Modern C++ for Computer Vision
Lecture 2: C++ Basic Syntax

Ignacio Vizzo, Rodrigo Marcuzzi, 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

"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
5    BLOCK COMMENT
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 enityty
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 }
```

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

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

C++ Types

"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-defind type
8 std::string;    // Also, user-defind type
```

- C++ is a strongly typed language.

C++ Variables

"Declared objects and declared references are variables, exepct 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
```

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, ilegal
7 int this_identifier_sadly_is_consider_valid_but_long;
```

C++ Keywords

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

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
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 ? :

If statement

```
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

```
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

Switch statement, Naive

```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

```
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

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

Example `while` loop:

```
bool condition = true;
while (condition) {
  condition = /* Magically update condition. */
}
```

- Usually used when the exact number of iterations is unknown before-hand
- 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

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

Spoiler Alert

New in C++ 17

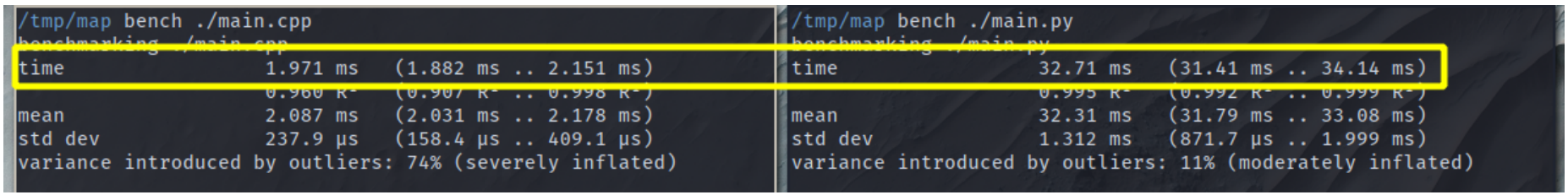
```cpp
std::map<char, int> my_dict{{'a', 27}, {'b', 3}};
for (const auto& [key, value] : my_dict) {
  cout << key << " has value " << value << endl;
}
```

Similar to

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

Spoiler Alert 2

The C++ is ≈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

```
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

"Out of the box" types in C++:

```
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].
11 std::array<int, 3> arr = {1, 2, 3};// Array of integers
```

[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` ⇔ `a = a + 1`, same for `-=`, `*=`, `/=`, etc.
- Avoid `==` for floating point types

[Reference]

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

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`

Coding Horror image from Code Complete 2 book by Steve McConnell

C-style strings are evil

Like everyhing else in C in general.

```
1  #include <cstring>
2  #include <iostream>
3
4  int main() {
5    const char source[] = "Copy this!";
6    char dest[5];
7    std::cout << source << '\n';
8
9    std::strcpy(dest, source);
10   std::cout << dest << '\n';
11
12   // source is const, no problem right?
13   std::cout << source << '\n';
14
15   return 0;
16 }
```

Strings

- `#include <string>` to use `std::string`
- Concatenate strings with `+`
- Check if `str` is empty with `str.empty()`
- Works out of the box with I/O streams

```
1  #include <iostream>
2  #include <string>
3
4  int main() {
5    const std::string source{"Copy this!"};
6    std::string dest = source;  // copy string
7
8    std::cout << source << '\n';
9    std::cout << dest << '\n';
10   return 0;
11 }
```

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;
```

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`

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 `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;`

```
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 }
```

I/O streams

- 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 }
```

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.

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;
}
```

```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>"`

http://en.cppreference.com/w/cpp/language/main_function

Program input parameters

```
1  #include <iostream>
2  #include <string>
3  using std::cout;
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 << "argv[" << i << "]: " << argv[i] << endl;
15   }
16   return 0;
17 }
```

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
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 - #pragma - #line

Expressions

Value categories
Evaluation order and sequencing
Constant expressions
Operators
assignment - arithmetic
increment and decrement
logical - comparison
member access and indirection
call, comma, ternary
`sizeof` - `alignof`(C++11)
`new` - `delete` - `typeid`
Operator overloading
Default comparisons(C++20)
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
Lvalue and rvalue references
Pointers - Arrays
Structured bindings(C++17)
Enumerations and enumerators
Storage duration and linkage
Language linkage
`inline` specifier
Inline assembly
`const`/`volatile`
`constexpr`(C++11)
`consteval`(C++20) - `constinit`(C++20)
`decltype`(C++11) - `auto`(C++11)
`alignas`(C++11)
`typedef` - Type alias(C++11)
Elaborated type specifiers
Attributes(C++11)
`static_assert`(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 expression(C++11)
Argument-dependent lookup
Overload resolution
Operator overloading
Address of an overload set
Coroutines (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(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

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