C++ for C Programmers

CS 211

Road map

Two introductions to C++

- What is C++, and why?
- Pro-C++ / Anti-C++

Road map

- Two introductions to C++
  - What is C++, and why?
  - Pro-C++ / Anti-C++
- C things you won't use anymore in C++
  - Standard C headers
  - `printf()`, `scanf()`, format specifiers & FILE*s
  - `malloc()` & `free()`

Road map

- Two introductions to C++
  - What is C++, and why?
  - Pro-C++ / Anti-C++
- C things you won't use anymore in C++
  - Standard C headers
  - `printf()`, `scanf()`, format specifiers & FILE*s
  - `malloc()` & `free()`
- Pass-by-reference

Road map

- Two introductions to C++
  - What is C++, and why?
  - Pro-C++ / Anti-C++
- C things you won't use anymore in C++
  - Standard C headers
  - `printf()`, `scanf()`, format specifiers & `FILE*`s
  - `malloc()` & `free()`
- Pass-by-reference
- Vectors

Up next

Two introductions to C++

- What is C++, and why?
- Pro-C++ / Anti-C++

C things you won't use anymore in C++

- Standard C headers
- `printf`, `scanf`, format specifiers & FILE*s
- `malloc` & `free`

Pass-by-reference

Vectors

Up next

- Two introductions to C++
  - What is C++, and why?
  - Pro-C++ / Anti-C++
- C things you won't use anymore in C++
  - Standard C headers
  - `printf`, `scanf`, format specifiers & FILE*s
  - `malloc` & `free`
- Pass-by-reference
- Vectors

What is C++?

What is C++?

- Feared by many; loved by few; understood by one[1]

What is C++?

- Feared by many; loved by few; understood by one[1]
- Originally an extension to C called “C with Classes”

What is C++?

- Feared by many; loved by few; understood by one[1]
- Originally an extension to C called “C with Classes”
- Intended to bring modern[2] abstraction mechanisms, such as data hiding and generics, to C

What is C++?

- Feared by many; loved by few; understood by one[1]
- Originally an extension to C called “C with Classes”
- Intended to bring modern[2] abstraction mechanisms, such as data hiding and generics, to C
- Adds many other things, too: destructors, exceptions, lambda, dynamic dispatch, inheritance, and a standard library of containers and algorithms

What is C++?

- Feared by many; loved by few; understood by one[1]
- Originally an extension to C called “C with Classes”
- Intended to bring modern[2] abstraction mechanisms, such as data hiding and generics, to C
- Adds many other things, too: destructors, exceptions, lambda, dynamic dispatch, inheritance, and a standard library of containers and algorithms
- But without slowing things down

What is C++?

- Feared by many; loved by few; understood by one[1]
- Originally an extension to C called "C with Classes"
- Intended to bring modern[2] abstraction mechanisms, such as data hiding and generics, to C
- Adds many other things, too: destructors, exceptions, lambda, dynamic dispatch, inheritance, and a standard library of containers and algorithms
- But without slowing things down: "Pay *(for language features)* as you go"

What is C++?

- Feared by many; loved by few; understood by one[1]
- Originally an extension to C called “C with Classes”
- Intended to bring modern[2] abstraction mechanisms, such as data hiding and generics, to C
- Adds many other things, too: destructors, exceptions, lambda, dynamic dispatch, inheritance, and a standard library of containers and algorithms
- But without slowing things down: “Pay *(for language features)* as you go”

[1] Bjarne Stroustrup, its designer

What is C++?

- Feared by many; loved by few; understood by one[1]
- Originally an extension to C called "C with Classes"
- Intended to bring modern[2] abstraction mechanisms, such as data hiding and generics, to C
- Adds many other things, too: destructors, exceptions, lambda, dynamic dispatch, inheritance, and a standard library of containers and algorithms
- But without slowing things down: "Pay *(for language features)* as you go"

[1] Bjarne Stroustrup, its designer

[2] Originally meaning the mid-1980s, but the newest standards are from 2011, 2014, and 2017. C++20 was finalized in Feb., approved in Sept., and will be published later this year. We're using C++14.

What is C++ used for?

Mozilla Firefox, Google Chrome, Microsoft Edge,

What is C++ used for?

Mozilla Firefox, Google Chrome, Microsoft Edge, Microsoft Office (Word, Excel, etc.), Adobe stuff (Photoshop, Illustrator, Acrobat, InDesign, etc.), AutoCAD,

What is C++ used for?

Mozilla Firefox, Google Chrome, Microsoft Edge, Microsoft Office (Word, Excel, etc.), Adobe stuff (Photoshop, Illustrator, Acrobat, InDesign, etc.), AutoCAD, Node.js's virtual machine (V8), most Java Virtual Machines, Microsoft .NET CLR,

What is C++ used for?

Mozilla Firefox, Google Chrome, Microsoft Edge, Microsoft Office (Word, Excel, etc.), Adobe stuff (Photoshop, Illustrator, Acrobat, InDesign, etc.), AutoCAD, Node.js's virtual machine (V8), most Java Virtual Machines, Microsoft .NET CLR, Spotify's audio servers, YouTube's video-processing engine, Bloomberg's real-time financial database system,

What is C++ used for?

Mozilla Firefox, Google Chrome, Microsoft Edge, Microsoft Office (Word, Excel, etc.), Adobe stuff (Photoshop, Illustrator, Acrobat, InDesign, etc.), AutoCAD, Node.js's virtual machine (V8), most Java Virtual Machines, Microsoft .NET CLR, Spotify's audio servers, YouTube's video-processing engine, Bloomberg's real-time financial database system, Oracle, MySQL, IBM DB2, Microsoft SQL Server, MongoDB,

What is C++ used for?

Mozilla Firefox, Google Chrome, Microsoft Edge, Microsoft Office (Word, Excel, etc.), Adobe stuff (Photoshop, Illustrator, Acrobat, InDesign, etc.), AutoCAD, Node.js's virtual machine (V8), most Java Virtual Machines, Microsoft .NET CLR, Spotify's audio servers, YouTube's video-processing engine, Bloomberg's real-time financial database system, Oracle, MySQL, IBM DB2, Microsoft SQL Server, MongoDB, Creation Engine (*Skyrim*, *Fallout* series), Frostbite Engine (*Anthem*, *Battlefield*s, *FIFA*s, *Need for Speed*s) Doom 3 Engine, Unreal Engine,...

What is C++ used for?

Mozilla Firefox, Google Chrome, Microsoft Edge, Microsoft Office (Word, Excel, etc.), Adobe stuff (Photoshop, Illustrator, Acrobat, InDesign, etc.), AutoCAD, Node.js's virtual machine (V8), most Java Virtual Machines, Microsoft .NET CLR, Spotify's audio servers, YouTube's video-processing engine, Bloomberg's real-time financial database system, Oracle, MySQL, IBM DB2, Microsoft SQL Server, MongoDB, Creation Engine (*Skyrim*, *Fallout* series), Frostbite Engine (*Anthem*, *Battlefield*s, *FIFA*s, *Need for Speed*s) Doom 3 Engine, Unreal Engine,...

Mainly: Writing big, complicated programs that also need to perform well

What is C++ used for?

Mozilla Firefox, Google Chrome, Microsoft Edge, Microsoft Office (Word, Excel, etc.), Adobe stuff (Photoshop, Illustrator, Acrobat, InDesign, etc.), AutoCAD, Node.js's virtual machine (V8), most Java Virtual Machines, Microsoft .NET CLR, Spotify's audio servers, YouTube's video-processing engine, Bloomberg's real-time financial database system, Oracle, MySQL, IBM DB2, Microsoft SQL Server, MongoDB, Creation Engine (*Skyrim*, *Fallout* series), Frostbite Engine (*Anthem*, *Battlefield*s, *FIFA*s, *Need for Speed*s) Doom 3 Engine, Unreal Engine,...

Mainly: Writing big, complicated programs that also need to perform well

You could write them in C, but C++ is a lot more flexible, less work, and provides better ways to manage complexity

Why are we using it?

Why are we using it?

- The second half of CS 211 is about learning to build larger programs and structure them using some new abstraction mechanisms

Why are we using it?

- The second half of CS 211 is about learning to build larger programs and structure them using some new abstraction mechanisms
- Other popular* languages that have the features we want, such as Java and C#, wouldn't let you take advantage of your newly-acquired C skills

Why are we using it?

- The second half of CS 211 is about learning to build larger programs and structure them using some new abstraction mechanisms
- Other popular* languages that have the features we want, such as Java and C#, wouldn't let you take advantage of your newly-acquired C skills

* In my experience, most of you don't want to learn an unpopular language :(

Why are we using it?

- The second half of CS 211 is about learning to build larger programs and structure them using some new abstraction mechanisms
- Other popular* languages that have the features we want, such as Java and C#, wouldn't let you take advantage of your newly-acquired C skills
- In C++, the concepts you've been learning still apply, but C++ has a lot of automagic to replace the manual drudgery

* In my experience, most of you don't want to learn an unpopular language :(

Up next

- Two introductions to C++
  - What is C++, and why?
  - Pro-C++ / Anti-C++
- C things you won't use anymore in C++
  - Standard C headers
  - `printf`, `scanf`, format specifiers & FILE*s
  - `malloc` & `free`
- Pass-by-reference
- Vectors

C	C++

C	C++
you must call `free(3)` yourself to deallocate heap objects	

C	C++
you must call `free(3)` yourself to deallocate heap objects	
need a unique name for every function	

C	C++
you must call `free(3)` yourself to deallocate heap objects	
need a unique name for every function	
operators like + and == work only on built-in types	

C	C++
you must call `free(3)` yourself to deallocate heap objects	helpfully frees heap objects when owners go out of scope
need a unique name for every function	
operators like + and == work only on built-in types	

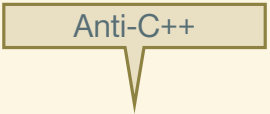
Pro-C++

C	C++
you must call `free(3)` yourself to deallocate heap objects	helpfully frees heap objects when owners go out of scope
need a unique name for every function	can overload function for different argument types
operators like + and == work only on built-in types	

C	C++
you must call `free(3)` yourself to deallocate heap objects	helpfully frees heap objects when owners go out of scope
need a unique name for every function	can overload function for different argument types
operators like + and == work only on built-in types	you can overload operators for user-defined types

Pro-C++

C	C++
you must call `free(3)` yourself to deallocate heap objects	helpfully frees heap objects when owners go out of scope
need a unique name for every function	can overload function for different argument types
operators like + and == work only on built-in types	you can overload operators for user-defined types



Pro-C++

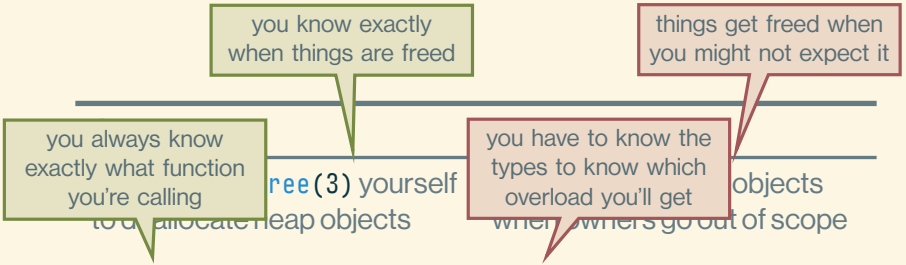
you know exactly when things are freed

things get freed when you might not expect it

C	C++
you must call `free(3)` yourself to deallocate heap objects	helpfully frees heap objects when owners go out of scope
need a unique name for every function	can overload function for different argument types
operators like + and == work only on built-in types	you can overload operators for user-defined types

Anti-C++

Pro-C++


you know exactly when things are freed
things get freed when you might not expect it
you always know exactly what function you're calling
you have to know the types to know which overload you'll get
ree(3) yourself
objects
need a unique name for every function
can overload function for different argument types
operators like + and == work only on built-in types
you can overload operators for user-defined types

Anti-C++

Pro-C++

you know exactly when things are freed

things get freed when you might not expect it

you always know exactly what function you're calling

you have to know the types to know which overload you'll get

`ree(3)` yourself

objects

to d allocate heap objects

when owners go out of scope

you know that / means "divide"

you know that `operator/()` takes two arguments

need a u function

can overload fun different argume

operators like + and == work only on built-in types

you can overload operators for user-defined types

Anti-C++

Up next

- Two introductions to C++
  - What is C++, and why?
  - Pro-C++ / Anti-C++
- C things you won't use anymore in C++
  - Standard C headers
  - `printf`, `scanf`, format specifiers & FILE*s
  - `malloc` & `free`
- Pass-by-reference
- Vectors

Up next

- Two introductions to C++
  - What is C++, and why?
  - Pro-C++ / Anti-C++
- C things you won't use anymore in C++
  - Standard C headers
  - `printf`, `scanf`, format specifiers & FILE*s
  - `malloc` & `free`
- Pass-by-reference
- Vectors

The standard C headers are renamed

Every C header...

```
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
```

The standard C headers are renamed

Every C header...

```
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
```

...loses the `.h` and gets a `c` on the front:

```
#include <cctype>
#include <cmath>
#include <cstdio>
#include <cstring>
```

The standard C headers are renamed and superseded

Every C header...

```
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
```

...loses the `.h` and gets a `c` on the front:

```
#include <cctype>
#include <cmath>
#include <cstdio>
#include <cstring>
```

And C++ has new ways of doing some things:

```
#include <iostream>
#include <string>
```

The standard C headers are renamed and superseded

Every C header...

```
#include <ctype.h>
#include <math.h>
#includ
#includ
```

You won't use these last two much…

...loses the `.h` and gets a `c` on the front:

```
#include <cctype>
#include <cmath>
#include <cstdio>
#include <cstring>
```

And C++ has new ways of doing some things:

…because these two are easier and safer

```
#include <iostream>
#include <string>
```

Up next

- Two introductions to C++
  - What is C++, and why?
  - Pro-C++ / Anti-C++
- C things you won't use anymore in C++
  - Standard C headers
  - `printf()`, `scanf()`, format specifiers & `FILE*`s
  - `malloc` & `free`
- Pass-by-reference
- Vectors

Input/output just got easier & safer & weirder-looking

```
#include <iostream>

int main()
{
    std::cout << "Enter a number to square:\n";
    double x;

    std::cin >> x;

    if (!std::cin) {
        std::cerr << "Error: could not read number!\n";
        return 1;
    }

    std::cout << x << " * " << x << " == " << x * x << "\n";
}
```

Input/output just got easier & safer & weirder-looking

```
#include <iostream>

int main()
{
    std::cout << "Enter a number to square:\n";
    double x;

    std::cin >> x;

    if (!std::cin) {
        std::cerr << "Error: could not read number!\n";
        return 1;
    }

    std::cout << x << " * " << x << " == " << x * x << "\n";
}
```

This is the `#include` for I/O

Input/output just got easier & safer & weirder-looking

The C++ standard library name is in the std namespace.

```
#include <iostream>

int main()
{
    std::cout << "Enter a number to square:\n";
    double x;

    std::cin >> x;

    if (!std::cin) {
        std::cerr << "Error: could not read number!\n";
        return 1;
    }

    std::cout << x << " * " << x << " == " << x * x << "\n";
}
```

Input/output just got easier & safer & weirder-looking

The stream insertion operator writes a value to an output stream.

```
#includ

int main()
{
    std::cout << "Enter a number to square:\n";
    double x;

    std::cin >> x;

    if (!std::cin) {
        std::cerr << "Error: could not read number!\n";
        return 1;
    }

    std::cout << x << " * " << x << " == " << x * x << "\n";
}
```

Input/output just got easier & safer & weirder-looking

```
#include <iostream>

int ma
{
    st                    r a number to square:\n";
    double x;

    std::cin >> x;

    if (!std::cin) {
        std::cerr << "Error: could not read number!\n";
        return 1;
    }

    std::cout << x << " * " << x << " == " << x * x << "\n";
}
```

The stream extraction operator reads from an input stream into an object.

Input/output just got easier & safer & weirder-looking

```
#include <iostream>

int main()
{
    std::cout << "Enter a number to square:\n";

    int x;
    std::cin >> x;

    if (!std::cin) {
        std::cerr << "Error: could not read number!\n";
        return 1;
    }

    std::cout << x << " * " << x << " == " << x * x << "\n";
}
```

To detect an I/O error on a stream, test the stream as if it were a `bool`.

Input/output just got easier & safer & weirder-looking

```
#include <iostream>

int main()
{
    std::cout << "Enter a number to square:\n";
    double x;

    std::cin >> x;

    if
        rror: could not read number!\n";
    }

    std::cout << x << " * " << x << " == " << x * x << "\n";
}
```

The stream operators are left-associative and return their left operand…

Input/output just got easier & safer & weirder-looking

```
#include <iostream>

int main()
{
    std::cout << "Enter a number to square:\n";
    double x;

    std::cin >> x;

    if
        rror: co                ber!\n";
    }

    (((((std::cout << x) << " * ") << x) << " == ") << x * x) << "
}
```

The stream operators are left-associative and return their left operand…

…so it's as if we'd parenthesized them all like this:

Up next

- Two introductions to C++
  - What is C++, and why?
  - Pro-C++ / Anti-C++
- C things you won't use anymore in C++
  - Standard C headers
  - `printf`, `scanf`, format specifiers & FILE*s
  - `malloc()` & `free()`
- Pass-by-reference
- Vectors

Dynamic memory allocation gets safer

Old C Way	Manual C++ Way

Dynamic memory allocation gets safer

Old C Way	Manual C++ Way
`posn_t p = malloc(sizeof *p);`	

Dynamic memory allocation gets safer

Old C Way	Manual C++ Way
`posn_t p = malloc(sizeof *p);`	`posn* p = new posn{3, 4};`

Dynamic memory allocation gets safer

Old C Way	Manual C++ Way
`posn_t p = malloc(sizeof *p);` `if (p == NULL)…`	`posn* p = new posn{3, 4};`

Dynamic memory allocation gets safer

Old C Way	Manual C++ Way
`posn_t p = malloc(sizeof *p);`	`posn* p = new posn{3, 4};`
`if (p == NULL)` …	[*included in* `new`]

Dynamic memory allocation gets safer

Old C Way	Manual C++ Way
`posn_t p = malloc(sizeof *p);`	`posn* p = new posn{3, 4};`
`if (p == NULL)…`	[*included in* `new`]
`p->x = 3; p->y = 4;`	

Dynamic memory allocation gets safer

Old C Way	Manual C++ Way
`posn_t p = malloc(sizeof *p);`	`posn* p = new posn{3, 4};`
`if (p == NULL)` …	[*included in* `new`]
`p->x = 3; p->y = 4;`	[*already did this too*]

Dynamic memory allocation gets safer

Old C Way	Manual C++ Way
`posn_t p = malloc(sizeof *p);`	`posn* p = new posn{3, 4};`
`if (p == NULL)` …	[*included in* `new`]
`p->x = 3; p->y = 4;`	[*already did this too*]
⋮	⋮

Dynamic memory allocation gets safer

Old C Way	Manual C++ Way
`posn_t p = malloc(sizeof *p);`	`posn* p = new posn{3, 4};`
`if (p == NULL)…`	[*included in* `new`]
`p->x = 3; p->y = 4;`	[*already did this too*]
⋮	⋮
`free(p);`	

Dynamic memory allocation gets safer

Old C Way	Manual C++ Way
`posn_t p = malloc(sizeof *p);`	`posn* p = new posn{3, 4};`
`if (p == NULL)…`	[*included in* `new`]
`p->x = 3; p->y = 4;`	[*already did this too*]
⋮	⋮
`free(p);`	`delete p;`

Dynamic memory allocation gets safer and easier

Old C Way

```
posn_t p = malloc(sizeof *p);
if (p == NULL)…
p->x = 3; p->y = 4;
⋮
free(p);
```

Manual C++ Way

```
posn* p = new posn{3, 4};
   [included in new]
   [already did this too]
⋮
delete p;
```

Automatic C++ Way

```
#include <memory>···
{
    std::unique_ptr<posn> p{ new posn{3, 4} };
    ⋮
}
```

Dynamic memory allocation gets safer and easier

Old C Way

```
posn_t p = malloc(sizeof *p);
if (p == NULL)…
p->x = 3; p->y = 4;
⋮
free(p);
```

Manual C++ Way

```
posn* p = new posn{3, 4};
   [included in new]
   [already did this too]
⋮
delete p;
```

Automatic C++ Way

```
#include <memory>···
{
    std::unique_ptr<posn> p{ new posn{3, 4} };
    ⋮
} // automatically deallocated here
```

Arrays, too

Old C Way	Manual C++ Way

Arrays, too

Old C Way	Manual C++ Way
`int* a = calloc(n, sizeof(int));`	

Arrays, too

Old C Way	Manual C++ Way
`int* a = calloc(n, sizeof(int));`	`int* a = new int[n]{};`

Arrays, too

Old C Way	Manual C++ Way
`int* a = calloc(n, sizeof(int));` `if (p == NULL)…`	`int* a = new int[n]{};`

Arrays, too

Old C Way	Manual C++ Way
`int* a = calloc(n, sizeof(int));`	`int* a = new int[n]{};`
`if (p == NULL)` …	[*included in array-*`new`]

Arrays, too

Old C Way	Manual C++ Way
`int* a = calloc(n, sizeof(int));`	`int* a = new int[n]{};`
`if (p == NULL)` …	[*included in array-*`new`]
⋮	⋮

Arrays, too

Old C Way	Manual C++ Way
`int* a = calloc(n, sizeof(int));`	`int* a = new int[n]{};`
`if (p == NULL)…`	[*included in array-*`new`]
⋮	⋮
`free(p);`	

Arrays, too

Old C Way	Manual C++ Way
`int* a = calloc(n, sizeof(int));`	`int* a = new int[n]{};`
`if (p == NULL)…`	[*included in array-new*]
⋮	⋮
`free(p);`	`delete [] p;`

Arrays, too

Old C Way

```
int* a = calloc(n, sizeof(int));
if (p == NULL)...
⋮
free(p);
```

Manual C++ Way

```
int* a = new int[n]{};
   [included in array-new]
⋮
delete [] p;
```

Automatic C++ Way

```
#include <vector>···
{
    std::vector<int> a(n);
    ⋮
}
```

Arrays, too

Old C Way

```
int* a = calloc(n, sizeof(int));
if (p == NULL)...
⋮
free(p);
```

Manual C++ Way

```
int* a = new int[n]{};
   [included in array-new]
⋮
delete [] p;
```

Automatic C++ Way

```
#include <vector>···
{
    std::vector<int> a(n);
    ⋮
} // automatically deallocated here
```

Arrays, too

Old C Way

```
int* a = calloc(n, sizeof(int));
if (p == NULL)…
⋮
free(p);
```

Manual C++ Way

```
int* a = new int[n]{};
   [included in array-new]
⋮
delete [] p;
```

Automatic C++ Way

```
#include <vector>···
{
    std::vector<int> a(n);
    ⋮
} // automatically deallocated here
```

Don't use this stuff...

...because this is much, much better!

Up next

- Two introductions to C++
  - What is C++, and why?
  - Pro-C++ / Anti-C++
- C things you won't use anymore in C++
  - Standard C headers
  - `printf`, `scanf`, format specifiers & FILE*s
  - `malloc` & `free`
- Pass-by-reference
- Vectors

C is completely pass-by-value

```
void f(int x, int* p) { ... }
```

In C, every variable names its own object:

- `x` stands for 4 bytes*, not overlapping with any other variable's object
- `p` stands for 8 bytes*, not overlapping with any other variable's object

C *simulates* pass-by-reference by letting you pass pointers, but you are still passing a value (a pointer value)

* on our particular architecture

C++ has pass-by-reference as well

```
void f(int x, int* p, int& r) { ... }
```

- `x` and `p` are as in C
- `r` refers to some other, existing `int` object
- `r` is borrowed and cannot be `nullptr`

Use `r` like an ordinary `int` — no need to dereference

C++ reference example: Increment

```
#include <211.h>

void inc_ptr(int* p)
{
    *p += 1;
}

void c_style(void)
{
    int x = 0;
    inc_ptr(&x);
    CHECK_INT( x, 1 );
}
```

C++ reference example: Increment

```c
#include <211.h>

void inc_ptr(int* p)
{
    *p += 1;
}

void c_style(void)
{
    int x = 0;
    inc_ptr(&x);
    CHECK_INT( x, 1 );
}
```

```c
#include <catch.hxx>

void inc_ref(int& r)
{
    r += 1;
}

TEST_CASE("C++-style")
{
    int x{0};
    inc_ref(x);
    CHECK( x == 1 );
}
```

C++ reference example: Increment

```
#include <2
```

Our C++ testing framework is defined in this header.

```
void inc_ptr(int* p)
{
    *p += 1;
}

void c_style(void)
{
    int x = 0;
    inc_ptr(&x);
    CHECK_INT( x, 1 );
}
```

```
#include <catch.hxx>

void inc_ref(int& r)
{
    r += 1;
}

TEST_CASE("C++-style")
{
    int x{0};
    inc_ref(x);
    CHECK( x == 1 );
}
```

C++ reference example: Increment

```
#include <2
void inc_ptr(int* p)
{
    *p += 1;
}

void c_style(void)
{
    int x = 0;
    inc_ptr(&x);
    CHECK_INT( x, 1 );
}
```

Our C++ testing framework is defined in this header.

It defines this form for defining tests,

```
#include <catch.hxx>

void inc_ref(int& r)
{
    r += 1;
}

TEST_CASE("C++-style")
{
    int x{0};
    inc_ref(x);
    CHECK( x == 1 );
}
```

C++ reference example: Increment

```
#include <2
void inc_ptr(int* p)
{
    *p += 1;
}

void c_style(void)
{
    int x =
    inc_ptr(
    CHECK_INT( x, 1 );
}
```

Our C++ testing framework is defined in this header.

It defines this form for defining tests,

and this one for checks.

```
#include <catch.hxx>

void inc_ref(int& r)
{
    r += 1;
}

TEST_CASE("C++-style")
{
    int x{0};
    inc_ref(x);
    CHECK( x == 1 );
}
```

C++ reference example: Increment

```c
#include <211.h>

void inc_ptr(int* p)
{
    *p += 1;
}

void c_style(void)
{
    int x = 0;
    inc_ptr(&x);
    CHECK_INT( x, 1 );
}
```

```c
#include <catch.hxx>

void inc_ref(int& r)
{
    r += 1;
}

TEST_CASE("C++ style")
{
    int x{0};
    inc_ref(x);
    CHECK( x == 1 );
}
```

C++ also offers some funny initialization syntaxes

C++ reference example: swap_ref

```
void swap_ref(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}
```

C++ reference example: swap_ref

```
void swap_ref(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}

TEST_CASE("C++-style swap")
{
    int x = 3, y = 4;
    swap_ref(x, y);
    CHECK( x == 4 ); CHECK( y == 3 );
}
```

C++ reference example: swap_ref

```
void swap_ref(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}

TEST_CASE("C++-style swap")
{
    int x = 3, y = 4;
    swap_ref(x, y);
    CHECK( x == 4 ); CHECK( y == 3 );
}
```

(swap_ref is std::swap<int>.)

How C++ desugars references to pointers

Syntactically sweet:

```
void swap(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}

swap(x, v[3]);
```

How C++ desugars references to pointers

Syntactically sweet:

```
void swap(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}

swap(x, v[3]);
```

Desugared:

```
void swap(int* rp, int* sp)
{
    int temp = *rp;
    *rp = *sp;
    *sp = temp;
}

swap(&x, &v[3]);
```

How C++ desugars references to pointers

Replace every declared reference…

Syntactically sweet:

```
void swap(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}

swap(x, v[3]);
```

…with a pointer,…

Desugared:

```
void swap(int* rp, int* sp)
{
    int temp = *rp;
    *rp = *sp;
    *sp = temp;
}

swap(&x, &v[3]);
```

How C++ desugars references to pointers

Replace every declared reference…

Syntactically sweet:

```
void swap(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}

swap(x, v[3]);
```

…with a pointer,…

Desugared:

```
void swap(int* rp, int* sp)
{
    int temp = *rp;
    *rp = *sp;
    *sp = temp;
}

swap(&x, &v[3]);
```

…each use of which is automatically dereferenced by inserting prefix *.

How C++ desugars references to pointers

Replace every declared reference…

Syntactically sweet:

```
void swap(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}

swap(x, v[3]);
```

Then automatically pass the address of each initializer…

…with a pointer,…

Desugared:

```
void swap(int* rp, int* sp)
{
    int temp = *rp;
    *rp = *sp;
    *sp = temp;
}

swap(&x, &v[3]);
```

…each use of which is automatically dereferenced by inserting prefix *.

…by adding prefix &.

It works anywhere you define variables,

and it can be `const` too

It works anywhere you define variables,
and it can be `const` too

Sweet:

```
entry& e = entries[i];
```

Desugared:

It works anywhere you define variables, and it can be `const` too

Sweet:

```
entry& e = entries[i];
```

Desugared:

```
entry* pe = &entries[i];
```

It works anywhere you define variables, and it can be `const` too

Sweet:

```
entry& e = entries[i];
std::string const& n = e.name;
```

Desugared:

```
entry* pe = &entries[i];
```

It works anywhere you define variables, and it can be `const` too

Sweet:

```
entry& e = entries[i];
std::string const& n = e.name;
```

Desugared:

```
entry* pe = &entries[i];
std::string const* pn = &pe->name;
```

It works anywhere you define variables, and it can be `const` too

Sweet:

```
entry& e = entries[i];
std::string const& n = e.name;

if (n == current) {
    ++e.count;
}
```

Desugared:

```
entry* pe = &entries[i];
std::string const* pn = &pe->name;
```

It works anywhere you define variables, and it can be `const` too

Sweet:

```
entry& e = entries[i];
std::string const& n = e.name;

if (n == current) {
    ++e.count;
}
```

Desugared:

```
entry* pe = &entries[i];
std::string const* pn = &pe->name;

if (*pn == current) {
    ++pe->count;
// ++(*pe).count
}
```

Example: Alternative swap definition

Does this work?

```
void alt_swap(int& r, int& s)
{
    int& temp = r;
    r = s;
    s = temp;
}
```

Example: Alternative swap definition

Does this work?

```
void alt_swap(int& r, int& s)
{
    int& temp = r;
    r = s;
    s = temp;
}
// becomes
void alt_swap(int* rp, int* sp)
{
    int* tempp = &*rp;
    *rp = *sp;
    *sp = *tempp;
}
```

Up next

Two introductions to C++

- What is C++, and why?
- Pro-C++ / Anti-C++

C things you won't use anymore in C++

- Standard C headers
- `printf`, `scanf`, format specifiers & FILE*s
- `malloc` & `free`

Pass-by-reference

Vectors

Using `std::vector` (1/3)

```
#include <catch.hxx>
#include <vector>

TEST_CASE("vector creation and access")
{
    std::vector<double> v1(10, 3.5);
    CHECK( v1.size() == 10 );
    CHECK( v1[9] == 3.5 );
    v1[9] = 17;
    CHECK( v1[9] == 17 );

    std::vector<int> v2{ 2, 4, 6, 8 };
    CHECK( v2.size() == 4 );
    CHECK( v2[1] == 4 );
}
```

Using `std::vector` (1/3)

```
#include <catch.hxx>
#include <vector>

TEST_CASE("vector creation and access")
{
    std::vector<double> v1(10, 3.5);
    CHECK( v1.size() == 10 );
    CHECK( v1[9] == 3.5 );
    v1[9] = 17;
    CHECK( v1[9] == 17 );

    std::vector<int> v2{ 2, 4, 6, 8 };
    CHECK( v2.size() == 4 );
    CHECK( v2[1] == 4 );
}
```

Parentheses take a size and, optionally, a value to fill with

Using `std::vector` (1/3)

```
#include <catch.hxx>
#include <vector>

TEST_CASE("vector creation and access")
{
    std::vector<double> v1(10, 3.5);
    CHECK( v1.size() == 10 );
    CHECK( v1[9] == 3.5 );
    v1[9] = 17;
    CHECK( v1[9] == 17 );

    std::vector<int> v2{ 2, 4, 6, 8 };
    CHECK( v2.size() == 4 );
    CHECK( v2[1] == 4 );
}
```

Parentheses take a size and, optionally, a value to fill with

Curly braces take a list of elements

Using `std::vector` (1/3)

```
#include <catch.hxx>
#include <vector>

TEST_CASE("vector creation and access")
{
    std::vector<double> v1(10, 3.5);
    CHECK( v1.size() == 10 );
    CHECK( v1[9] == 3.5 );
    v1[9] = 17;
    CHECK( v1[9] == 17 );

    std::vector<int> v2{ 2, 4, 6, 8 };
    CHECK( v2.size() == 4 );
    CHECK( v2[1] == 4 );
}
```

The size() member function returns the number of elements

Using `std::vector` (1/3)

```
#include <catch.hxx>
#include <vector>

TEST_CASE("vector creation and access")
{
    std::vector<double> v1(10, 3.5);
    CHECK( v1.size() == 10 );
    CHECK( v1[9] == 3.5 );
    v1[9] = 17;
    CHECK( v1[9] == 17 );

    std::vector<int> v2{ 2, 4, 6, 8 };
    CHECK( v2.size() == 4 );
    CHECK( v2[1] == 4 );
}
```

Index using square brackets, just like with a C array

Using `std::vector` (1/3)

```
#include <catch.hxx>
#include <vector>

TEST_CASE("vector creation and access")
{
    std::vector<double> v1(10, 3.5);
    CHECK( v1.size() == 10 );
    CHECK( v1[9] == 3.5 );
    v1[9] = 17;
    CHECK( v1[9] == 17 );

    std::vector<int> v2{ 2, 4, 6, 8 };
    CHECK( v2.size() == 4 );
    CHECK( v2[1] == 4 );
}
```

Index using square brackets, just like with a C array

As in C, indexing out of bounds is UB

Using `std::vector` (2/3)

```
using VI = std::vector<int>;

TEST_CASE("growing and shrinking")
{
    VI v;

    CHECK( v == VI{} );
    v.push_back(2);
    CHECK( v == VI{2} );
    v.push_back(5);
    v.push_back(9);
    CHECK( v == VI{2, 5, 9} );

    v.pop_back();
    CHECK( v == VI{2, 5} );
}
```

Using `std::vector` (2/3)

```
using VI = std::vector<int>;

TEST_CASE("growing and shrinking")
{
    VI v;

    CHECK( v == VI{} );
    v.push_back(2);
    CHECK( v == VI{2} );
    v.push_back(5);
    v.push_back(9);
    CHECK( v == VI{2, 5, 9} );

    v.pop_back();
    CHECK( v == VI{2, 5} );
}
```

Like typedef but not backward

Using `std::vector` (2/3)

```
using VI = std::vector<int>;

TEST_CASE("growing and shrinking")
{
    VI v;

    CHECK( v == VI{} );
    v.push_back(2);
    CHECK( v == VI{2} );
    v.push_back(5);
    v.push_back(9);
    CHECK( v == VI{2, 5, 9} );

    v.pop_back();
    CHECK( v == VI{2, 5} );
}
```

Like typedef but not backward

Grows the vector by appending an element to the back

Using `std::vector` (2/3)

```
using VI = std::vector<int>;

TEST_CASE("growing and shrinking")
{
    VI v;

    CHECK( v == VI{} );
    v.push_back(2);
    CHECK( v == VI{2} );
    v.push_back(5);
    v.push_back(9);
    CHECK( v == VI{2, 5, 9} );

    v.pop_back();
    CHECK( v == VI{2, 5} );
}
```

Like typedef but not backward

Grows the vector by appending an element to the back

Shrinks the vector by removing the last element

Using `std::vector` (3/3)

```cpp
#include <stdexcept>

TEST_CASE("optional bounds checking")
{
    std::vector<int> v{2, 3, 4};

    CHECK(v.at(2) == 4);
    v.at(2) = 8;
    CHECK(v.at(2) == 8);

    CHECK_THROWS_AS(v.at(3), std::out_of_range);

    v[10] = 12;            // UB!
    CHECK( v[10] == 12 ); // also UB!
}
```

Using `std::vector` (3/3)

```cpp
#include <stdexcept>

TEST_CASE("optional bounds checking")
{
    std::vector<int> v{2, 3, 4};

    CHECK(v.at(2) == 4);
    v.at(2) = 8;
    CHECK(v.at(2) == 8);

    CHECK_THROWS_AS(v.at(3), std::out_of_range);

    v[10] = 12;            // UB!
    CHECK( v[10] == 12 ); // also UB!
}
```

If you want bounds checking, use `vector<T>::at()` instead of `vector<T>::operator[]()`

Using `std::vector` (3/3)

```
#include <stdexcept>

TEST_CASE("optional bounds checking")
{
    std::vector<int> v{2, 3, 4};

    CHECK(v.at(2) == 4);
    v.at(2) = 8;
    CHECK(v.at(2) == 8);

    CHECK_THROWS_AS(v.at(3), std::out_of_range);

    v[10] = 12;            // UB!
    CHECK( v[10] == 12 ); // also UB!
}
```

`at()` returns by reference, so you can assign to it

Using `std::vector` (3/3)

```cpp
#include <stdexcept>

TEST_CASE("optional bounds checking")
{
    std::vector<int> v{2, 3, 4};

    CHECK(v.at(2) == 4);
    v.at(2) = 8;
    CHECK(v.at(2) == 8);

    CHECK_THROWS_AS(v.at(3), std::out_of_range);

    v[10] = 12;            // UB!
    CHECK( v[10] == 12 ); // also UB!
}
```

When given a bad index, `at()` throws an exception called `std::out_of_range`

Using `std::vector` (3/3)

```
#include <stdexcept>

TEST_CASE("optional bounds checking")
{
    std::vector<int> v{2, 3, 4};

    CHECK(v.at(0) == 4);
    CHECK(v.at(2) == 8);

    CHECK_THROWS_AS(v.at(3), std::out_of_range);

    v[10] = 12;            // UB!
    CHECK( v[10] == 12 ); // also UB!
}
```

Here's how you check for an exception in a unit test

When given a bad index, `at()` throws an exception called `std::out_of_range`

Using `std::vector` (3/3)

```
#include <stdexcept>

TEST_CASE("optional bounds checking")
{
    std::vector<int> v{2, 3, 4};

    CHECK(v.at(0) == 4);
    CHECK(v.at(2) == 8);

    CHECK_THROWS_AS(v.at(3), std::out_of_range);

    v[10] = 12;            // UB!
    CHECK( v[10] == 12 ); // also UB!
}
```

std::out_of_range is defined in here

Here's how you check for an exception in a unit test

When given a bad index, at() throws an exception called std::out_of_range

`std::vector` is passed by value...

```
void faulty_inc_vec(std::vector<int> v)
{
    for (size_t i = 0; i < v.size(); ++i)
        ++v[i];
}

TEST_CASE("vector passed by value")
{
    std::vector<int> v{ 2, 3, 4 };
    faulty_inc_vec(v);
    CHECK( v == std::vector<int>{ 3, 4, 5 } );
}
```

...unless passed by reference

```
void inc_vec(std::vector<int>& v)
{
    for (size_t i = 0; i < v.size(); ++i)
        ++v[i];
}

TEST_CASE("vector passed by reference")
{
    std::vector<int> v{ 2, 3, 4 };
    inc_vec(v);
    CHECK( v == std::vector<int>{ 3, 4, 5 } );
}
```

Vectors offer easier (and more generic) iteration

```
double sum_vec(std::vector<double> const& v)
{
    double result = 0;
    for (double d : v) {
        result += d;
    }
    return result;
}
```

Vectors offer easier (and more generic) iteration

```
double sum_vec(std::vector<double> const& v)
{
    double result = 0;
    for (double d : v) {
        result += d;
    }
    return result;
}
```

Ranges over elements, not indices!

Vectors offer easier (and more generic) iteration

```
double sum_vec(std::vector<double> const& v)
{
    double result = 0;
    for (double d : v) {
        result += d;
    }
    return result;
}


void dec_vec_wrong(std::vector<int> &v)
{
    for (int z : v) --z;
}
```

Vectors offer easier (and more generic) iteration

```
double sum_vec(std::vector<double> const& v)
{
    double result = 0;
    for (double d : v) {
        result += d;
    }
    return result;
}


void dec_vec_wrong(std::vector<int> &v)
{
    for (int z : v) --z;
}
```

Makes `z` its own `int` object, which means mutating `z` has *no effect* on `v`

Vectors offer easier (and more generic) iteration

```
double sum_vec(std::vector<double> const& v)
{
    double result = 0;
    for (double d : v) {
        result += d;
    }
    return result;
}


void dec_vec_right(std::vector<int> &v)
{
    for (int& z : v) --z;
}
```

Makes `z` an alias of each `int` object inside `v`, so mutating `z` also mutates `v`

More `std::vector<T>` operations

- `bool empty() const;`
- `void clear();`
- `T& front();`
- `T& back();`
- `T const& front() const;`
- `T const& back() const;`
- `void resize(size_t count, T const& fill);`
- `void resize(size_t count);`

More `std::vector<T>` operations

- `bool empty() const;`
- `void clear();`
- `T& front();`
- `T& back();`
- `T const& front() const;`
- `T const& back() const;`
- `void resize(size_t count, T const& fill);`
- `void resize(size_t count);`

`v.empty()` is *true* when `v` is empty (and it doesn't mutate the `v`)

More `std::vector<T>` operations

- `bool empty() const;`
- `void clear();`
- `T& front();`
- `T& back();`
- `T const& front() const;`
- `T const& back() const;`
- `void resize(size_t count, T const& fill);`
- `void resize(size_t count);`

`v.clear()` empties `v`

More `std::vector<T>` operations

- `bool empty() const;`
- `void clear();`
- `T& front();`
- `T& back();`
- `T const& front() const;`
- `T const& back() const;`
- `void resize(size_t count, T const& fill);`
- `void resize(size_t count);`

`v.front()` is equivalent to `v[0]` (including UB if `v.empty()`)

More `std::vector<T>` operations

- `bool empty() const;`
- `void clear();`
- `T& front();`
- `T& back();`
- `T const& front() const;`
- `T const& back() const;`
- `void resize(size_t count, T const& fill);`
- `void resize(size_t count);`

`v.back()` is equivalent to `v[v.size() - 1]`

(including UB if `v.empty()`)

More `std::vector<T>` operations

- `bool empty() const;`
- `void clear();`
- `T& front();`
- `T& back();`
- `T const& front() const;`
- `T const& back() const;`
- `void resize(size_t count, T const& fill);`
- `void resize(size_t count);`

These `const this` overloads say that when the vector is constant then so is the reference you get back

More `std::vector<T>` operations

- `bool empty() const;`
- `void clear();`
- `T& front();`
- `T& back();`
- `T const& front() const;`
- `T const& back() const;`
- `void resize(size_t count, T const& fill);`
- `void resize(size_t count);`

Changes the size; if growing, copies `fill` for new elements

More `std::vector<T>` operations

- `bool empty() const;`
- `void clear();`
- `T& front();`
- `T& back();`
- `T const& front() const;`
- `T const& back() const;`
- `void resize(size_t count, T const& fill);`
- `void resize(size_t count);`

Changes the size; if growing, default-constructs new elements

More `std::vector<T>` operations

- `bool empty() const;`
- `void clear();`
- `T& front();`
- `T& back();`
- `T const& front() const;`
- `T const& back() const;`
- `void resize(size_t count, T const& fill);`
- `void resize(size_t count);`

See API reference for more:
`https://en.cppreference.com/w/cpp/container/vector`

– Next time: Farewell to C –