible.

Tests and Loops

Stroustrup 2.2.4

Tests

```
bool accept() {
    std::cout << "Do you want to proceed (y or n)?\n";
    char answer {};          // initializes with value 0
    std::cin >> answer;
    if (answer == 'y') {
        std::cout << "yay\n";
        return true;
    }
    return false;
}

int main() {
    accept();
    return 0;
}
```

https://godbolt.org/z/3jxrhc51s

Example adapted from Stroustrup 2.2.4

Control code chart [edit]

Binary	Oct	Dec	Hex	Abbreviation			Unicode Control Pictures[b]	Caret notation[c]	C escape sequence[d]	Name (1967)
				1963	1965	1967				
000 0000	000	0	00	NULL	NUL		NUL	^@	\0	Null
000 0001	001	1	01	SOM	SOH		SOH	^A		Start of Heading
000 0010	002	2	02	EOA	STX		STX	^B		Start of Text
000 0011	003	3	03	EOM	ETX		ETX	^C		End of Text
000 0100	004	4	04	EOT			EOT	^D		End of Transmission
000 0101	005	5	05	WRU	ENQ		ENQ	^E		Enquiry
000 0110	006	6	06	RU	ACK		ACK	^F		Acknowledgement
000 0111	007	7	07	BELL	BEL		BEL	^G	\a	Bell
000 1000	010	8	08	FE0	BS		BS	^H	\b	Backspace[e][f]
000 1001	011	9	09	HT/SK	HT		HT	^I	\t	Horizontal Tab[g]
000 1010	012	10	0A	LF			LF	^J	\n	Line Feed
000 1011	013	11	0B	VTAB	VT		VT	^K	\v	Vertical Tab
000 1100	014	12	0C	FF			FF	^L	\f	Form Feed
000 1101	015	13	0D	CR			CR	^M	\r	Carriage Return[h]

ASCII table from https://en.wikipedia.org/wiki/ASCII

Tests

```
bool accept() {
    std::cout << "Do you want to proceed (y or n)?\n";
    char answer {};          // initializes with value 0
    std::cin >> answer;
    if (answer == 'y') {
        std::cout << "yay\n";
        return true;
    }
    return false;
}

int main() {
    accept();
    return 0;
}
```

https://godbolt.org/z/3jxrhc51s

Example adapted from Stroustrup 2.2.4

Tests

```
bool accept() {
    std::cout << "Do you want to proceed (y or n)?\n";
    char answer {};          // initializes with value 0
    std::cin >> answer;
    if (answer == 'y') {
        std::cout << "yay\n";
        return true;
    }
    return false;
}

int main() {
    accept();
    return 0;
}
```

https://godbolt.org/z/3jxrhc51s

Example adapted from Stroustrup 2.2.4

Loops

```
bool accept() {
    std::cout << "Do you want to proceed (y or n)?\n";
    char answer {};          // initializes with value 0
    int tries {0};
    while (tries < 4) {
        std::cin >> answer;
        if (answer == 'y') {
            std::cout << "yay\n";
            return true;
        } else if (answer == 'n') {
            std::cout << "bye bye\n";
            return false;
        }
        ++tries;
    }
    return false;
}
```

https://godbolt.org/z/41d43q3jK

Loops

```
bool accept() {
    std::cout << "Do you want to proceed (y or n)?\n";
    char answer {};          // initializes with value 0
    int tries {0};
    while (tries < 4) {
        std::cin >> answer;
        if (answer == 'y') {
            std::cout << "yay\n";
            return true;
        } else if (answer == 'n') {
            std::cout << "bye bye\n";
            return false;
        }
        ++tries;
    }
    return false;
}
```

https://godbolt.org/z/41d43q3jK

Loops

```
bool accept() {
    std::cout << "Do you want to proceed (y or n)?\n";
    char answer {};          // initializes with value 0
    for (int tries {0}; tries < 4; ++tries) {
        std::cin >> answer;
        if (answer == 'y') {
            std::cout << "yay\n";
            return true;
        } else if (answer == 'n') {
            std::cout << "bye bye\n";
            return false;
        }
    }
    return false;
}
```

https://godbolt.org/z/Khndzhrvs

Question

Do you prefer the while loop solution or the for loop solution?

While Loop

```
bool accept() {
    std::cout << "Do you want to proceed (y or n)?\n";
    char answer {};          // initializes with value 0
    int tries {0};
    while (tries < 4) {
        std::cin >> answer;
        if (answer == 'y') {
            std::cout << "yay\n";
            return true;
        } else if (answer == 'n') {
            std::cout << "bye bye\n";
            return false;
        }
        ++tries;
    }
    return false;
}
```

In the while loop, `tries` is still in scope after the while loop ends.

For Loop

```
bool accept() {
    std::cout << "Do you want to proceed (y or n)?\n";
    char answer {};          // initializes with value 0
    for (int tries {0}; tries < 4; ++tries) {
        std::cin >> answer;
        if (answer == 'y') {
            std::cout << "yay\n";
            return true;
        } else if (answer == 'n') {
            std::cout << "bye bye\n";
            return false;
        }
    }
    return false;
}
```

The scope of `tries` is limited to the for loop itself.

Warnings

What does this code do?

```
int main() {
    int x = 0;
    if (x = 1) {
        std::cout << "Hello, World!\n";
    }
    return 0;
}
```

https://godbolt.org/z/P91hdec8K

What does this code do?

```
int main() {
    int x = 0;
    if (x = 1) {
        std::cout << "Hello, World!\n";
    }
    return 0;
}
```

https://godbolt.org/z/P91hdec8K

This code prints out Hello, World!

Compile with Warnings

```
int main() {
    int x = 0;
    if (x = 1) {
        std::cout << "Hello, World!\n";
    }
    return 0;
}
```

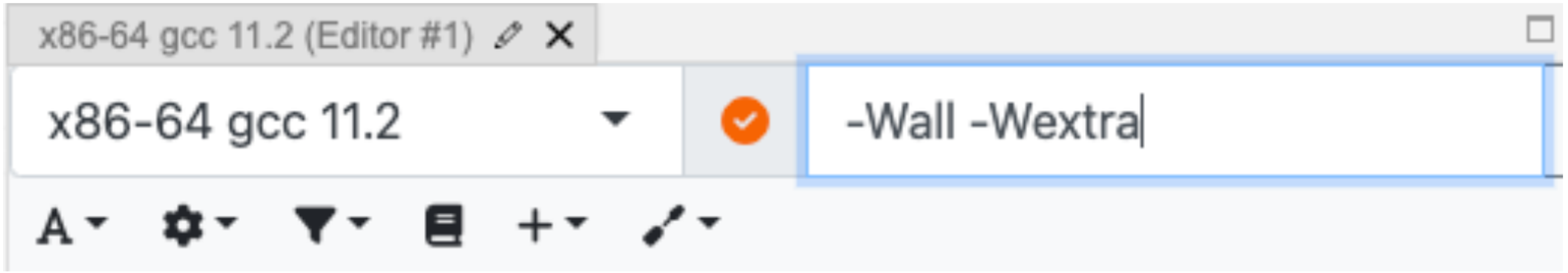
Typos happen. Get the compiler to help you out!

In this course we will always compile with the flags `-Wall -Wextra`

```
g++ -std=c++20 -Wall -Wextra main.cpp -o main
```

Compile with Warnings

Godbolt has a box where we can add compiler flags.



https://godbolt.org/z/1Yzza74hc

When we compile with these warnings the compiler tells us:

warning: suggest parentheses around assignment used as truth value.

Compile with Warnings

```
int main() {
    int x = 0;
    if ((x = 1)) {
        std::cout << "Hello, World!\n";
    }
    return 0;
}
```

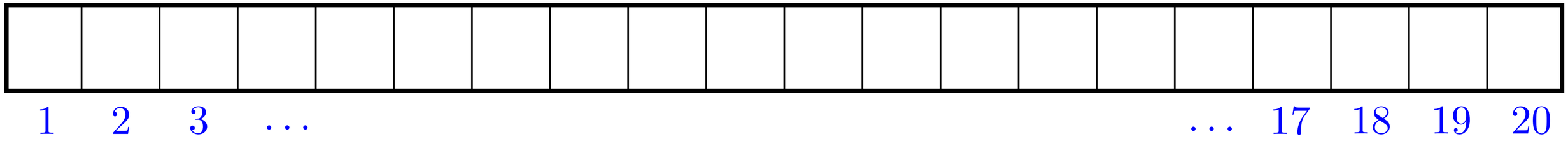
https://godbolt.org/z/9xGssPvrx

Now this compiles without a warning.

Pointers

Stroustrup 2.2.4

Memory



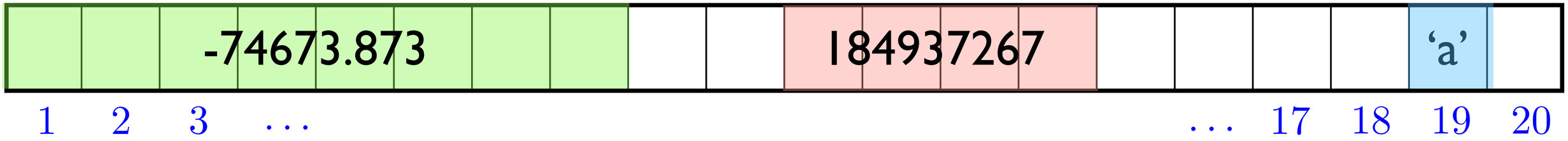
Imagine the memory as a long tape consisting of cells.

Each cell can hold 1 byte = 8 bits of information.

Each cell is labeled by an integer address.

On a 64 bit processor, memory addresses are 64 bit integers.

Memory

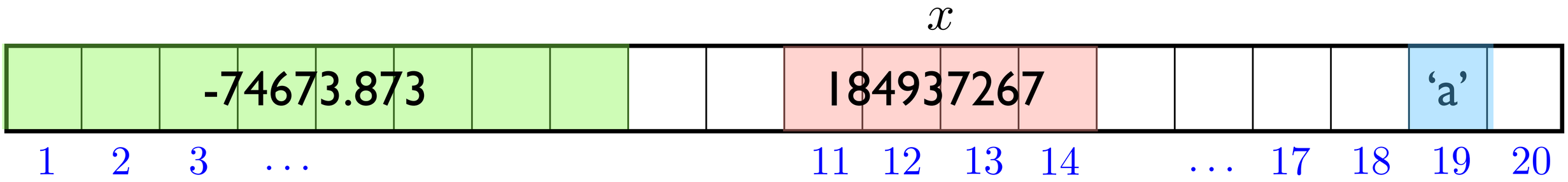


A `char` typically takes up a single cell.

An `int` typically takes up four cells.

A `double` typically takes up eight cells.

Address Operator

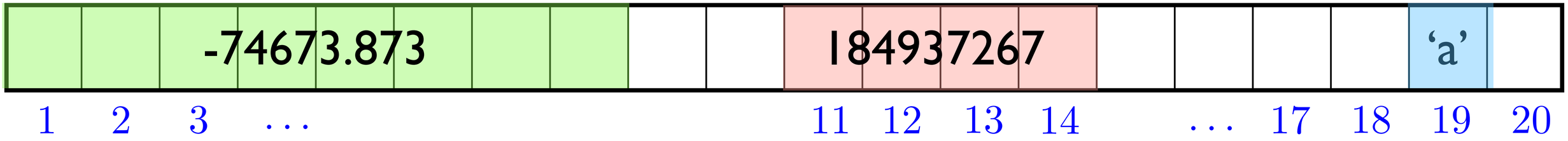


The `int` 184937267 sits in cells 11 through 14.

We can see the address of a variable with the $\&$ operator.

This will return the address of the first byte the variable occupies, i.e. address 11 in our `int` example.

Address Operator



We can see the address of a variable with the & operator.

```
int main() {
    int x = 184937267;
    std::cout << &x << '\n';
    return 0;
}
```

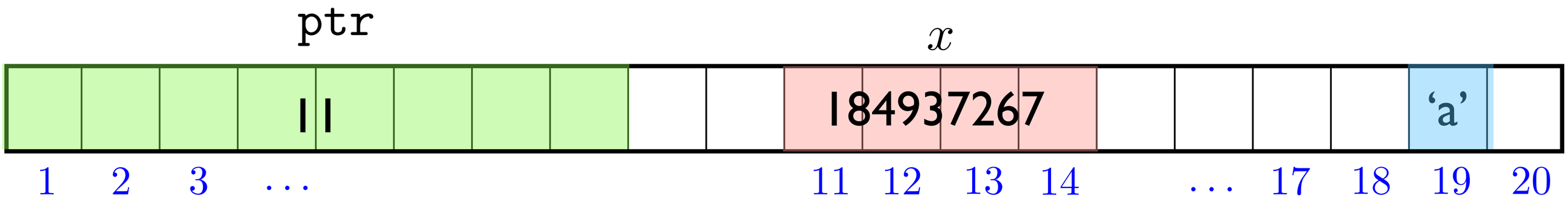
https://godbolt.org/z/1M9Mv1W39

This will print out an address in hexadecimal like 0x7fffd34d1ffc.

The 0x indicates it is in hex, then 12 hex digits follow.

A hex digit can hold 4 bits of info.

Pointers



A pointer is a variable holding an address.

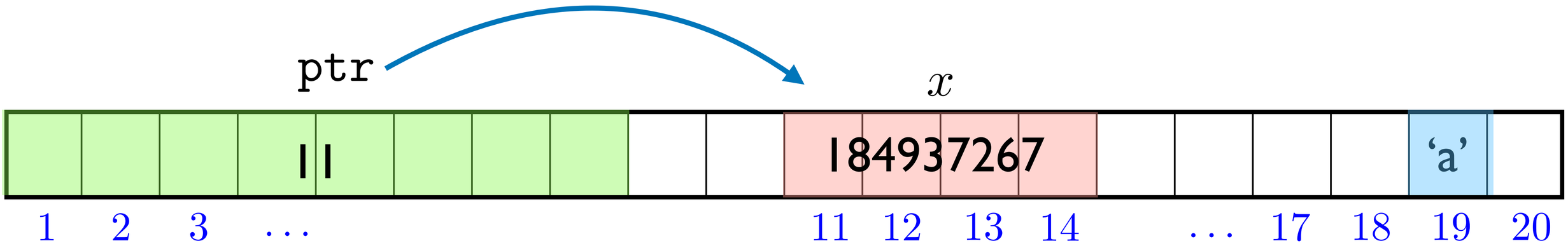
```
int main() {
    int x = 184937267;
    int* ptr = &x;
    std::cout << "ptr is " << ptr << '\n';
    std::cout << sizeof(ptr) << '\n';
    return 0;
}
```

https://godbolt.org/z/cxsa8Ke6M

If the pointer holds the address of an `int` its type is `int*` .

The star can go anywhere:

```
int* ptr
int * ptr
int *ptr
```



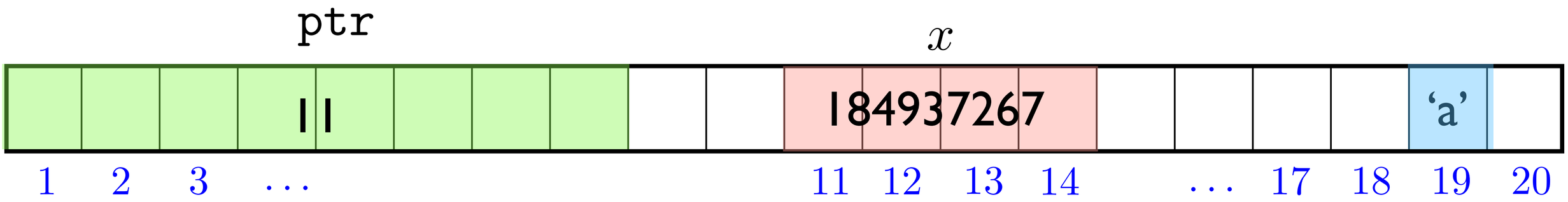
A pointer is a variable holding an address.

```
int main() {
    int x = 184937267;
    int* ptr = &x;
    std::cout << "ptr is " << ptr << '\n';
    std::cout << sizeof(ptr) << '\n';
    return 0;
}
```

https://godbolt.org/z/cxsa8Ke6M

The name pointer arises because we think that it points at the memory location that is its value.

Question



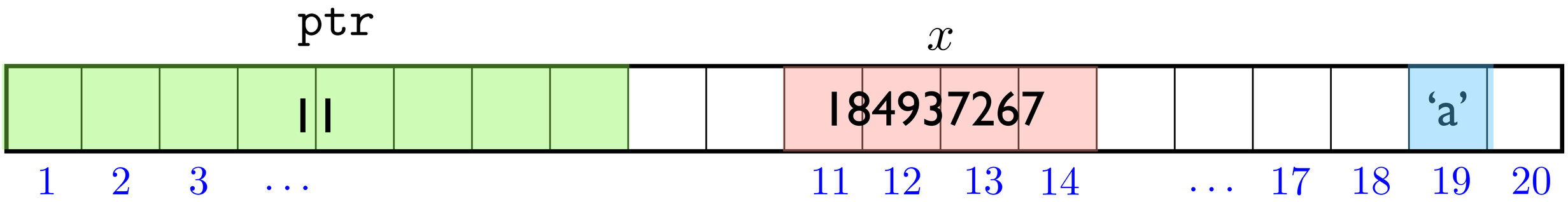
A pointer is a variable holding an address.

```
int main() {
    int x = 184937267;
    int* ptr = &x;
    std::cout << "ptr is " << ptr << '\n';
    std::cout << sizeof(ptr) << '\n';
    return 0;
}
```

How many bytes is `ptr` ?

https://godbolt.org/z/cxsa8Ke6M

Pointers



A pointer is a variable holding an address.

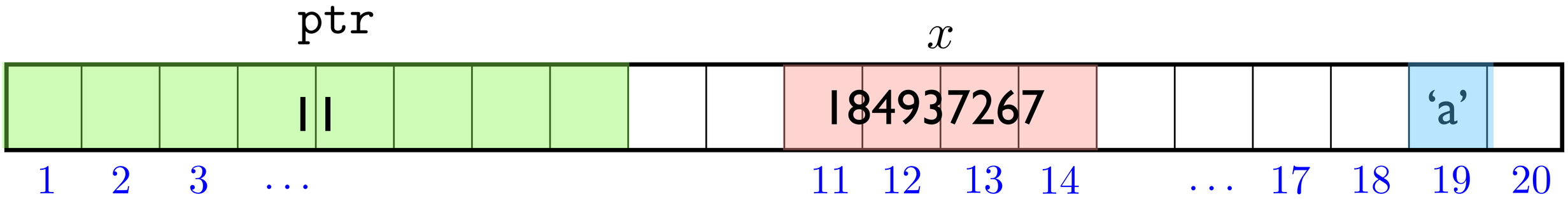
```
int main() {
    int x = 184937267;
    int* ptr = &x;
    std::cout << "ptr is " << ptr << '\n';
    std::cout << sizeof(ptr) << '\n';
    return 0;
}
```

https://godbolt.org/z/cxsa8Ke6M

How many bytes is `ptr` ?

`ptr` holds an address so on a 64-bit arch it will be 8 bytes.

Dereferencing



A pointer is a variable holding an address.

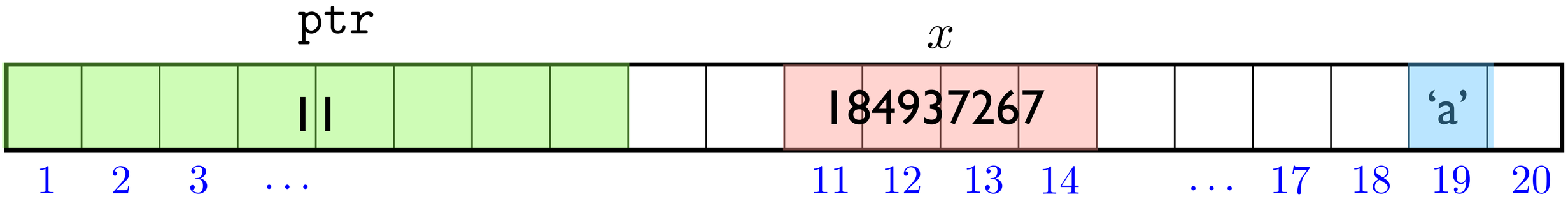
```
int main() {
    int x = 184937267;
    int* ptr = &x;
    std::cout << "the value of x is " << *ptr << '\n';
    return 0;
}
```

https://godbolt.org/z/vEnzKzb3q

What is the difference between an `int*` and a `char*`?

We need to know the type to dereference a pointer.

Dereferencing



A pointer is a variable holding an address.

```
int main() {
    int x = 184937267;
    int* ptr = &x;
    std::cout << "the value of x is " << *ptr << '\n';
    return 0;
}
```

https://godbolt.org/z/vEnzKzb3q

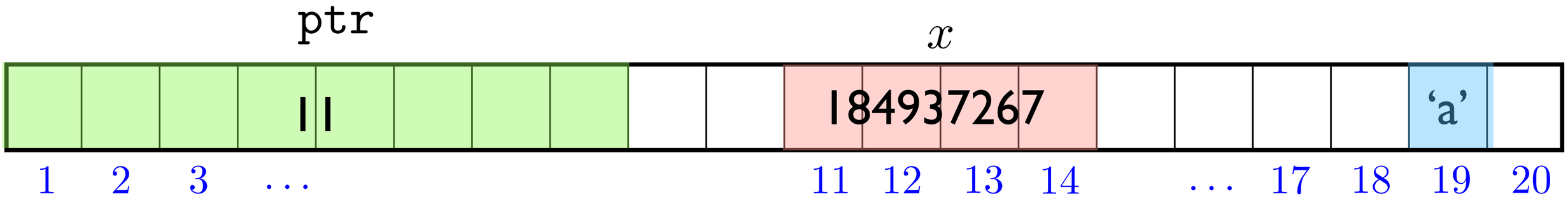
In the picture above

$$\texttt{ptr} == 11$$

$$*\texttt{ptr} == 184937267$$

What is the value of $\&\texttt{ptr}$?

Pointer Address



A pointer is a variable holding an address.

```
int main() {
    int x = 184937267;
    int* ptr = &x;
    std::cout << "the value of x is " << *ptr << '\n';
    return 0;
}
```

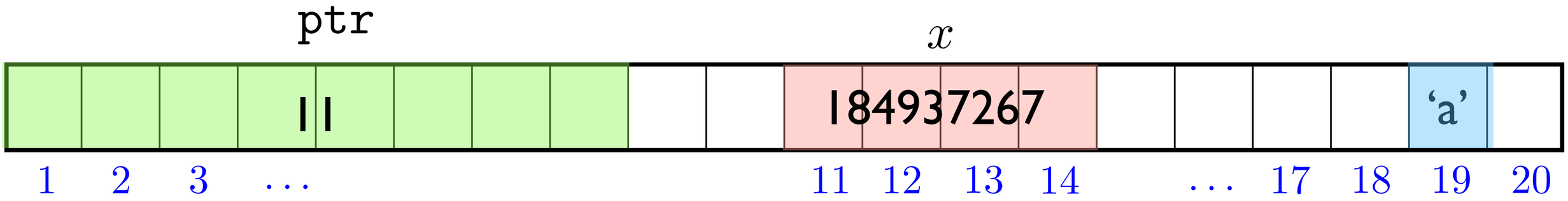
https://godbolt.org/z/vEnzKzb3q

In the picture above

$$\&\texttt{ptr} == 1$$

This is the address of the first byte of `ptr`.

Dereference Assignment



A pointer is a variable holding an address.

```
int main() {
    int x = 184937267;
    int* ptr = &x;
    *ptr = 5;
    std::cout << x << '\n';
    return 0;
}
```

https://godbolt.org/z/Ph8vE4z83

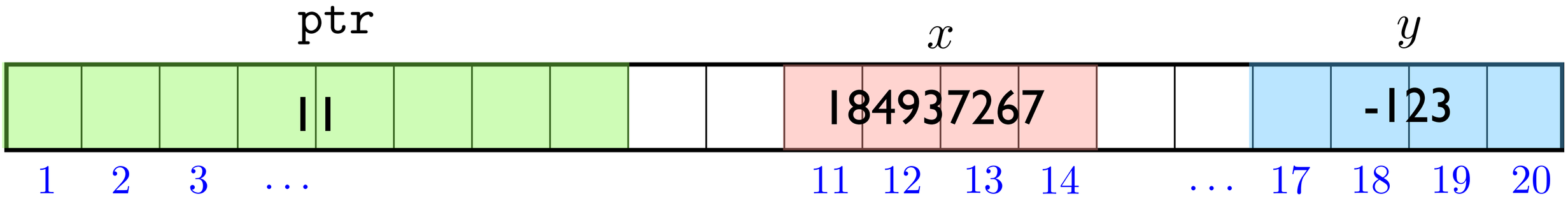
We can use a pointer to x to write to the address of x.

$$*\texttt{ptr} = 5$$

This writes the value 5 in the address stored in `ptr`.

The value of `x` is now 5.

Pointer Assignment



A pointer is a variable holding an address.

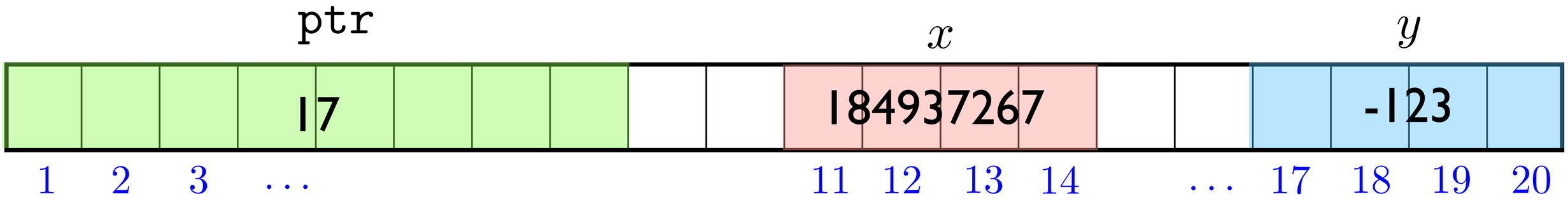
```
int main() {
    int x = 184937267;
    int y = -123;
    int* ptr = &x;
    ptr = &y;
    *ptr = 5;
    std::cout << x << '\n';
    return 0;
}
```

https://godbolt.org/z/4PrbTrEjr

On this line we change the value of `ptr` .

In the picture above, the value of `ptr` becomes 17.

Pointer Assignment



A pointer is a variable holding an address.

```
int main() {
    int x = 184937267;
    int y = -123;
    int* ptr = &x;
    ptr = &y;
    *ptr = 5;
    std::cout << x << '\n';
    return 0;
}
```

https://godbolt.org/z/4PrbTrEjr

On this line we change the value of `ptr`.

In the picture above, the value of `ptr` becomes 17.

We can change the address a pointer points to.

Null Pointer

```
int main() {
    int* ptr = 0;
    // BAD EXAMPLE! Dereferencing memory address 0
    // results in a segmentation fault.
    std::cout << *ptr << '\n';
    return 0;
}
```

https://godbolt.org/z/q8vE3G4dd

The memory address 0 is special.

A program is not allowed to read or write to memory address 0.

Accessing memory location 0 results in a segmentation fault.

Null Pointer

```
int main() {
    int* ptr = nullptr;
    if (ptr != nullptr) {
        std::cout << "do something with ptr\n";
    }
    return 0;
}
```

https://godbolt.org/z/cYTKaaqxq

A pointer with address 0 is called a null pointer.

Null pointers are used as sentinel values.

C++11 introduced the `nullptr` keyword.

Best practice: Use `nullptr` instead of 0 or NULL.

Null Pointer

```
int main() {
    int* ptr = nullptr;
    if (ptr != nullptr) {
        std::cout << "do something with ptr\n";
    }
    return 0;
}
```

https://godbolt.org/z/cYTKaaqxq

`nullptr` is shared by all pointer types.

We can use this with an `int*` or `char*` or anything else.

Chasing Seg Faults

Compiling with the flag -fsanitize=address can help find seg faults and other memory errors.

When we add this flag to our seg fault example we get the following:

```
==1==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000
==1==The signal is caused by a READ memory access.
==1==Hint: address points to the zero page.
    #0 0x401221 in main /app/example.cpp:7
    #1 0x7f82b730a082 in __libc_start_main (/lib/x86_64-linux-gnu/lib
    #2 0x40111d in _start (/app/output.s+0x40111d)
```

https://godbolt.org/z/Grc6jKeEa

It identifies the line number that causes the seg fault.

References

References

A reference is an alias for another variable.

```
int main() {
    int x = 184937267;
    int& xRef = x;
    // prints out 184937267
    std::cout << xRef << '\n';
    return 0;
}
```

https://godbolt.org/z/j1sfcecGY

`xRef` is a reference to an integer. In this context, ampersand is used to denote a reference type, it is not in the role of the address operator.

References

```
int main() {
    int x = 184937267;
    int& xRef = x;
    x = 5;
    // xRef is now 5 too
    std::cout << "xRef is " << xRef << '\n';
    xRef = 10;
    // x is now 10 too
    std::cout << "x is " << x << '\n';
    return 0;
}
```

https://godbolt.org/z/P7nsGx4En

References

```
int main() {
    int x = 184937267;
    int& xRef = x;
    std::cout << "the address of x is " << &x << '\n';
    std::cout << "the address of xRef is " << &xRef << '\n';
    return 0;
}
```

https://godbolt.org/z/c89soEcqa

These addresses are be the same.

References vs Pointers

References are similar to pointers but there are a few key differences:

- References must be initialized: no reference analog of `int * ptr;`
- A reference is bound to the variable with which it is initialized. We cannot later let it stand for a different variable.
- There is no analog of a null pointer for references.

A big advantage of references is that the syntax is easier. You can act on a reference just like you do the variable it aliases.

Arrays

Stroustrup 2.2.4

Initializing an Array

```
int main() {
    int arr1[3] {1,2,3};                // type int[3] holds 1,2,3
    char arr2[] = {'a', 'b', 'c'};      // type char[3] holds 'a','b','c'
    bool arr3[4] {false, true};         // type bool[4] holds false,true,false,false
    int arr4[5] {};                     // type int[5] holds 0,0,0,0,0
    return 0;
}
```

https://godbolt.org/z/YsKnaK34W

Bad Example

When we explicitly state the size, the number of initializers cannot be larger than the size.

```
int main() {
    int arr1[3] {1,2,3,4,5};     // does not compile
    return 0;
}
```

https://godbolt.org/z/vhzqsE1v4

Dangerous Example

When we don't provide an initializer list the elements of the array are not initialized.

range-based for loop

```
int main() {
    int arr1[5];    // no initialization of elements
    for (int x : arr1) {
        std::cout << x << '\n';      //UNDEFINED BEHAVIOR
    }
    return 0;
}
```

https://godbolt.org/z/YKbjrY1sa

Range-Based For

```
int main() {
    int arr[3] {1, 2, 3};
    for (int x : arr) {
        ++x;                // increments copy of array element
    }
    // prints 1 2 3
    // array elements unchanged
    for (int x : arr) {
        std::cout << x << ' ';
    }
    return 0;
}
```

https://godbolt.org/z/aMPvasvvM

Range-Based For

```
int main() {
    int arr[3] {1, 2, 3};
    // take elements of array by reference
    for (int& x : arr) {
        ++x;                // increments array element!
    }
    // prints 2 3 4
    for (int x : arr) {
        std::cout << x << ' ';
    }
    return 0;
}
```

https://godbolt.org/z/Po5zfEPTP

Arrays and Pointers

Arrays are closely related to pointers.

```
int main() {
    int arr[3] {1, 2, 3};
    // these print out the same thing
    std::cout << arr << '\n';
    std::cout << &arr[0] << '\n';
    return 0;
}
```

https://godbolt.org/z/bszeqzedj

The name of the array is a synonym for the address of the first element.

Contiguous Memory

Elements of an array are stored contiguously in memory

```
int main() {
    bool arr[3] {true, false, true};
    for (bool& x : arr) {
        std::cout << &x << '\n';
    }
    return 0;
}
```

output

```
0x7ffc1e76ca9d
0x7ffc1e76ca9e
0x7ffc1e76ca9f
```

https://godbolt.org/z/rPf75T5WK

Booleans take up one byte of memory. The elements of the array have adjacent addresses.

Contiguous Memory

Elements of an array are stored contiguously in memory

```
int main() {
    int arr[3] {1, 2, 3};
    for (int& x : arr) {
        std::cout << &x << '\n';
    }
    return 0;
}
```

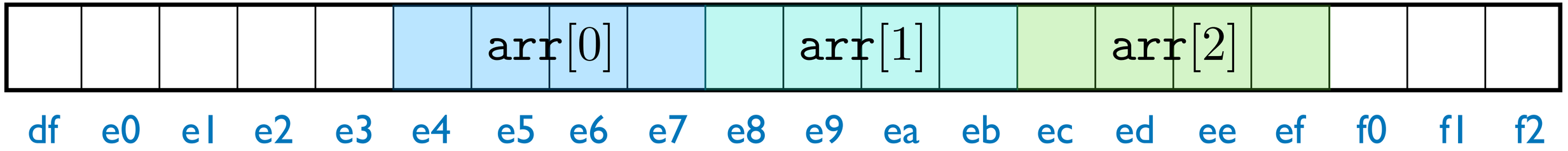
output

```
0x7fffcc364de4
0x7fffcc364de8
0x7fffcc364dec
```

https://godbolt.org/z/P51sYonKv

Ints take 4 bytes of memory. Now the addresses increment by 4 each time.

Picture



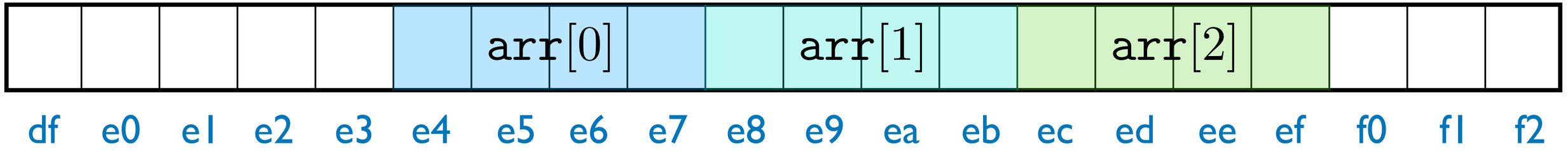

```cpp
int main() {
    int arr[3] {1, 2, 3};
    for (int& x : arr) {
        std::cout << &x << '\n';
    }
    return 0;
}
```

https://godbolt.org/z/P51sYonKv

output

```
&arr[0] 0x7fffcc364de4
&arr[1] 0x7fffcc364de8
&arr[2] 0x7fffcc364dec
```

Pointer Arithmetic

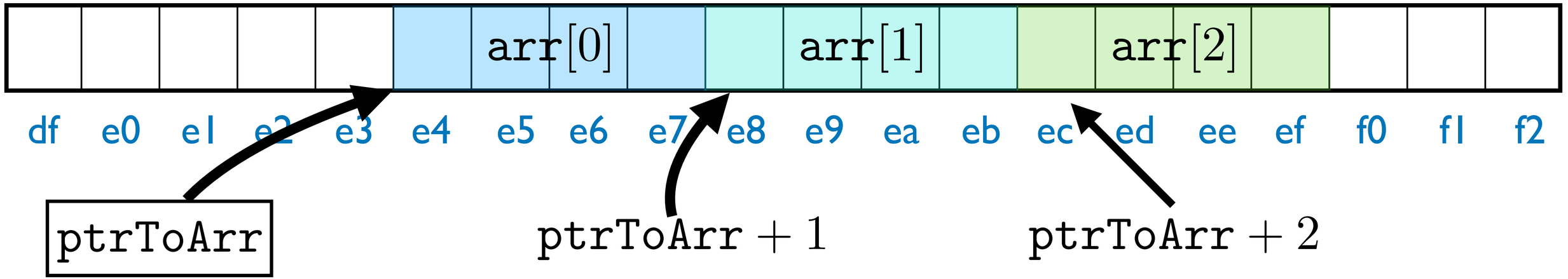


Pointers increment by the size of the type to which they point.

```
int main() {
    int arr[3] {1, 2, 3};
    int* ptrToArr = &arr[0];
    std::cout << ptrToArr + 1 << '\n';
    std::cout << &arr[1] << '\n';
    return 0;
}
```

https://godbolt.org/z/7hqYdTb4v

Pointer Arithmetic




```cpp
int main() {
    int arr[3] {1, 2, 3};
    int* ptrToArr = &arr[0];
    std::cout << ptrToArr + 1 << '\n';
    std::cout << &arr[1] << '\n';
    return 0;
}
```

https://godbolt.org/z/7hqYdTb4v

Iterate with Pointer

```
int main() {
    int arr[3] {1, 2, 3};
    int* ptrToArr = &arr[0];
    for (int i = 0; i < 3; ++i) {
        std::cout << *(ptrToArr + i) << '\n';
    }
    return 0;
}
```

https://godbolt.org/z/sMasWYqaf

Decay to Pointer

```
int main() {
    int arr[3] {1, 2, 3};
    for (int i = 0; i < 3; ++i) {
        std::cout << *(arr + i) << '\n';
    }
    return 0;
}
```

https://godbolt.org/z/zG1b7GPno

An array is not identical to a pointer to its first element, but it decays to a pointer to its first element in many contexts.

C++ Trivia

$\texttt{arr}[i]$ is syntactic sugar for $*(\texttt{arr} + i)$

```
int main() {
    int arr[3] {1, 2, 3};
    // perverse example
    for (int i = 0; i < 3; ++i) {
        std::cout << i[arr] << '\n';
    }
    return 0;
}
```

https://godbolt.org/z/h7PseqGWz