

Modern C++ for Computer Vision and Image Processing

Lecture 2: Core C++

Ignacio Vizzo and Cyrill Stachniss

C++, the legals



C++ Program

“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

Identifiers → letter, underscore, digit
→ මෙයින්ගේ නොමැති නුගේ නැම්
Comments → සෘංචා ලේඛ්‍ර, — අක්‍රියා

```
int minutes = 30;
```

“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**.”

```
1 const, auto, friend, false, ... // < C++ Keywords
2 // comment type 1
3 /* comment type 2 */
4 /* comment type 3
   BLOCK COMMENT
5 */
6 */
7 "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."

```
1 3.5f;           // value entity
2 std::string str1; // object entity
3 namespace std;  // namespace entity
4 void MyFunc(); // function entity
5 const int& a = b; // reference entity
6 enum MyEnum {};// enum entity
7 #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.”

```
1 int foo;           // introduce entity named "foo"
2
3 void MyFunc(); // introduce entity named "MyFunc"
4
5 // introduce entity named "GreatFunction"
6 // Also, this is a definition of "GreatFunction",
7 void GreatFunction() {
8     // do stuff
9 }
```

Function

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.”

```
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 “;”

C++ Names

Scope തെളിവും മനസ്സിലുണ്ട്

"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**."

```
1 int my_variable; // "my_variable" is the name
2
3 { Scope           // {<-this defines a new scope
4   float var_fl;    // var_f is valid within this scope
5 } End of scope // }<-this defines end of the scope
6
7 var_fl; Not declared // Error, var_fl outside its scope
8
9 int var_fl;          // Valid, var_fl not declared OK
```

C++ Types

composite data type →
(වෙත දත්ත සැකිල්ල යොමු)

Fundamental
Compound
User defined
Complete
Incomplete

“Each object, reference, function, expression in C++ is associated with a **type**, which may be **fundamental**, **compound**, or **user-defined**, **complete** or **incomplete**, etc.”

```
1 float a;           // float is the fundamental type of a
2 bool b;            // bool is fundamental
3
4 MyType c;          // MyType is user defined, incomplete
5 MyType c{};         // MyType is user defined, complete
6
7 std::vector;        // Also, user-defined type
8 std::string;        // Also, user-defined type
```

even Functions, Arrays, Pointers, References, Enum, Class

C++ Variables

Local var object दृ. व्यक्ति

Class वर्गी. Non-Static

data member व्यक्ति वर्गी

जोही object, reference declare होय
variable निर्भावना.

“Declared objects and declared references are variables, except for **non-static** data members.”

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

Static \leftrightarrow Allocated for the lifeline of program

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."

```
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 Ü_ß_vär;     // valid
6 int 6_a;         // NOT valid, illegal ✗
7 int this_identifier_sadly_is_consider_valid_but_long;
```

C++ Keywords

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

C++ Expressions

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

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

Control structures

If statement

```
1 if (STATEMENT) {  
2     // This is executed if STATEMENT == true  
3 } else if (OTHER_STATEMENT) {  
4     // This is executed if:  
5     // (STATEMENT == false) && (OTHER_STATEMENT == true)  
6 } else {  
7     // This is executed if neither is true  
8 }
```

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

Switch statement

Int ဒီပါနပ်နမ်နာမျက်နှာ
မြန်မာ?

```
1 switch(STATEMENT) {  
2     case CONST_1:  
3         // This runs if STATEMENT == CONST_1.  
4         break;  
5     case CONST_2:  
6         // This runs if STATEMENT == CONST_2.  
7         break;  
8     default:  
9         // This runs if no other options worked.  
10 }
```

- Used to conditionally execute code
- Can have many `case` statements
- `break` exits the `switch` block
- `STATEMENT` usually returns `int` or `enum` value

Switch statement, C style

```
1 #include <stdio.h>
2 int main() {
3     // Color could be:
4     // RED    == 1
5     // GREEN  == 2
6     // BLUE   == 3
7     int color = 2;
8     switch (color) {
9         case 1: printf("red\n"); break;
10        case 2: printf("green\n"); break;
11        case 3: printf("blue\n"); break;
12    }
13    return 0;
14 }
```

Type *Unsafely* *may be* *1000^{is}* *statement.*

cd 01-type-safe

Switch statement, C++ style

Type Safe

```
1 #include <iostream>
2
3 int main() {
4     enum class RGB { RED, GREEN, BLUE };
5     RGB color = RGB::GREEN; ↖
6
7     switch (color) {
8         case RGB::RED:    std::cout << "red\n"; break;
9         case RGB::GREEN: std::cout << "green\n"; break;
10        case RGB::BLUE:  std::cout << "blue\n"; break;
11    }
12    return 0;
13 }
```

While loop

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

Example `while` loop:

```
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

```
1 for (INITIAL_CONDITION; END_CONDITION; INCREMENT) {  
2     // This happens until END_CONDITION == false  
3 }
```

Example `for` loop:

```
1 for (int i = 0; i < COUNT; ++i) {  
2     // This happens COUNT times.  
3 }
```

- 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

C++11

Container boy

ಡಿಕ್ಸಾಕ್ಟ್ ಡಿಯಂ ಫರ್ ಮೆನ್‌ ಇಟ್‌

- 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

ಡಿಕ್ಸಾಕ್ಟ್
ನುಂಟೊಬ್

```
1 for (const auto& value : container) {  
2     // This happens for each value in the container.  
3 }
```



Spoiler Alert

New in C++ 17

Range For loop
Iterating a container

```
1 std::map<char, int> my_dict{{'a', 27}, {'b', 3}};  
2 for (const auto& [key, value] : my_dict) {  
3     cout << key << " has value " << value << endl;  
4 }
```

↑
Similar to Python ↓

```
1 my_dict = {'a': 27, 'b': 3}  
2 for key, value in my_dict.items():  
3     print(key, "has value", value)
```

Spoiler Alert 2

Range For Loop (ಉದ್ದೇಶ) Contains Iteration
vs Python R. Loop ಒಂದು 15 ಗಳಲ್ಲಿ

The C++ is \approx 15 times faster than Python

```
/tmp/map_bench ./main.cpp
benchmarking ./main.cpp
time          1.971 ms  (1.882 ms .. 2.151 ms)
mean          0.998 μs  (0.987 μs .. 0.999 μs)
std dev       2.087 ms  (2.031 ms .. 2.178 ms)
variance introduced by outliers: 74% (severely inflated)
```

```
/tmp/map_bench ./main.py
benchmarking ./main.py
time          32.71 ms  (31.41 ms .. 34.14 ms)
mean          0.995 μs  (0.992 μs .. 0.997 μs)
std dev       32.31 ms  (31.79 ms .. 33.08 ms)
variance introduced by outliers: 11% (moderately inflated)
```

Exit loops and iterations

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

```
1 while (true) {  
2     int i = /* Magically get new int. */  
3     if (i % 2 == 0) {  
4         cerr << i << endl;  
5     } else {  
6         break;  
7     }  
8 }
```

Built-in types

Built-in types

Floating point binary representation?

“Out of the box” types in C++:

```
1 bool this_is_fun = true;      // Boolean: true or false.  
2 char caret_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]

<http://en.cppreference.com/w/cpp/language/types>

Operations on arithmetic types

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

[Reference]

https://en.cppreference.com/w/cpp/language/arithmetic_types

Why?

Floating point හැසිඡෙනු == මුදලමාත්‍රී පෙනීමෙහි
මෙතැනු නේ?

Are we crazy?

cd osfloat

$$\frac{1}{3} \Rightarrow 0.\overline{3}$$

```
1 #include <iostream>
2 int main() {
3     // Create an innocent float variable
4     const float var = 84.78;
5
6     // Let's compare the same number, they should be the
7     // same...
8     const bool cmp_result = (84.78 == var);
9     std::cout << "84.78 equal to " << var << "???\n"
10    << std::boolalpha << cmp_result << '\n';
11 }
```

Code ഒരു ഭാഷയിൽ.

അതിന്റെ ശ്രദ്ധിക്കുന്ന

true or false ???

floating point നി ബിനറി രീഫറൻസ് പരിപാലന പരമ്പരയിൽ പരിപാലന ചെയ്യുന്നത്



Some additional operations

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

```
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`

cd or float

Coding Horror image from Code Complete 2 book by Steve McConnell



++(a+b) ⇒ Error

Variables

Declaring variables

Variable declaration always follows pattern:

<TYPE> <NAME> [= <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;
```

Initialize මෙයුමෙන්ද?

Naming variables

→ hasNoValue vs hasValue

- 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

වැඩි ටොටුනු ඇත්තා snake-case සඳහා

Google 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

```
1 int main() { // Start of main scope.  
2     float some_float = 13.13f; // Create variable.  
3     { // New inner scope.  
4         auto another_float = some_float; // Copy variable.  
5     } // another_float dies.  
6     return 0;  
7 } // 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 starting with a small letter `k`:

- `const float kImportantFloat = 20.0f;`
- `const int kSomeInt = 20;`
- `const std::string kHello = "hello";`

کے ایجاد

- **const is part of type:**

variable `kSomeInt` has type `const int`

- **Tip:** Variable တွေကို အနဲ့ဖြစ်ဖို့ အသာဆုံး constant တွေဟာဖော်ပါ။
must constant variable တွေကို အသာဆုံး ရေးလေ့ရှိ၏
CamelCase မူ法း မှာ အသုည်တွေပေါ်၏

References to variables

for same memory address නිම්මානී.
බැංකුවේ තැබුණු.

Address Of = &

- 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

↑
අල්බුම්ස්ස්

part of type
int = brg = a;
address of
int * = z = dc

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;`

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int num = 42; // Name has to fit on slides
5     int& ref = num;
6     const int& kRef = num;
7     ref = 0;✓
8     cout << ref << " " << num << " " << kRef << endl;
9     num = 42;✓
10    cout << ref << " " << num << " " << kRef << endl;
11    return 0;
12 }
```

reference മരും ക്രമിച്ച നാടിപറ്റി. തന്റെ സൗക്ഷ്മികതയും const വിളംബിക്കാം

Streams

(Program හි I/O Devices යුතු කළ
කට්ඨා / කෘෂී ප්‍රති තේ characters sequence
න් streams නිසැක කි)

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

```
1 #include <iostream>
2 int main() {
3     int some_number;
4     std::cout << "please input any number" << std::endl;
5     std::cin >> some_number;
6     std::cout << "number = " << some_number << std::endl;
7     std::cerr << "boring error message" << std::endl;
8     return 0;
9 }
```

What does this program do?

C କୋଡ଼

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     char filename[] = "00205.txt";
6     char *pch; break a string into a series
7     pch = strtok(filename, "."); delimiters list
8     while (pch != NULL) {
9         printf("%s\n", pch);
10        pch = strtok(NULL, "."); optional delimiter is used.
11    }
12    return 0;
13 }
```

Stream କୋଡ଼ ପାଇଁ ବ୍ୟାଖ୍ୟାନ.

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. ↳

```
1 #include <iomanip> ← setw of iomanip
2 #include <iostream>
3 #include <sstream>
4 using namespace std;
5
6 int main() {
7     // Combine variables into a stringstream.
8     stringstream filename{"00205.txt"};
9
10    // Create variables to split the string stream
11    int num = 0;
12    string ext;
13
14    // Split the string stream using simple syntax
15    filename >> num >> ext;
16
17    // Tell your friends
18    cout << "Number is: " << num << endl;
19    cout << "Extension is: " << ext << endl;
20    return 0;
21 }
```

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>"`

`int main (int argc, char *argv [])`
`int main (int argc, char **argv [])`

Program input parameters

```
1 #include <iostream>
2 #include <string>
3 using std::cout;      base run pkg exec arg1 arg2
4 using std::endl;
5
6 int main(int argc, char const *argv[]) {
7     // Print how many parameteres we received
8     cout << "Got " << argc << " params\n";
9
10    // First program argument is always the program name
11    cout << "Program: " << argv[0] << endl;
12
13    for (int i = 1; i < argc; ++i) { // from 1 on
14        cout << "Param: " << argv[i] << endl;
15    }
16    return 0;
17 }
```

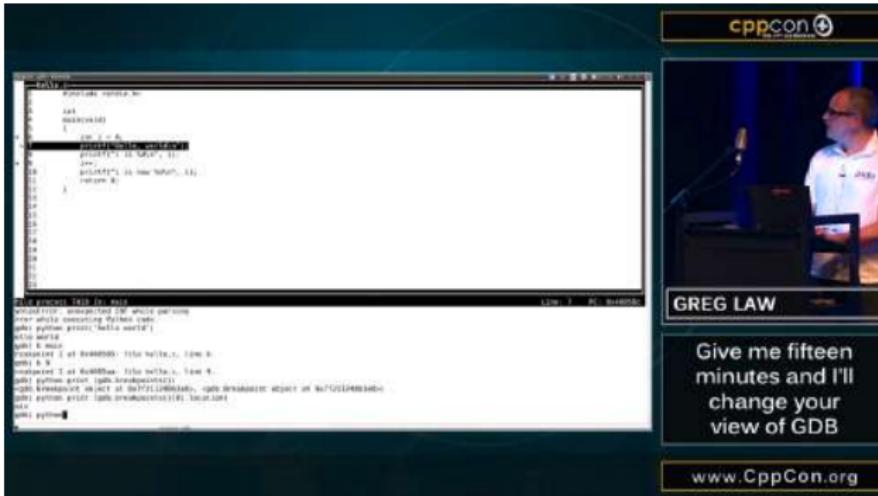
Annotations on the right side of the code:

- Handwritten text: "command arg1 arg2" is written above the first two arguments.
- Yellow bracket: A bracket groups the first two arguments, labeled "arg1 arg2".
- Yellow circle: A circle surrounds the index "0" in the expression "argv[0]".
- Yellow arrow: An arrow points from the index "0" to the value "arg1".
- Yellow bracket: A bracket groups the last two arguments, labeled "arg1 arg2".
- Yellow arrow: An arrow points from the index "1" to the value "arg2".

Suggested Video

GDB → Debugger
Valgrind → mem check
debugger date

"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 ODR
Base classes
qualified - unqualified
As-if rule
Undefined behavior
Memory model and data races
Phases of translation
The `main` function
Modularity

C++ Keywords

`if` - `else` - `while` - `and`
`else if` - `#endif`
`using` - `#pragma` - `inline`

Expressions

Value categories
Evaluation order and sequencing
Constant expressions
Operator overloading
assignment - arithmetic
increment and decrement
logical - comparison
dereferencing and induction
call, comma, ternary
`sizeof` - `alignof`
`new` - `delete` - typeid
Operator overloading
Default comparison (`<=`)
operator precedence
Conversion
`implicit` - `explicit` - user-defined
`static cast` - `dynamic cast`
`const cast` - `reinterpret cast`
Ubreak
Booleans - integer - floating
characters - strings
`multimap` (`<=`)
user-defined (`<=`)

Declaration

Namespace declaration
Namespace alias
Local and module references
Pointers - Arrays
Structured bindings (`=>`)
Enumerations and enumerators
Scope resolution and linkage
Language linkage
`int`/`long` specifier
Inline assembly
`constexpr`
`constexpr` (`<=`)
`constexpr` (`=>`) - `constinit` (`<=`)
`old` (`Type` (`<=`) - `auto`) (`<=`)
`alignspec` (`<=`)
`typeid` - `Type` (`<=`) (`<=`)
Elaborated type specifiers
`attribute` (`<=`)
`static` - `assert` (`<=`)

Initialization

`default` (`initialization`)
`Value` (`initialization`) (`<=`)
`Copy` (`initialization`)
`Direct` (`initialization`)
`Aggregate` (`initialization`)
`List` (`initialization`) (`<=`)
`Reference` (`initialization`)
`Static` non-local `initialization`
`zero` - `constant`
`Dynamic` non-local `initialization`
`android` - `uninitialized`
`Copy` (`initialization`)

Functions

`function` (`declaration`)
`Default` (`arguments`)
`Variable` (`arguments`)
`Lambda` (`declaration`) (`<=`)
Argument-dependent lookup
Overload resolution
Operator overloading
Address of an overload set
`Coroutines` (`<=`)

Statements

`if` - `switch`
`for` - `range` (`for` (`<=`))
`while` - `do` `while`
`continue` - `break` - `goto` - `return`
Synchronized and atomic (`rw`)

Classes

Class types - Union types
`struct`-`class`-`name`
Data members - Member functions
Static members - Related classes
Derived class - Virtual declaration
Empty base optimization
Virtual inheritance
`override` (`<=`) - `final` (`<=`)
Member access - friend
Bit fields - The `this` pointer
Constructors and member initializer lists
Default constructor - Destructor
Copy construction - Copy assignment
Move construction - Move assignment
Move assignment (`<=`)
Converting constructor - `explicit` specifier

Templates

Template parameters and arguments
Class template - Function template
Class member template
Variable template (`<=`)
Template argument deduction
Explicit specialization
Class template argument deduction (`<=`)
Type deduction (`<=`)
Parameter pack (`<=`) - `Sized...`, `...C...<=`
Fold-expression (`<=`)
Dependent names - SFINAE
Constraints and concepts (`<=`)

Exceptions

`throw` expression
`try`-`catch` block
`function`-`try`-`block`
`noexcept` (`declaration`) (`<=`)
`noexcept` (`operator`) (`<=`)
Dynamic exception specification (`<=`)

Miscellaneous

History of C++
Extending the namespace and
Acronyms

Idioms

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

<https://en.cppreference.com/w/cpp/language>