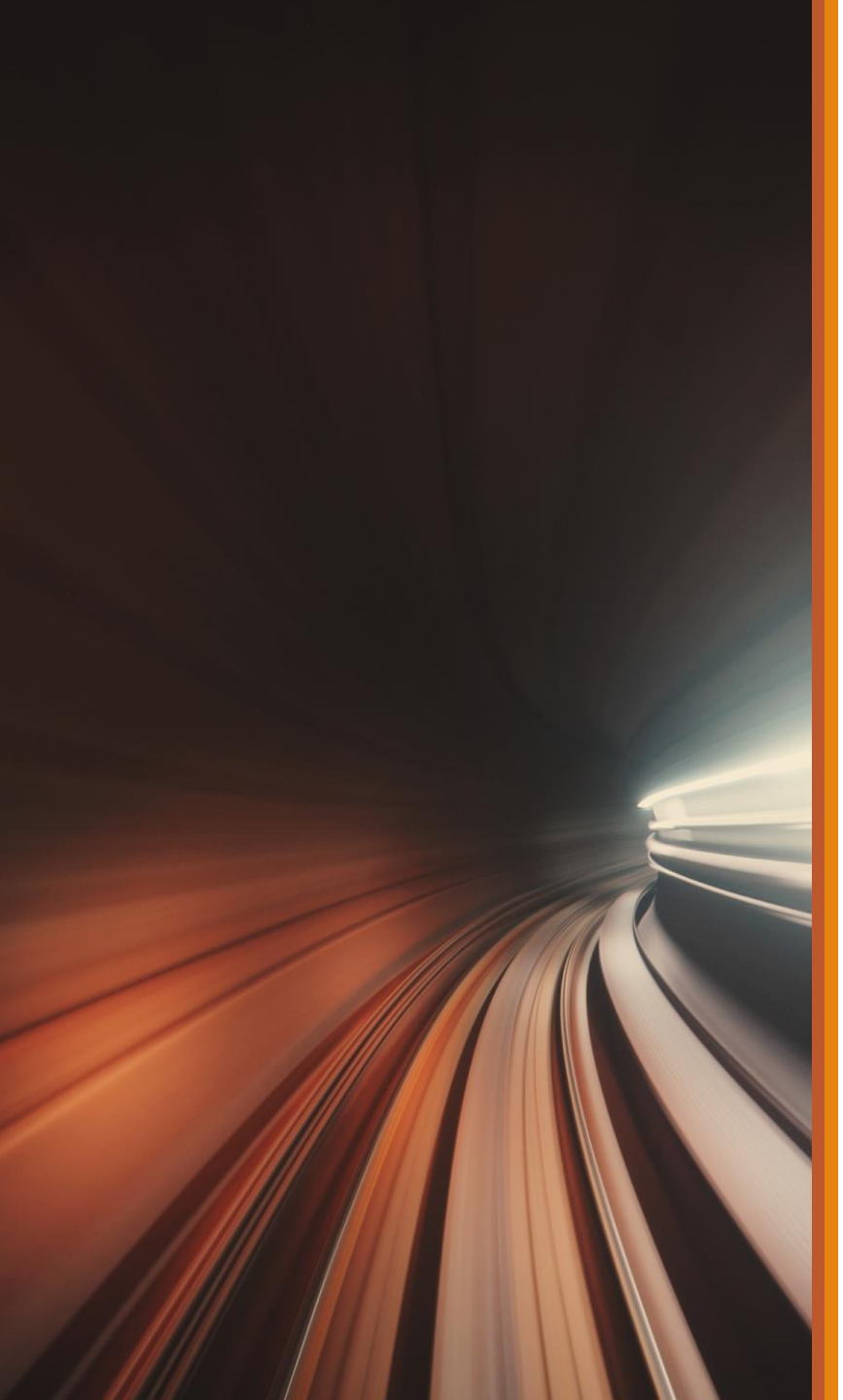
C++ Roast

PRESENTED BY **TIM STRAUBINGER**



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.

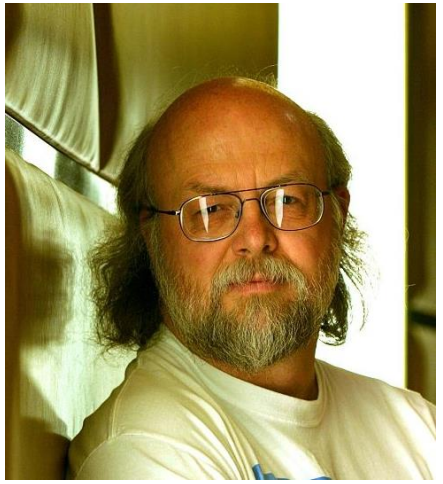
Bjarne Stroustrup (Inventor of C++) versus **James Gosling** (Inventor of Java)

"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**."

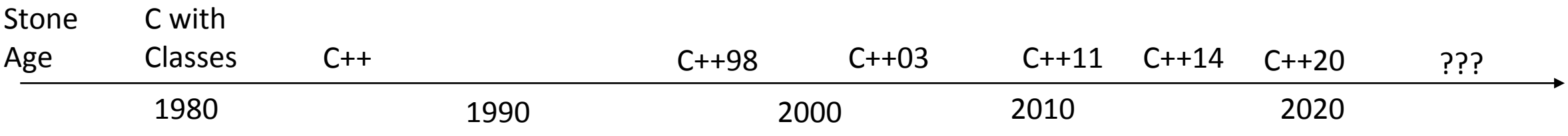
The Design and Evolution of C++

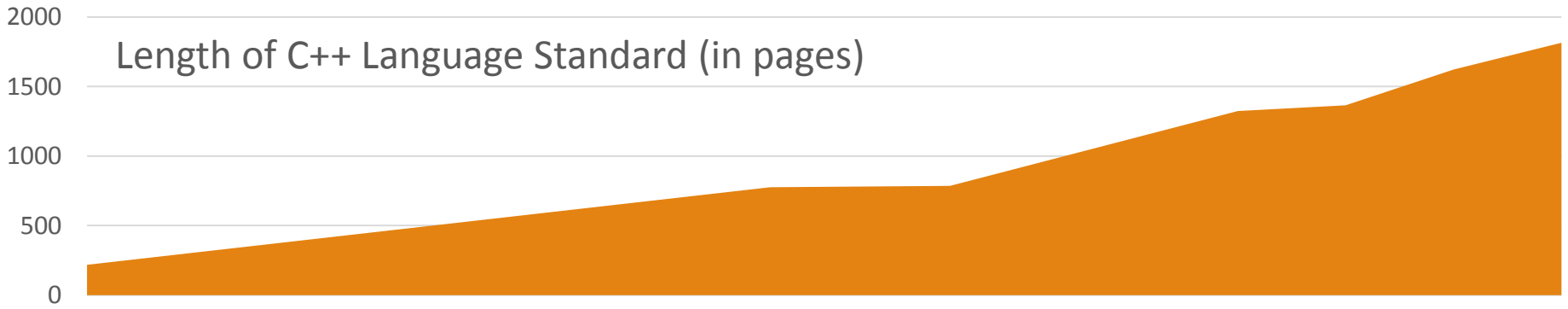
"I left out operator overloading as a **fairly personal choice** because I had seen too many people abuse it in C++."

http://www.gotw.ca/publications/c_family_interview.htm

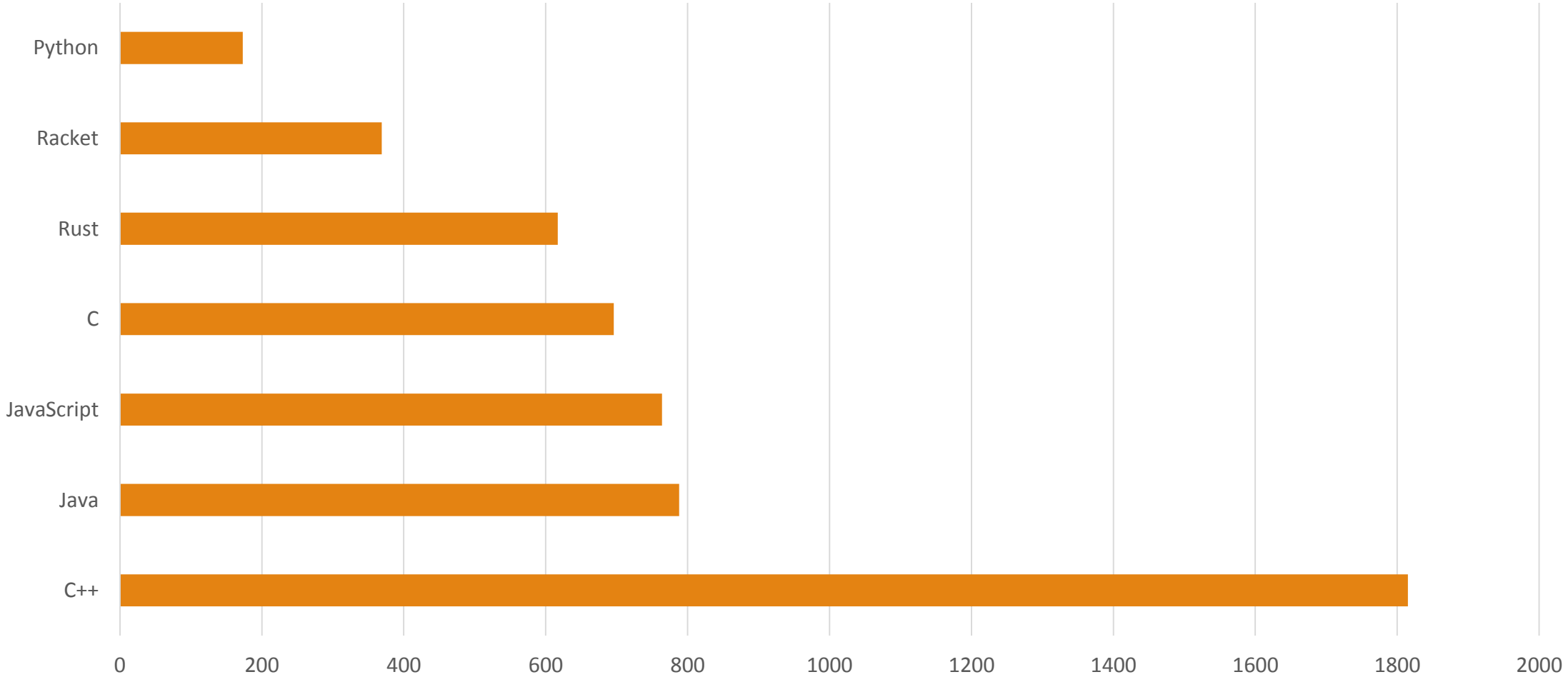


C++ is Not Done Being Invented





Length of Language Specification (Number of Pages)



Recent Versions of C++

C++11

- 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)

C++14

- Not much happened
- You can write binary numbers now: `0b1011`

C++17

- 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)

C++20

- 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

```
1  #include <iostream>
2
3  int main(){
4      std::cout << 'Hello, world!';
5  }
```

```
1919706145
```

Hello World: Attempt 2

```
1  #include <iostream>
2
3  int main(){
4      std::cout << "Hello, world!";
5  }
```

```
Hello, world!
```

How to Concatenate Strings

```
1  #include <iostream>
2
3  int main(){
4      std::cout << 'H' + 'i';
5  }
```

```
177
```

How to Concatenate Strings

```
1  #include <iostream>
2
3  int main(){
4      std::cout << "Hello, " + 'world!';
5  }
```

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

How to Concatenate Strings

```cpp
#include <iostream>

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

```
main.cpp: In function 'int main()':
main.cpp:4:28: error: invalid operands of types 'const char [8]' and 'const char [7]' to binary 'operator+'
    4 |     std::cout << "Hello, " + "world!";
      |                  ~~~~~~~~~ ^ ~~~~~~~~
      |                  |         |
      |                  |         const char [7]
      |                  const char [8]
```

How to Concatenate Strings

```
1  #include <iostream>
2
3  int main(){
4      std::cout << "Hi" + '!';
5  }
```

How to Concatenate Strings

```
1  #include <iostream>
2
3  int main(){
4      std::cout << "Hi" - '!';
5  }
```

```
_Đf.
```

How to Convert Numbers to Strings

```
1  #include <iostream>
2
3  int main(){
4      auto s1 = "Your lucky number is: ";
5      auto s2 = s1 + 10;
6      std::cout << s2;
7  }
```

```
number is:
```

How to Convert Numbers to Strings

That's kind of verbose…

```
1  #include <iostream>
2
3  int main(){
4      long int a = 36762444129608;
5      std::string s = std::to_string(a);
6      std::cout << s;
7  }
```

```
36762444129608
```

```
1  #include <iostream>
2
3  int main(){
4      long int a = 36762444129608;
5      std::string s = (char*)&a;
6      std::cout << s;
7  }
```

```
Hello!
```

How to Convert Numbers to Strings

Why not convert it directly to a `char*`?

```cpp
#include <iostream>

int main(){
    int i = 111;
    double d = 5.55;
    bool b = false;
    uint8_t u = 99;
    std::cout << i << '\n';
    std::cout << d << '\n';
    std::cout << b << '\n';
    std::cout << u << '\n';
}
```

```
111
5.55
0
c
```

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
- long long
- long long int
- signed long long
- signed long long int
- unsigned long long
- unsigned long 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::uintmax_t
- std::streamoff
- std::streamsize

Note: `std::byte` is not a number!

```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';
}
```

```
0
6.95255e-310
0
```

Working with Numbers

Numbers don’t need initial values

(compiled with –O0 on g++)

```
1  #include <iostream>
2
3  int main(){
4      int i;
5      double d;
6      bool b;
7      uint8_t u;
8      std::cout << i << '\n';
9      std::cout << d << '\n';
10     std::cout << b << '\n';
11     std::cout << u << '\n';
12 }
```

```
0
0
0
```

Working with Numbers

Increase your compiler's optimization level to get better numbers

(compiled with –O1 on g++)

```
1  #include <iostream>
2
3  int main(){
4      int i;
5      double d;
6      bool b;
7      std::uint8_t u;
8      std::cout << i << '\n';
9      std::cout << d << '\n';
10     std::cout << b << '\n';
11     std::cout << u << '\n';
12 }
```

```
718172376
0
0
```

Working with Numbers

Try a different compiler and see what works best for you

(compiled with –O2 on clang++)

```cpp
#include <iostream>

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

```
 -1826825216
0
0
```

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)'
    4 |     std::cout << std::min(2.5, 3);
```

The Entire Error Message

```
main.cpp: In function 'int main()':
main.cpp:4:33: error: no matching function for call to 'min(double, int)'
    4 |     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/ostream:38,
                 from /usr/local/include/c++/9.2.0/iostream:39,
                 from main.cpp:1:
/usr/local/include/c++/9.2.0/bits/stl_algobase.h:198:5: note: candidate: 'template<class _Tp> constexpr const _Tp& std::min(const
_Tp&, const _Tp&)'
  198 |     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')
    4 |     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/ostream:38,
                 from /usr/local/include/c++/9.2.0/iostream:39,
                 from main.cpp:1:
/usr/local/include/c++/9.2.0/bits/stl_algobase.h:246:5: note: candidate: 'template<class _Tp, class _Compare> constexpr const
_Tp& std::min(const _Tp&, const _Tp&, _Compare)'
  246 |     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')
    4 |     std::cout << std::min(2.5, 3);
      |                            ^
```

Macros to the Rescue!

```
1  #include <iostream>
2
3  #define min(a, b) (a < b ? a : b)
4
5  int main(){
6      std::cout << min(2.5, 3);
7  }
```

```
2.5
```

Hey, that works way better!

Macros to the Rescue!

Let's replace `min` with `product`

```
1  #include <iostream>
2
3  #define product(a, b) a * b
4
5  int main(){
6      std::cout << product(2, 1 + 1);
7  }
```

```
3
```

```cpp
#include <iostream>

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

```
bash: line 7:  5410 Done                    echo "ABCDEFG"
       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!";
    }
}
```

```
Yes!
```

```cpp
#include <iostream>

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

```
Yes!
```

```cpp
#include <iostream>

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

```
Yes!
```

```cpp
#include <iostream>

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

```
Yes!
```

```cpp
#include <iostream>

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

```
Yes!
```

```cpp
#include <iostream>

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

```
Yes!
```

```
1  #include <iostream>
2
3  char* foo(){
4      return "Hello, world!";
5  }
6
7  int main(){
8      std::cout << foo;
9  }
```

```
1
```

Let's Introduce Functions

```
1  #include <iostream>
2
3  char* foo(){
4      return "Hello, world!";
5  }
6
7  int main(){
8      std::cout << foo();
9  }
```

```
Hello, world!
```

Let's Introduce Functions

```
1  #include <iostream>
2
3  int foo(){
4
5  }
6
7  int main(){
8      std::cout << foo();
9  }
```

```
4196041
```

Return Values are Optional

```
1  #include <iostream>
2
3  char* foo(){
4
5  }
6
7  int main(){
8      std::cout << foo();
9  }
```

```
UH□□□□□□□□□H□p `
```

Return Values are Optional

```cpp
#include <iostream>

bool foo(){

}

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

Functions Can Be Used Anywhere

Compiled with –O2 on g++

Functions Can Be Used Anywhere

```
1  #include <iostream>
2
3  bool foo(){
4
5  }
6
7  int main(){
8      if (foo()){
9          std::cout << "true";
10     } else {
11         std::cout << "false";
12     }
13 }
```

```
bash: line 7: 22781 Illegal instruction     (core dumped) ./a.out
```

Compiled with –O0 on clang++

How to Pass Arguments to a Function

PASS BY VALUE (DEFAULT)

```
1  #include <iostream>
2
3  int foo(int x){
4      x += 10;
5      return x;
6  }
7
8  int main(){
9      std::cout << foo(22);
10 }
```

```
32
```

PASS BY REFERENCE (NOTE THE &)

```
1  #include <iostream>
2
3  void foo(int& x){
4      x += 10;
5  }
6
7  int main(){
8      int i = 22;
9      foo(i);
10     std::cout << i;
11 }
```

```
32
```

How to Return from a Function

RETURN BY VALUE

```
1  #include <iostream>
2
3  int foo(int x){
4      x += 10;
5      return x;
6  }
7
8  int main(){
9      std::cout << foo(22);
10 }
```

```
32
```

RETURN BY REFERENCE (NOTE THE &)

```
1  #include <iostream>
2
3  int& foo(int x){
4      x += 10;
5      return x;
6  }
7
8  int main(){
9      std::cout << foo(22);
10 }
```

```
bash: line 7: 12089 Done                    echo "Hello!"
     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!");
}
```

```
true
```

Arrays

```
1  #include <iostream>
2
3  int main(){
4      int arr[] = { 1, 2, 3, 4, 5 };
5      std::cout << arr;
6  }
```

```
0x7fff6d38aa30
```

Arrays

```
1  #include <iostream>
2
3  int main(){
4      int arr[] = { 1, 2, 3, 4 };
5      for (int i = 0; i <= 4; ++i){
6          std::cout << arr[i] << ' ';
7      }
8  }
```

```
1 2 3 4 593744096
```

Arrays

```
1  #include <iostream>
2
3  int main(){
4      int arr[4] = { 1, 2, 3, 4 };
5
6      for (int x : arr){
7          std::cout << x << ' ';
8      }
9  }
```

```
1 2 3 4
```

```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.
```

Passing Arrays to Functions

```cpp
#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
```

Passing Arrays to Functions

```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 << ' ';
    }
}
```

```
1 2 3
```

Dynamic Memory Allocation

Yay! It works

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

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

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

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

```
1 2 3
```

Dynamic Memory Allocation

What's that? Don't use `malloc()`? 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 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::make_unique<ivec>();

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

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

    delete s.get();
}
```

```
bash: line 7:  9949 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]

[1] https://en.cppreference.com/w/cpp/language/ub

[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

```cpp
#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.
```

Using Undefined Behavior for Great Good

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: `#define`

Object-like macro
(token is removed from source code)

```
1   #include <iostream>
2
3   #define return
4
5   int foo(){
6       // This line gets replaced by "42;"
7       // which is a no-op
8       return 42;
9   }
10
11  int main(){
12      // Yay undefined behavior!
13      std::cout << foo();
14  }
```

```
4196041
```

Object-like macro
(token is replaced in source code)

```
1   #include <iostream>
2
3   #define return throw
4
5   int foo(){
6       // This line gets replaced by "throw 42;"
7       return 42;
8   }
9
10  int main(){
11      std::cout << foo();
12  }
```

```
terminate called after throwing an instance of 'int'
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)

```
1  #include <iostream>
2
3  #define min(a, b) (a < b ? a : b)
4
5  int main(){
6      std::cout << min(2.5, 3);
7  }
```

```
2.5
```

`min(2.5, 3)`
is replaced with:
`2.5 < 3 ? 2.5 : 3`
Expressions are evaluated twice!

```
1  #include <iostream>
2
3  #define product(a, b) a * b
4
5  int main(){
6      std::cout << product(2, 1 + 1);
7  }
```

```
3
```

`product(2, 1 + 1)`
is replaced with:
`2 * 1 + 1`
There is no encapsulation

Preprocessor Basics: `#define`

```
1  #include <iostream>
2  #define System S s;s
3  #define public
4  #define static
5  #define void int
6  #define main(x) main()
7  struct F{void println(char* s){std::cout << s << std::endl;}};
8  struct S{F out;};
9
10 public static void main(String[] args) {
11   System.out.println("Hello World!");
12 }
```

```
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
    7 |     std::cout << "The biggest int is: " << std::numeric_limits<int>::max();
      |                                                                          ^
main.cpp:4: note: macro "max" defined here
    4 | #define max(a, b) ((a) > (b) ? (a) : (b))
      |
```

How `#include` works

When the preprocessor encounters a line like this:

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

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

file pi.h

```
3.141592654
```

file main.cpp

```
int main(){
    std::cout << "There are " << (180.0 /
#include "pi.h"
    ) << " degrees per radian";
}
```

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


Templates

Function Templates: Quick Intro

```cpp
#include <iostream>

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

int main(){
    print<int>(101);
    print<double>(1.01);
}
```

```
101
1.01
```

```cpp
#include <iostream>

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

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

```
202
2.02
```

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

Template Specialization

```cpp
#include <iostream>

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

template<>
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.
```

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

```cpp
#include <iostream>

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

template<>
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.
```

Template Specialization

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?

Template Specialization: S.F.I.N.A.E. Tricks

How real C++ developers overload templates

```cpp
#include <iostream>
#include <type_traits>

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

template<typename T>
typename std::enable_if<std::negation<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:

```
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:

```
// 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)

```
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

```
boost::asio::detail::executor_op<boost::asio::detail::binder2<boost::asio::detail::write_op<boost::asio::basic_stream_socket<boost::asio::ip
::tcp>,boost::beast::buffers_cat_view<boost::asio::mutable_buffer,boost::beast::buffers_prefix_view<boost::beast::buffers_suffix<std::vector
<boost::asio::const_buffer,std::allocator<boost::asio::const_buffer> > > >
>,boost::beast::buffers_cat_view<boost::asio::mutable_buffer,boost::beast::buffers_prefix_view<boost::beast::buffers_suffix<std::vector<boos
t::asio::const_buffer,std::allocator<boost::asio::const_buffer> > > >
>::const_iterator,boost::asio::detail::transfer_all_t,boost::beast::websocket::stream<boost::asio::basic_stream_socket<boost::asio::ip::tcp>
>::write_some_op<std::vector<boost::asio::const_buffer,std::allocator<boost::asio::const_buffer>
>,boost::asio::executor_binder<std::_Binder<std::_Unforced,void (__cdecl wsserver::session::*)(boost::system::error_code,unsigned __int64)
__ptr64,std::shared_ptr<wsserver::session>,std::_Ph<1> const & __ptr64,std::_Ph<2> const &
__ptr64>,boost::asio::strand<boost::asio::io_context::executor_type> > > >,boost::system::error_code,unsigned
__int64>,std::allocator<void>,boost::asio::detail::scheduler_operation>::executor_op<boost::asio::detail::binder2<boost::asio::detail::write
_op<boost::asio::basic_stream_socket<boost::asio::ip::tcp>,boost::beast::buffers_cat_view<boost::asio::mutable_buffer,boost::beast::buffers_
prefix_view<boost::beast::buffers_suffix<std::vector<boost::asio::const_buffer,std::allocator<boost::asio::const_buffer> > > >
>,boost::beast::buffers_cat_view<boost::asio::mutable_buffer,boost::beast::buffers_prefix_view<boost::beast::buffers_suffix<std::vector<boos
t::asio::const_buffer,std::allocator<boost::asio::const_buffer> > > >
>::const_iterator,boost::asio::detail::transfer_all_t,boost::beast::websocket::stream<boost::asio::basic_stream_socket<boost::asio::ip::tcp>
>::write_some_op<std::vector<boost::asio::const_buffer,std::allocator<boost::asio::const_buffer>
>,boost::asio::executor_binder<std::_Binder<std::_Unforced,void (__cdecl wsserver::session::*)(boost::system::error_code,unsigned __int64)
__ptr64,std::shared_ptr<wsserver::session>,std::_Ph<1> const & __ptr64,std::_Ph<2> const &
__ptr64>,boost::asio::strand<boost::asio::io_context::executor_type> > > >,boost::system::error_code,unsigned
__int64>,std::allocator<void>,boost::asio::detail::scheduler_operation><boost::asio::detail::binder2<boost::asio::detail::write_op<boost::as
io::basic_stream_socket<boost::asio::ip::tcp>,boost::beast::buffers_cat_view<boost::asio::mutable_buffer,boost::beast::buffers_prefix_view<b
oost::beast::buffers_suffix<std::vector<boost::asio::const_buffer,std::allocator<boost::asio::const_buffer> > > >
>,boost::beast::buffers_cat_view<boost::asio::mutable_buffer,boost::beast::buffers_prefix_view<boost::beast::buffers_suffix<std::vector<boos
t::asio::const_buffer,std::allocator<boost::asio::const_buffer> > > >
>::const_iterator,boost::asio::detail::transfer_all_t,boost::beast::websocket::stream<boost::asio::basic_stream_socket<boost::asio::ip::tcp>
>::write_some_op<std::vector<boost::asio::const_buffer,std::allocator<boost::asio::const_buffer>
>,boost::asio::executor_binder<std::_Binder<std::_Unforced,void (__cdecl wsserver::session::*)(boost::system::error_code,unsigned __int64)
__ptr64,std::shared_ptr<wsserver::session>,std::_Ph<1> const & __ptr64,std::_Ph<2> const &
__ptr64>,boost::asio::strand<boost::asio::io_context::executor_type> > > >,boost::system::error_code,unsigned __int64> >
```

Weird Syntax

```
char *(*(*a[])())()
```

a is an Array of pointers to functions returning pointers to functions returning pointers to char

```
void (*signal(int, void (*fp)(int)))(int);
```

`signal` is a function passing an `int` and a pointer to a function passing an `int` returning nothing (`void`) returning a pointer to a function passing an `int` returning nothing (`void`)

http://c-faq.com/decl/spiral.anderson.html

```
[] () {} ()
```

An immediately-invoked lambda returning void

```
%:include <iostream>

struct X
<%
    compl X() <%%> // destructor
    X() <%%>
    X(const X bitand) = delete; // copy constructor

    bool operator not_eq(const X bitand other)
    <%
       return this not_eq bitand other;
    %>
%>;

int main(int argc, char* argv<::>)
<%
    // lambda with reference-capture:
    auto greet = <:bitand:>(const char* name)
    <%
        std::cout << "Hello " << name
                  << " from " << argv<:0:> << '\n';
    %>;

    if (argc > 1 and argv<:1:> not_eq nullptr) <%
        greet(argv<:1:>);
    %>
%>
```

Alternative Tokens In Case You Can't Type [or {

Ease of Over-Engineering

How to pass a single `int` to a function

```
1   void foo(int);
2   void foo(int&);
3   void foo(int&&);
4   void foo(int*);
5   void foo(const int*);
6   void foo(const int);
7   void foo(const int&);
8   void foo(const int&&);
9   void foo(const int*);
10  void foo(const int* const);
11  template<int> void foo();
```

You have many choices

Note: a few of these will be ambiguous

```
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
11 void foo(std::initializer_list<int>);
12
13 void foo(std::vector<int>);
14
15 // DANGEROUS
16 void foo(...);
17
18 template<int...>
19 void foo();
```

How to pass many `int`s to a function

You have many choices

How to write a member function

```
1  class Bar {
2  public:
3      void foo();
4  };
```

How to write a member function

```
1  class Bar {
2  public:
3      void foo();
4      void foo() const;
5  };
```

How to write a member function

```
1  class Bar {
2  public:
3      void foo();
4      void foo() const;
5      void foo() volatile;
6      void foo() const volatile;
7  };
8
```

```
1  class Bar {
2  public:
3      void foo() &;
4      void foo() const &;
5      void foo() volatile &;
6      void foo() const volatile &;
7      void foo() &&;
8      void foo() const &&;
9      void foo() volatile &&;
10     void foo() const volatile &&;
11 };
```

How to write a member function

Note: none of these are ambiguous

Operator Overloading

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	other
a = b a += b a -= b a *= b a /= b a %= b a &= b a |= b a ^= b a <<= b a >>= b	++a --a a++ a--	+a -a a + b a - b a * b a / b a % b ~a a & b a | b a ^ b a << b a >> b	!a a && b a || b	a == b a != b a < b a > b a <= b a >= b a <=> b	a[b] *a &a a->b a.b a->*b a.*b	a(...) a, b ? :

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

```
group       = '(' >> expression >> ')';
factor      = integer | group;
term        = factor >> *(('*' >> factor) | ('/' >> factor));
expression  = term >> *(('+' >> term) | ('-' >> term));
```

https://www.boost.org/doc/libs/1_67_0/libs/spirit

```
unsigned int c = ( o-----o
                   |     !
                   !     !
                   !     !
                   o-----o ).area;

assert( c == (I-----I) * (I-------I) );
```

```
assert( ( o-----o
          |     !
          !     !
          !     !
          o-----o ).area == ( o---------o
                              |         !
                              !         !
                              o---------o ).area );
```

Fun with Operator Overloading: *Analog Literals*

http://www.eelis.net/C++/analogliterals.xhtml

Fun with Operator Overloading: Analog Literals