Today’s Agenda

• A Brief History of C++

• Gentle Introduction to C++ with Examples and by Trial & Error
  • ...and what a terrible idea that will prove to be

• The Dark Side of C++

• Why Compilation is Terrible

• Templates

• Weird Syntax
  • Types
  • Memory
  • Strings

• Ease of Over-Engineering

• Historical Baggage

• Hidden Pitfalls
A Brief History of C++
C++ began being invented in 1979 by Danish computer scientist Bjarne Stroustrup
Bjarne Stroustrup is a humble man.

Bjarne does not want to tell you what to do.

Bjarne wants to empower you to do anything you can imagine.

And Bjarne trusts you to know right from wrong.
Bjarne Stroustrup (Inventor of C++)

versus James Gosling (Inventor of Java)

“Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way [...] Often, I was tempted to outlaw a feature I personally disliked, I refrained from doing so because I did not think I had the right to force my views on others.”

The Design and Evolution of C++

“I left out operator overloading as a fairly personal choice because I had seen too many people abuse it in C++.“

http://www.gotw.ca/publications/c_family_interview.htm
## C++ is Not Done Being Invented

<table>
<thead>
<tr>
<th>Stone Age</th>
<th>C with Classes</th>
<th>C++</th>
<th>C++98</th>
<th>C++03</th>
<th>C++11</th>
<th>C++14</th>
<th>C++20</th>
<th>???</th>
</tr>
</thead>
</table>

### Length of C++ Language Standard (in pages)

- 1980: 0 pages
- 1990: 500 pages
- 2000: 1000 pages
- 2010: 1500 pages
- 2020: 2000 pages
- ???: 2200 pages

C++ is a continuously evolving language, with each new standard expanding its capabilities and features.
Length of Language Specification (Number of Pages)

- Python: 200 pages
- Racket: 400 pages
- Rust: 600 pages
- C: 600 pages
- JavaScript: 800 pages
- Java: 800 pages
- C++: 1800 pages
# Recent Versions of C++

<table>
<thead>
<tr>
<th>C++11</th>
<th>C++14</th>
<th>C++17</th>
<th>C++20</th>
</tr>
</thead>
</table>
| • Fundamentally changed the language to allow more efficient resource management  
• First gave any meaning at all to multithreaded code  
• Made templates go from slightly nuts to completely nuts (variadic templates) | • Not much happened  
• You can write binary numbers now: 0b1011 | • A few things happened  
• It’s now really easy to write code that runs before your code ever runs  
• First gave any meaning at all to the file system  
• Made templates even more nuts (fold expressions) | • Fundamentally changes how you use algorithms (ranges)  
• Fundamentally changes how you package and reuse code (modules)  
• Adds the spaceship operator <=  
• First appearance of time and date in C++  
• First gave any meaning at all to endianness  
• Makes templates a little more sane |
Gentle Introduction to C++

WITH EXAMPLES AND BY TRIAL & ERROR
Hello World: Attempt 1

```cpp
#include <iostream>

int main()
{
    std::cout << 'Hello, world!';
}
```
Hello World: Attempt 2

```cpp
#include <iostream>

int main(){
    std::cout << "Hello, world!";
}
```

Hello, world!
How to Concatenate Strings

```cpp
#include <iostream>

int main()
{
    std::cout << 'H' + 'i';
}
```
How to Concatenate Strings

```
#include <iostream>

int main()
{
    std::cout << "Hello, " + 'world!';
}
```

```
bash: line 7: 4103 Segmentation fault (core dumped) ./a.out
```
How to Concatenate Strings

```cpp
#include <iostream>

int main(){
    std::cout << "Hello, " + "world!";
}
```
How to Concatenate Strings

```cpp
#include <iostream>

int main()
{
    std::cout << "Hi" + '!' ;
}
```
How to Concatenate Strings

```cpp
#include <iostream>

int main(){
    std::cout << "Hi" - '!';
}
```
How to Convert Numbers to Strings

```cpp
#include <iostream>

int main()
{
    auto s1 = "Your lucky number is: ";
    auto s2 = s1 + 10;
    std::cout << s2;
}
```

number is:
How to Convert Numbers to Strings

That’s kind of verbose...

```cpp
#include <iostream>

int main()
{
    long int a = 36762444129608;
    std::string s = std::to_string(a);
    std::cout << s;
}
```

36762444129608
How to Convert Numbers to Strings

Why not convert it directly to a char*?

```c++
#include <iostream>

int main(){
    long int a = 36762444129608;
    std::string s = (char*)a;
    std::cout << s;
}
```

Hello!
Working with Numbers

C++ has numbers for every occasion
Numbers for Every Occasion

- short
- short int
- signed short
- signed short int
- unsigned short
- unsigned short int
- int
- signed
- signed int
- unsigned
- unsigned int
- long
- long int
- signed long
- signed long int
- unsigned long
- unsigned long int
- signed char
- unsigned char
- char
- wchar_t
- char8_t
- char16_t
- char32_t
- float
- double
- long double
- std::size_t
- std::ptrdiff_t
- std::intptr_t
- std::uintptr_t
- std::int8_t
- std::int16_t
- std::int32_t
- std::int64_t
- std::int_fast8_t
- std::int_fast16_t
- std::int_fast32_t
- std::int_fast64_t
- std::int_least8_t
- std::int_least16_t
- std::int_least32_t
- std::int_least64_t
- std::intmax_t
- std::uint8_t
- std::uint16_t
- std::uint32_t
- std::uint64_t
- std::uint_fast8_t
- std::uint_fast16_t
- std::uint_fast32_t
- std::uint_fast64_t
- std::uint_least8_t
- std::uint_least16_t
- std::uint_least32_t
- std::uint_least64_t
- std::uintmax_t
- std::streamoff
- std::streamsize

Note: std::byte is not a number!
Working with Numbers

Numbers don’t need initial values

(Compiled with –O0 on g++)
Working with Numbers

Increase your compiler’s optimization level to get better numbers

(compiled with –O1 on g++)

```cpp
#include <iostream>

int main(){
    int i;
    double d;
    bool b;
    uint8_t u;
    std::cout << i << \n';
    std::cout << d << \n';
    std::cout << b << \n';
    std::cout << u << \n';
}
```
Working with Numbers

Try a different compiler and see what works best for you

(compiled with –O2 on clang++)
Working with Numbers

Printing whitespace can have its consequences.

(compiled with -O2 on clang++)
Which of these Numbers is smaller?

```cpp
#include <iostream>

int main()
{
    std::cout << std::min(2.5, 3);
}
```

main.cpp: In function 'int main()':
main.cpp:4:33: error: no matching function for call to 'min(double, int)'
  std::cout << std::min(2.5, 3);
main.cpp: In function 'int main()':
main.cpp:4:33: error: no matching function for call to 'min(double, int)'
   | std::cout << std::min(2.5, 3);
   ^
In file included from /usr/local/include/c++/9.2.0/bits/char_traits.h:39,
   from /usr/local/include/c++/9.2.0/ios:40,
   from /usr/local/include/c++/9.2.0/iostream:38,
   from /usr/local/include/c++/9.2.0/ostream:38,
   from /usr/local/include/c++/9.2.0/bits/stl_algobase.h:198:
   note: candidate: 'template<class Tp> constexpr const Tp& std::min(const Tp&, const Tp&)
   | min(const Tp& __a, const Tp& __b)
   ^~~
/usr/local/include/c++/9.2.0/bits/stl_algobase.h:198:5: note:   template argument deduction/substitution failed:
main.cpp:4:33: note: deduced conflicting types for parameter 'const Tp' ('double' and 'int')
   | std::cout << std::min(2.5, 3);
   ^
In file included from /usr/local/include/c++/9.2.0/bits/char_traits.h:39,
   from /usr/local/include/c++/9.2.0/ios:40,
   from /usr/local/include/c++/9.2.0/iostream:38,
   from /usr/local/include/c++/9.2.0/ostream:38,
   from /usr/local/include/c++/9.2.0/bits/stl_algobase.h:246:
   note: candidate: 'template<class Tp, class Compare> constexpr const Tp& std::min(const Tp&, const Tp&, Compare)
   | min(const Tp& __a, const Tp& __b, Compare __comp)
   ^~~
/usr/local/include/c++/9.2.0/bits/stl_algobase.h:246:5: note:   template argument deduction/substitution failed:
main.cpp:4:33: note: deduced conflicting types for parameter 'const Tp' ('double' and 'int')
   | std::cout << std::min(2.5, 3);
   ^
Macros to the Rescue!

Hey, that works way better!
Macros to the Rescue!

Let’s replace `min` with `product`

```cpp
#include <iostream>

#define product(a, b) a * b

int main()
{
    std::cout << product(2, 1 + 1);
}
```
Reading User Input

```cpp
#include <iostream>

int main(){
    char* str = "";
    std::cin >> str;
    std::cout << str;
}
```

```
bash: line 7: 5410 Done
5411 Segmentation fault (core dumped) ./a.out
```

Reading User Input
if statements

JavaScript is not the only place where things get “truthy”

```cpp
#include <iostream>

int main()
{
    if (true)
    {
        std::cout << "Yes!";
    }
}
```

```cpp
#include <iostream>

int main()
{
    if (42)
    {
        std::cout << "Yes!";
    }
}
```

```cpp
#include <iostream>

int main()
{
    if (0.999)
    {
        std::cout << "Yes!";
    }
}
```

```cpp
#include <iostream>

int main()
{
    if ("Yes!")
    {
        std::cout << "Yes!";
    }
}
```

```cpp
#include <iostream>

int main()
{
    if (std::cout)
    {
        std::cout << "Yes!";
    }
}
```

```cpp
#include <iostream>

int main()
{
    if (main)
    {
        std::cout << "Yes!";
    }
}
```
Let’s Introduce Functions

```cpp
#include <iostream>

char* foo(){
    return "Hello, world!";
}

int main(){
    std::cout << foo;
}
```
Let’s Introduce Functions

```cpp
#include <iostream>

char* foo() {
    return "Hello, world!";
}

int main(){
    std::cout << foo();
}

Hello, world!
```
```cpp
#include <iostream>

int foo()
{
}

int main()
{
    std::cout << foo();
}
```
#include <iostream>

char* foo() {
}

int main() {
    std::cout << foo();
}

Return Values are Optional
Functions Can Be Used Anywhere

```cpp
#include <iostream>

bool foo(){
}

int main(){
    if (foo()){
        std::cout << "true";
    } else {
        std::cout << "false";
    }
}
```

Compiled with -O2 on g++
Functions Can Be Used Anywhere

```cpp
#include <iostream>

bool foo(){
}

int main(){
    if (foo()){
        std::cout << "true";
    } else {
        std::cout << "false";
    }
}
```

Compiled with -O0 on clang++
How to Pass Arguments to a Function

PASS BY VALUE (DEFAULT)

```cpp
#include <iostream>

int foo(int x){
    x += 10;
    return x;
}

int main()
{
    std::cout << foo(22);
}
```

PASS BY REFERENCE (NOTE THE &)

```cpp
#include <iostream>

void foo(int& x){
    x += 10;
}

int main()
{
    int i = 22;
    foo(i);
    std::cout << i;
}
```
How to Return from a Function

RETURN BY VALUE

```cpp
#include <iostream>

int foo(int x){
    x += 10;
    return x;
}

int main(){
    std::cout << foo(22);
}
```

RETURN BY REFERENCE (NOTE THE &)

```cpp
#include <iostream>

int& foo(int x){
    x += 10;
    return x;
}

int main(){
    std::cout << foo(22);
}
```

```
bash: line 7: 12089 Done
12090 Segmentation fault (core dumped) | ./a.out
```
Functions can be Overloaded

Multiple functions can have the same name in C++ as long as they accept different arguments.

The correct function will be chosen using the type of the argument you pass.

```cpp
#include <iostream>

void print(std::string s){
    std::cout << s;
}

void print(bool b){
    std::cout << (b ? "true" : "false");
}

int main(){
    print("Hello, world!");
}
```
Arrays

```cpp
#include <iostream>

int main()
{
    int arr[] = { 1, 2, 3, 4, 5 };
    std::cout << arr;
}
```

0x7fff6d38aa30
Arrays

```cpp
#include <iostream>

int main()
{
    int arr[] = { 1, 2, 3, 4 };
    for (int i = 0; i <= 4; ++i)
    {
        std::cout << arr[i] << ' ';
    }
}
```

1 2 3 4 5 93744096
Arrays

```cpp
#include <iostream>

int main(){
    int arr[4] = { 1, 2, 3, 4 };

    for (int x : arr){
        std::cout << x << ' ';
    }
}
```
Passing Arrays to Functions

```cpp
#include <iostream>

void print(int a[]){
    for (int x : a){
        std::cout << x << ' ';
    }
}

int main(){
    int arr[4] = { 1, 2, 3, 4 };
    print(arr);
}
```

main.cpp:4:18: error: cannot build range expression with array function parameter 'a' since parameter with array type 'int []' is treated as pointer type 'int *'
    for (int x : a){
    ^
main.cpp:3:16: note: declared here
void print(int a[]){
    ^
1 error generated.
#include <iostream>

void foo(int a[]){
    std::cout << "foo: " << sizeof(a) << '\n';
}

int main(){
    int arr[4] = { 1, 2, 3, 4 };  
    std::cout << "main: " << sizeof(arr) << '\n';
    foo(arr);
}

main: 16
foo: 8
```cpp
#include <iostream>
#include <vector>

int main(){
    using ivec = std::vector<int>;
    auto s = (ivec*)malloc(sizeof(ivec));

    s->push_back(1);
    s->push_back(2);
    s->push_back(3);

    for (const auto& i : *s){
        std::cout << i << ' ';
    }
}
Yay! It works
```
Dynamic Memory Allocation

What’s that? Don’t use malloc()? Okay, fine.
Dynamic Memory Allocation

What’s that? I should use “smart pointers” instead of `new`? Okay, fine.

```cpp
#include <iostream>
#include <vector>
#include <memory>

int main(){
    using ivec = std::vector<int>;
    auto s = std::unique_ptr<ivec>{};

    s->push_back(1);
    s->push_back(2);
    s->push_back(3);

    for (const auto& i : *s){
        std::cout << i << ' ';
    }
}
```

bash: line 7: 9526 Segmentation fault (core dumped) ./a.out
Dynamic Memory Allocation

What’s that? I still need to allocate memory? `std::unique_ptr` doesn’t do my work for me? That’s dumb.

Guess I’d better free the memory myself too, to avoid memory leaks.
The Dark Side of C++

Undefined Behavior
Undefined Behavior

• “Renders the entire program meaningless if certain rules of the language are violated.” [1]

• “There are no restrictions on the behavior of the program” [1]

• “Compilers are not required to diagnose undefined behavior […], and the compiled program is not required to do anything meaningful.” [1]

• “Because correct C++ programs are free of undefined behavior, compilers may produce unexpected results when a program that actually has UB is compiled with optimization enabled” [1]

• If a program encounters UB when given a set of inputs, there are no requirements on its behavior “not even with regard to operations preceding the first undefined operation” [2]

[2] C++20 Working Draft, Section 4.1.1.5
Undefined Behavior in Simpler Terms

If you do something wrong, **literally anything** can happen when your code runs.

This includes:

- Your code runs and does nothing
- Your code runs as you expect it to
- Your code crashes with a helpful error message
- Your code crashes for no explainable reason
- Your code runs as you expect it to, but fails horribly on a different compiler, different computer, different day, etc
- Your code passes all tests, but hackers can steal your passwords
- Demons come flying out of your nose
Undefined Behavior in the C++ Standard

• The word “undefined” appears 278 times in the latest C++ Standard Draft

• That’s not all:
  • “Undefined behavior may be expected when this document omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data”
Examples of Undefined Behavior

- Reading from an uninitialized variable (Note: most variables are uninitialized by default)
- Reading an array out of bounds (Note: you are usually responsible for knowing the array’s size)
- Forgetting to put a newline at the end of a source code file (until C++11)
- Dereferencing the null pointer
- Dereferencing a pointer that does not point to an object of the pointer’s type
- Returning a pointer or reference to a local variable
- Signed integer overflow (Note: this probably causes most C++ programs in existence to have UB)
- Infinite loops with no side effects
#include <iostream>

int fermat() {
    const int MAX = 1000;
    int a=1, b=1, c=1;
    // Endless loop with no side effects is UB
    while (1) {
        if (((a*a*a) == ((b*b*b)+(c*c*c)))) return 1;
        a++;
        if (a>MAX) { a=1; b++; }
        if (b>MAX) { b=1; c++; }
        if (c>MAX) { c=1; }
    }
    return 0;
}

int main() {
    if (fermat())
        std::cout << "Fermat's Last Theorem has been disproved.\n";
    else
        std::cout << "Fermat's Last Theorem has not been disproved.\n";
}

Possible output:

Fermat's Last Theorem has been disproved.
Why Compilation is Terrible

INTRODUCING THE PREPROCESSOR
The Preprocessor in C++

The C++ preprocessor is a token-replacing program that modifies your source code during lexical analysis.

The preprocessor has no concept of C++ syntax or grammar.

The preprocessor is blind to the syntax, semantics, and scoping rules of C++.

Every sensible programmer hates the C++ preprocessor passionately.

It is also the standard way to combine and reuse source code!
Preprocessor Basics: \texttt{#define}

Object-like macro  
\textit{(token is removed from source code)}

```cpp
#include <iostream>

#define return

int foo(){
    // This line gets replaced by "42;"
    // which is a no-op
    return 42;
}

int main(){
    // Yay undefined behavior!
    std::cout << foo();
}
```

Object-like macro  
\textit{(token is replaced in source code)}

```cpp
#include <iostream>

#define return throw

int foo(){
    // This line gets replaced by "throw 42;"
    return 42;
}

int main(){
    std::cout << foo();
}
```

\texttt{terminate called after throwing an instance of \textquotesingle\texttt{int}\textquotesingle}
\texttt{bash: line 7: 5829 Aborted (core dumped) ./a.out}
Preprocessor Basics: `#define`

Function-like macro
(token is replaced with list of tokens
and arguments are substituted)

```
#include <iostream>

#define min(a, b) (a < b ? a : b)

int main(){
    std::cout << min(2.5, 3);
}
```

```
#include <iostream>

#define product(a, b) a * b

int main(){
    std::cout << product(2, 1 + 1);
}
```

min(2.5, 3)
is replaced with:
2.5 < 3 ? 2.5 : 3

Expressions are evaluated twice!

product(2, 1 + 1)
is replaced with:
2 * 1 + 1

There is no encapsulation
Preprocessor Basics: \#define

```cpp
#include <iostream>
#define System S s;
#define public
#define static
#define void int
#define main(x) main()

struct F{
  void println(char* s){
    std::cout << s << std::endl;};
  struct F out;};

public static void main(String[] args) {
  System.out.println("Hello World!");
}
```

Hello World!

https://stackoverflow.com/a/653028
Macros are Blind. Macros are Evil.

A header file by Microsoft for Windows development defines two macros: `min` and `max`.

This was a very bad idea.

```cpp
#include <iostream>
#include <limits>
#include <Windows.h>

int main() {
    std::cout << "The biggest int is: " << std::numeric_limits<int>::max();
}
```

```
main.cpp:7:74: error: macro "max" requires 2 arguments, but only 1 given
    std::cout << "The biggest int is: " << std::numeric_limits<int>::max();
main.cpp:4: note: macro "max" defined here
#define max(a, b) ((a) > (b) ? (a) : (b))
```
How `#include` works

When the preprocessor encounters a line like this:

```c
#include "foo.h"
```

It literally copies and pastes the contents of that file verbatim!

<table>
<thead>
<tr>
<th>file pi.h</th>
<th>file main.cpp</th>
</tr>
</thead>
</table>
| 3.141592654 | int main() {
              |     std::cout << "There are " << (180.0 / 
              |     #include "pi.h"
              |     ) << " degrees per radian";
              | } |

This forces the compiler to frequently recompile every `#included` file. Files are typically big and include lots of other files recursively. This can cause compilation times to skyrocket.
Templates
Function Templates: Quick Intro

Note: `print<int>` and `print<double>` are fundamentally different entities
Template
Specialization

Because every instantiation of a template with different template arguments is a different entity, you can specialize templates for a certain type.
Because every instantiation of a template with different template arguments is a different entity, you can specialize templates for a certain type.

But this only works for one type at a time.

What if we want to, say, have one function for any integer and another function for everything else?

```cpp
#include <iostream>

template<typename T>
void print(T t) {
    std::cout << t << '\n';
}

void print(double) {
    std::cout << "Sorry, printing doubles is not allowed.\n";
}

int main() {
    print(202);
    print(2.02);
}
```

202
Sorry, printing doubles is not allowed.
```
#include <iostream>
#include <type_traits>

// Enable_if template for std::is_integral
template<typename T>
typename std::enable_if<std::is_integral<T>::value, void>::type
print(T t){
    std::cout << "Printing an integer: " << t << 'n';
}

// Enable_if template for negation of std::is_integral
template<typename T>
typename std::enable_if<!std::is_integral<T>::value, void>::type
print(T t){
    std::cout << "Printing something else: " << t << 'n';
}

int main(){
    print(202);
    print(2.02);
    print("This is a string");
}
```

Printing an integer: 202
Printing something else: 2.02
Printing something else: This is a string
Templates can Create Really Complicated Types

Many standard containers are templates.

This is std::vector, a resizable container:

```cpp
template<class T, class Allocator<T> = std::allocator<T>>
class vector;
```

By default, this uses T twice.

This sort of thing can lead to an exponential explosion of type complexity as you start nesting things.
Templates can Create Really Complicated Types

Code that looks like this to you:

```cpp
// 3D array of integers (could be used to represent a tensor)
std::vector<std::vector<std::vector<int>>>
```

actually looks like this to the compiler:
(and to you, once you need to read error messages)

```cpp
std::vector<std::vector<std::vector<int, std::allocator<int> > >, 
std::allocator<std::vector<int, std::allocator<int> > > >, 
std::allocator<std::vector<std::vector<int, std::allocator<int> > >, std::allocator<std::vector<int, std::allocator<int> > > > > >
```
I Found This Type While Profiling Code
Weird Syntax
char *(*(*a[]))()()  

a is an Array of pointers to functions returning pointers to functions returning pointers to char
signal is a function passing an int and a pointer to a function passing an int returning nothing (\texttt{void}) returning a pointer to a function passing an int returning nothing (\texttt{void})

\begin{verbatim}
void (*signal(int, void (*fp)(int)))(int);
\end{verbatim}
An immediately-invoked lambda returning void
Alternative Tokens
In Case You Can’t Type [ or {
Ease of Over-Engineering
How to pass a single int to a function

You have many choices

Note: a few of these will be ambiguous
How to pass many ints to a function

You have many choices

```c++
1    void foo();
2    void foo(int)
3    void foo(int, int);
4    void foo(int, int, int);
5    void foo(int, int, int, int);
6    // etc...
7
8    // DANGEROUS
9    void foo(int* array, int length);
10   void foo(std::initializer_list<int>);
11   void foo(std::vector<int>);
12
13   // DANGEROUS
14   void foo(...);
15
16   template<int...>
17   void foo();
```
How to write a member function

```cpp
class Bar {
    public:
    void foo();
};
```
How to write a member function

```cpp
class Bar {
    public:
        void foo();
        void foo() const;
};
```
How to write a member function

class Bar {
    public:
    void foo();
    void foo() const;
    void foo() volatile;
    void foo() const volatile;
};
How to write a member function

Note: none of these are ambiguous

class Bar {
    public:
    void foo() &;
    void foo() const &;
    void foo() volatile &;
    void foo() const volatile &;
    void foo() &&;
    void foo() const &&;
    void foo() volatile &&;
    void foo() const volatile &&;
};
Nearly all of these operators can be customized to do literally anything, depending on the types of a and b
Fun with Operator Overloading: Boost.Spirit Parser Generator

Suppose we want to parse these strings:

- 12345
- -12345
+12345
1 + 2
1 * 2
1/2 + 3/4
1 + 2 + 3 + 4
1 * 2 * 3 * 4
(1 + 2) * (3 + 4)
(-1 + 2) * (3 + -4)
1 + ((6 * 200) - 20) / 6
(1 + (2 + (3 + (4 + 5))))

Here’s an EBNF Specification

group ::= '(', expression, ')'
factor ::= integer | group
term ::= factor (("*" factor) | ('/' factor))*
expression ::= term (("+" term) | ('-' term))*

And here’s some C++ which returns a parser for that EBNF grammar

https://www.boost.org/doc/libs/1_67_0/libs/spirit
Fun with Operator Overloading: Analog Literals

http://www.eelis.net/C++/analogliterals.xhtml
Fun with Operator Overloading: Analog Literals