C++, the legals
“A C++ program is a sequence of text files (typically header and source files) that contain declarations. They undergo translation to become an executable program, which is executed when the C++ implementation calls its main function.”
C++ Keywords

“Certain words in a C++ program have special meaning, and these are known as keywords. Others can be used as identifiers. Comments are ignored during translation. Certain characters in the program have to be represented with escape sequences.”

```
const, auto, friend, false, ... ///< C++ Keywords
// comment type 1
/* comment type 2 */
/* comment type 3
BLOCK COMMENT
*/
"Hello C++ \n"; ///</< "\n" is an escape character
```
C++ Entities

“The entities of a C++ program are values, objects, references, functions, enumerators, types, class members, templates, template specializations, namespaces. Preprocessor macros are not C++ entities.”

```cpp
3.5f; // value entity
std::string str1; // object entity
namespace std; // namespace entity
void MyFunc(); // function entity
const int& a = b; // reference entity
define MyEnum {}; // enum entity
#define UGLY_MACRO(X) // NOT a C++ entity
```
C++ Declarations

“Declarations may introduce entities, associate them with names and define their properties. The declarations that define all properties required to use an entity are definitions.”

```cpp
int foo; // introduce entity named "foo"

void MyFunc(); // introduce entity named "MyFunc"

// introduce entity named "GreatFunction"
// Also, this is a definition of "GreatFunction",
void GreatFunction() {
    // do stuff
}
```
C++ Definitions

“Definitions of functions usually include sequences of statements, some of which include expressions, which specify the computations to be performed by the program.”

```cpp
1 // Function Definition
2 void MyFunction() {
3   int a; // statement
4   int b; // statement
5   int c = a + b; // a + b is an expression
6 }
```

**NOTE:** Every C++ statement ends with a semicolon “;”
Names encountered in a program are associated with the declarations that introduced them. Each name is only valid within a part of the program called its **scope**.

```
int my_variable;  // "my_variable" is the name
{
    float var_fl;  // var_fl is valid within this scope
}
var_fl;          // Error, var_fl outside its scope
int var_fl;      // Valid, var_fl not declared
```
"Each object, reference, function, expression in C++ is associated with a type, which may be fundamental, compound, or user-defined, complete or incomplete, etc."

```cpp
float a; // float is the fundamental type of a
bool b; // bool is fundamental

MyType c; // MyType is user defined, incomplete
MyType c{}; // MyType is user defined, complete

std::vector; // Also, user-defined type
std::string; // Also, user-defined type
```
“Declared objects and declared references are variables, except for non-static data members.”

```cpp
1 int foo;       // variable
2 bool know_stuff;  // also, variable
3 MyType my_var; // variable
4 MyType::var;   // static data member, variable
5 MyType.data_member; // non-static data member
```
C++ Identifiers

“An identifier is an arbitrarily long sequence of digits, underscores, lowercase and uppercase Latin letters, and most Unicode characters. A valid identifier must begin with a **non-digit**. Identifiers are case-sensitive.”

```c++
1 int s_my_var; // valid identifier
2 int S_my_var; // valid but different
3 int SMYVAR;  // also valid
4 int A_6_;    // valid
5 int Üẞ_vär;  // valid
6 int 6_a;    // NOT valid, illegal
7 int this_identifier_sadly_is_consider_valid_but_long;
```
# C++ Keywords

<table>
<thead>
<tr>
<th>alignas (since C++11)</th>
<th>default(1)</th>
<th>register(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignof (since C++11)</td>
<td>delete(1)</td>
<td>reinterpret_cast</td>
</tr>
<tr>
<td>and</td>
<td>do</td>
<td>requires</td>
</tr>
<tr>
<td>and_eq</td>
<td>double</td>
<td>return</td>
</tr>
<tr>
<td>asm</td>
<td>dynamic_cast</td>
<td>short</td>
</tr>
<tr>
<td>atomic_cancel (TM TS)</td>
<td>else</td>
<td>signed</td>
</tr>
<tr>
<td>atomic_commit (TM TS)</td>
<td>enum</td>
<td>sizeof(1)</td>
</tr>
<tr>
<td>atomic_noexcept (TM TS)</td>
<td>explicit</td>
<td>static</td>
</tr>
<tr>
<td>auto(1)</td>
<td>export(1)(3)</td>
<td>static_assert (since C++11)</td>
</tr>
<tr>
<td>bitand</td>
<td>extern(1)</td>
<td>static_cast</td>
</tr>
<tr>
<td>bitor</td>
<td>false</td>
<td>struct(1)</td>
</tr>
<tr>
<td>bool</td>
<td>float</td>
<td>switch</td>
</tr>
<tr>
<td>break</td>
<td>for</td>
<td>synchronized (TM TS)</td>
</tr>
<tr>
<td>case</td>
<td>friend</td>
<td>template</td>
</tr>
<tr>
<td>catch</td>
<td>goto</td>
<td>this</td>
</tr>
<tr>
<td>char</td>
<td>if</td>
<td>thread_local (since C++11)</td>
</tr>
<tr>
<td>char8_t (since C++20)</td>
<td>inline(1)</td>
<td>throw</td>
</tr>
<tr>
<td>char16_t (since C++11)</td>
<td>int</td>
<td>true</td>
</tr>
<tr>
<td>char32_t (since C++11)</td>
<td>long</td>
<td>try</td>
</tr>
<tr>
<td>class(1)</td>
<td>mutable(1)</td>
<td>typedef</td>
</tr>
<tr>
<td>compl</td>
<td>namespace</td>
<td>typeid</td>
</tr>
<tr>
<td>concept (since C++20)</td>
<td>new</td>
<td>typename</td>
</tr>
<tr>
<td>const</td>
<td>noexcept (since C++11)</td>
<td>union</td>
</tr>
<tr>
<td>consteval (since C++20)</td>
<td>not</td>
<td>unsigned</td>
</tr>
<tr>
<td>constexpr (since C++11)</td>
<td>not_eq</td>
<td>using(1)</td>
</tr>
<tr>
<td>constinit (since C++20)</td>
<td>nullptr</td>
<td>virtual</td>
</tr>
<tr>
<td>const_cast</td>
<td>operator</td>
<td>void</td>
</tr>
<tr>
<td>continue</td>
<td>or</td>
<td>volatile</td>
</tr>
<tr>
<td>co_await (since C++20)</td>
<td>or_eq</td>
<td>wchar_t</td>
</tr>
<tr>
<td>co_return (since C++20)</td>
<td>private</td>
<td>while</td>
</tr>
<tr>
<td>co_yield (since C++20)</td>
<td>protected</td>
<td>xor</td>
</tr>
<tr>
<td>decltype (since C++11)</td>
<td>public</td>
<td>xor_eq</td>
</tr>
</tbody>
</table>
| reflexivity (reflection TS) | }
C++ Expressions

“An expression is a sequence of operators and their operands, that specifies a computation.”

<table>
<thead>
<tr>
<th>Common operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
</tr>
<tr>
<td>a = b</td>
</tr>
<tr>
<td>a += b</td>
</tr>
<tr>
<td>a -= b</td>
</tr>
<tr>
<td>a *= b</td>
</tr>
<tr>
<td>a /= b</td>
</tr>
<tr>
<td>a %= b</td>
</tr>
<tr>
<td>a &amp;= b</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a ^= b</td>
</tr>
<tr>
<td>a &lt;&lt;= b</td>
</tr>
<tr>
<td>a &gt;&gt;= b</td>
</tr>
</tbody>
</table>
Control structures
If statement

```java
if (STATEMENT) {
    // This is executed if STATEMENT == true
} else if (OTHER_STATEMENT) {
    // This is executed if:
    // (STATEMENT == false) && (OTHER_STATEMENT == true)
} else {
    // This is executed if neither is true
}
```

- Used to conditionally execute code
- All the `else` cases can be omitted if needed
- `STATEMENT` can be any boolean expression
Switch statement

```java
switch (STATEMENT) {
    case CONST_1:
        // This runs if STATEMENT == CONST_1.
        break;
    case CONST_2:
        // This runs if STATEMENT == CONST_2.
        break;
    default:
        // This runs if no other options worked.
}
```

- Used to conditionally execute code
- Can have many `case` statements
- `break` exits the `switch` block
- `STATEMENT` usually returns `int` or `enum` value
```c
#include <stdio.h>

int main() {
    // Color could be:
    // RED  == 1
    // GREEN == 2
    // BLUE  == 3
    int color = 2;
    switch (color) {
    case 1: printf("red\n"); break;
    case 2: printf("green\n"); break;
    case 3: printf("blue\n"); break;
    }
    return 0;
}
```
Switch statement, C++ style

```cpp
#include <iostream>

int main() {
    enum class RGB { RED, GREEN, BLUE };  
    RGB color = RGB::GREEN;

    switch (color) {
    case RGB::RED: std::cout << "red\n"; break;
    case RGB::GREEN: std::cout << "green\n"; break;
    case RGB::BLUE: std::cout << "blue\n"; break;
    }
    return 0;
}
```
While loop

```java
1 while (STATEMENT) {
2     // Loop while STATEMENT == true.
3 }
```

Example **while** loop:

```java
1 bool condition = true;
2 while (condition) {
3     condition = /* Magically update condition. */
4 }
```

- Usually used when the exact number of iterations is unknown before-wise
- Easy to form an endless loop by mistake
For loop

```
for (INITIAL_CONDITION; END_CONDITION; INCREMENT) {
  // This happens until END_CONDITION == false
}
```

Example `for` loop:

```
for (int i = 0; i < COUNT; ++i) {
  // This happens COUNT times.
}
```

- In C++ `for` loops are very fast. Use them!
- Less flexible than `while` but less error-prone
- Use `for` when number of iterations is fixed and `while` otherwise
Range for loop

- Iterating over a standard containers like `array` or `vector` has simpler syntax
- Avoid mistakes with indices
- Show intent with the syntax
- Has been added in C++ 11

```cpp
for (const auto& value : container) {
  // This happens for each value in the container.
}
```
New in C++ 17

```cpp
std::map<char, int> my_dict{{'a', 27}, {'b', 3}};
for (const auto& [key, value] : my_dict) {
    cout << key << " has value " << value << endl;
}
```

Similar to

```python
my_dict = {'a': 27, 'b': 3}
for key, value in my_dict.items():
    print(key, "has value", value)
```
The C++ is $\approx 15$ times faster than Python
Exit loops and iterations

- We have control over loop iterations
- Use `break` to exit the loop
- Use `continue` to skip to next iteration

```cpp
while (true) {
    int i = /* Magically get new int. */
    if (i % 2 == 0) {
        cerr << i << endl;
    } else {
        break;
    }
}
```
Built-in types
Built-in types

“Out of the box” types in C++:

```cpp
1 bool this_is_fun = true;  // Boolean: true or false.
2 char carret_return = '\n';  // Single character.
3 int meaning_of_life = 42;  // Integer number.
4 short smaller_int = 42;  // Short number.
5 long bigger_int = 42;  // Long number.
6 float fraction = 0.01f;  // Single precision float.
7 double precise_num = 0.01;  // Double precision float.
8 auto some_int = 13;  // Automatic type [int].
9 auto some_float = 13.0f;  // Automatic type [float].
10 auto some_double = 13.0;  // Automatic type [double].
```

[Reference]
Operations on arithmetic types

- All **character**, **integer** and **floating point** types are arithmetic
- Arithmetic operations: +, -, *, /
- Comparisons <, >, <=, >=, == return `bool`
- `a += 1 ⇔ a = a + 1`, same for -=, *=, /=, etc.
- Avoid `==` for floating point types

[Reference]
Are we crazy?

```cpp
#include <iostream>

int main() {
    // Create an innocent float variable
    const float var = 84.78;

    // Let's compare the same number, they should be the same...
    const bool cmp_result = (84.78 == var);
    std::cout << "84.78 equal to " << var << "???\n"
               << std::boolalpha << cmp_result << '\n';
    return 0;
}
```

true or false ???
Some additional operations

- **Boolean variables** have logical operations:
  - **or**: `||`, **and**: `&&`, **not**: `!`

```c
1 bool is_happy = (!is_hungry && is_warm) || is_rich
```

- **Additional operations on integer variables**:
  - `/` is integer division: i.e. `7 / 3 == 2`
  - `%` is modulo division: i.e. `7 % 3 == 1`
  - **Increment** operator: `a++ ⇔ ++a ⇔ a += 1`
  - **Decrement** operator: `a-- ⇔ --a ⇔ a -= 1`
  - Do not use de- increment operators within another expression, i.e. `a = (a++) + ++b`
Variables
Declaring variables

Variable declaration always follows pattern:
\[ \text{<TYPE>} \ \text{<NAME>} \ \ [ = \ <\text{VALUE}>] ; \]

- Every variable has a type
- Variables cannot change their type
- **Always initialize** variables if you can

```
1 bool sad_uninitialized_var;
2 bool initializing_is_good = true;
```
Naming variables

- Name **must** start with a letter
- Give variables **meaningful names**
- Don’t be afraid to **use longer names**
- **Don’t include type** in the name
- **Don’t use negation** in the name
- **GOOGLE-STYLE** name variables in **snake_case** all lowercase, underscores separate words
- **C++** is case sensitive: **some_var** is different from **some_Var**

Google naming rules: [https://google.github.io/styleguide/cppguide.html#General_Naming_Rules](https://google.github.io/styleguide/cppguide.html#General_Naming_Rules)
Variables live in scopes

- There is a single global scope
- Local scopes start with `{` and ends with `}`
- All variables **belong to the scope** where they have been declared
- All variables die in the end of **their** scope
- This is the core of C++ memory system

```c++
int main() { // Start of main scope.
    float some_float = 13.13f; // Create variable.
    { // New inner scope.
        auto another_float = some_float; // Copy variable.
    } // another_float dies.
    return 0;
} // some_float dies.
```
Any variable can be const

- Use `const` to declare a `constant`
- The compiler will guard it from any changes
- Keyword `const` can be used with any type
- `GOOGLE-STYLE` name constants in CamelCase starting with a small letter `k`:
  - `const` float `kImportantFloat` = 20.0f;
  - `const` int `kSomeInt` = 20;
  - `const` std::string `kHello` = "hello"

- `const` is part of type: variable `kSomeInt` has type `const` int
- **Tip:** declare everything `const` unless it must be changed
References to variables

- We can create a reference to any variable
- Use & to state that a variable is a reference
  - float& ref = original_variable;
  - std::string& hello_ref = hello;
- Reference is part of type: variable ref has type float&
- Whatever happens to a reference happens to the variable and vice versa
- Yields performance gain as references avoid copying data
Const with references

- References are fast but reduce control
- To avoid unwanted changes use `const`
  - `const float& ref = original_variable;`
  - `const std::string& hello_ref = hello;`

```cpp
#include <iostream>
using namespace std;

int main() {
    int num = 42;  // Name has to fit on slides
    int& ref = num;
    const int& kRef = num;
    ref = 0;
    cout << ref << " " << num << " " << kRef << endl;
    num = 42;
    cout << ref << " " << num << " " << kRef << endl;
    return 0;
}
```
Streams
I/O streams (Lecture 0)

- Handle `stdin`, `stdout` and `stderr`:
  - `std::cin` — maps to `stdin`
  - `std::cout` — maps to `stdout`
  - `std::cerr` — maps to `stderr`
- `#include <iostream>` to use I/O streams
- Part of C++ standard library

```cpp
#include <iostream>

int main() {
    int some_number;
    std::cout << "please input any number" << std::endl;
    std::cin >> some_number;
    std::cout << "number = " << some_number << std::endl;
    std::cerr << "boring error message" << std::endl;
    return 0;
}
```
What does this program do?

```c
#include <stdio.h>
#include <string.h>

int main() {
    char filename[] = "00205.txt";
    char *pch;
    pch = strtok(filename, ".");
    while (pch != NULL) {
        printf("%s\n", pch);
        pch = strtok(NULL, ".");
    }
    return 0;
}
```
String streams

Already known streams:
- Standard output: cerr, cout
- Standard input: cin
- Filestreams: fstream, ifstream, ofstream

New type of stream: stringstream
- Combine int, double, string, etc. into a single string
- Break up strings into int, double, string etc.
```cpp
#include <iomanip>
#include <iostream>
#include <sstream>
using namespace std;

int main() {
    // Combine variables into a stringstream.
    stringstream filename{ "00205.txt"};

    // Create variables to split the string stream
    int num = 0;
    string ext;

    // Split the string stream using simple syntax
    filename >> num >> ext;

    // Tell your friends
    cout << "Number is: " << num << endl;
    cout << "Extension is: " << ext << endl;
    return 0;
}
```
Program input parameters

- Originate from the declaration of main function
- Allow passing arguments to the binary
  ```
  int main(int argc, char const *argv[]);
  ```
  - `argc` defines number of input parameters
  - `argv` is an array of string parameters
  - By default:
    - `argc == 1`
    - `argv == "<binary_path>"`

Program input parameters

```cpp
#include <iostream>
#include <string>
using std::cout;
using std::endl;

int main(int argc, char const *argv[]) {
    // Print how many parameters we received
    cout << "Got " << argc << " params\n";

    // First program argument is always the program name
    cout << "Program: " << argv[0] << endl;

    for (int i = 1; i < argc; ++i) { // from 1 on
        cout << "Param: " << argv[i] << endl;
    }
    return 0;
}
```
Suggested Video

“Give me 15 minutes & I’ll change your view of GDB”

https://youtu.be/PorfLSr3DDI
References

C++ language
This is a reference of the core C++ language constructs.

Basic concepts
Comments
ASCII chart
Names and identifiers
Types - Fundamental types
Object - Scope - Lifetime
Definitions and DGR
Name lookup
qualified - unqualified
As-if rule
Undefined behavior
Memory model and data races
Phases of translation
The main() function
Modules (C++20)

C++ Keywords
Preprocessor
#if - #ifdef - #else - #endif
#define - ## - #include
#error - offsetof

Expressions
Value categories
Evaluation order and sequencing
Constant expressions
Operators
assignment - arithmetic
increment and decrement
logical - comparison
member access and indirect
call, comma, ternary
sizeof - alignas(C++11)
new - delete - typeid
Operator overloading
Default comparisons(C++10)
Operator precedence
Conversions
implicit - explicit - user-defined
static_cast - dynamic_cast
const_cast - reinterpret_cast
Literals
boolean - integer - floating
character - string
nullptr(C++11)
user-defined (C++11)

Declaration
Namespace declaration
Namespace alias
Lexical and value references
Pointers - Arrays
Structured bindings (C++17)
Enumerations and enumerators
Storage duration and linkage
Language linkage
inline specifier
inline assembly
constexpr(C++11)
constexpr(C++20) - constexpr(C++20)
dcltype(C++11) - auto(C++11)
dcltype(C++11) - void(C++11)
typedef - Type alias(C++11)
Elaborated type specifiers
Attributes(C++11)

Initialization
Default initialization
Value initialization(C++03)
Copy initialization
Direct initialization
Aggregate initialization
List initialization(C++11)
Reference initialization
Static non-local initialization
zero - constant
Dynamic non-local initialization
ordered - unordered
Copy elision

Functions
Function declaration
Default arguments
Variadic arguments
Lambda expressions(C++11)
Argument-dependent lookup
Overload resolution
Operator overloading
Address of an overload set
Covariance (C++20)

Statements
if - switch
for - range-for(C++11)
while - do-while
continue - break - goto - return
synchronized and atomic (TM TS)

Classes
Class types - Union types
injected-class-name
Data members - Member functions
Static members - Nested classes
Derived class - using-declaration
Empty base optimization
Virtual function - Abstract class
override(C++11) - final(C++11)
Member access - friend
Bit fields - The this pointer
Constructors and member initializer lists
Default constructor - Destructor
Copy constructor - Copy assignment
Move constructor - Move assignment(C++11)
Move assignment(C++11)
Converting constructor - explicit specifier

Templates
Template parameters and arguments
Class template - Function template
Class member template
Variable template(C++14)
Template argument deduction
Explicit specialization
Class template argument deduction(C++17)
Partial specialization
Parameter packs(C++11) - sizeof...(C++11)
Fold-expressions(C++17)
Dependent names - SFINAE
Constraints and concepts (C++20)

Exceptions
throw-expression
try-catch block
function - try-block
noexcept specifier(C++11)
noexcept operator(C++11)
Dynamic exception specification (until C++17)

Miscellaneous
History of C++
Extending the namespace std
Acronyms

Idioms
Resource acquisition is initialization
Rule of three/five/zero
Pointer to implementation