Introduction to C++

Outline

1. Introduction
2. Basic Syntax
3. Control Flow

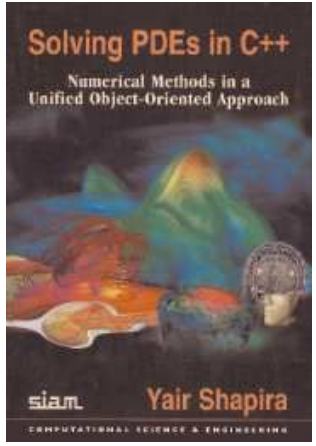
Slides/Exercices:

git clone https://github.com/MolSSI-Education/introductory-cpp

Part I
Introduction

Why Learn C++?

- Popular: lots of scientific code written in C++
- Powerful: fast, flexible, portable, scalable
- Multi-paradigm: procedural, functional, object-oriented, generic programming
- Wide support from vendors (LLVM, Microsoft, Intel, Oracle, IBM, Free Software Foundation, …)



What is C++?

- Object-oriented language developed in 1979 (Bjarne Stroustrup, Bell Labs)
- Considered to be the “successor” to C (procedural language). Most (**but not all!**) C features are a subset of C++.
- Complex language but its features are designed to be zero-cost, i.e. if your program doesn’t use a feature, it won’t slow it down.
- Very few people know all of the standard. You can use what you are comfortable with and learn as you go along.

Evolution of C++

C++98 1st C++ standard (ISO)

C++03 Addressed many core language defect reports and library defect reports ...

C++11 1st major update in 13 years. Brings many improvements (lambda's, automatic type inference, …)

C++14 Relatively minor update but brings some nifty new features (return type deduction, …)

C++17 Brings lots of stuff like folds, structured bindings, constexpr if, class template deduction, ...

C++20 To be released in 2020. Adds more new major features.

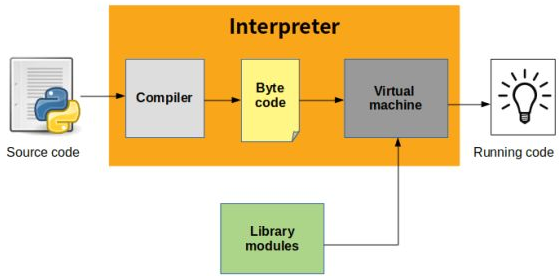
Programming Language Implementation

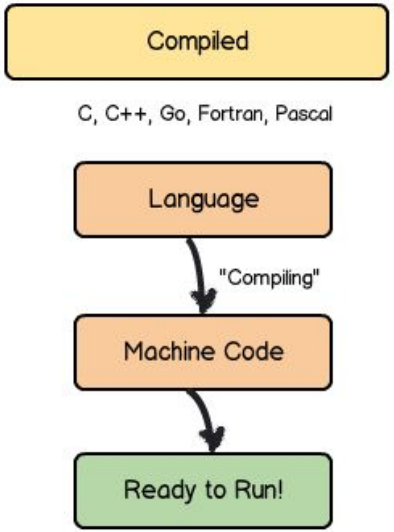
High-level programming languages are translated to machine code via one of the following methods:

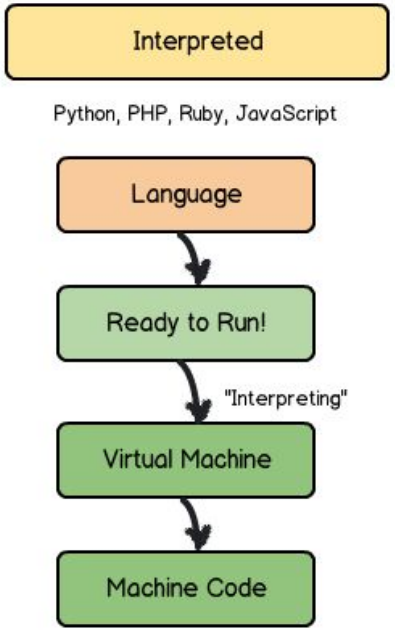
1 - Compilation

2 - Interpretation

3 - Complex combination of both







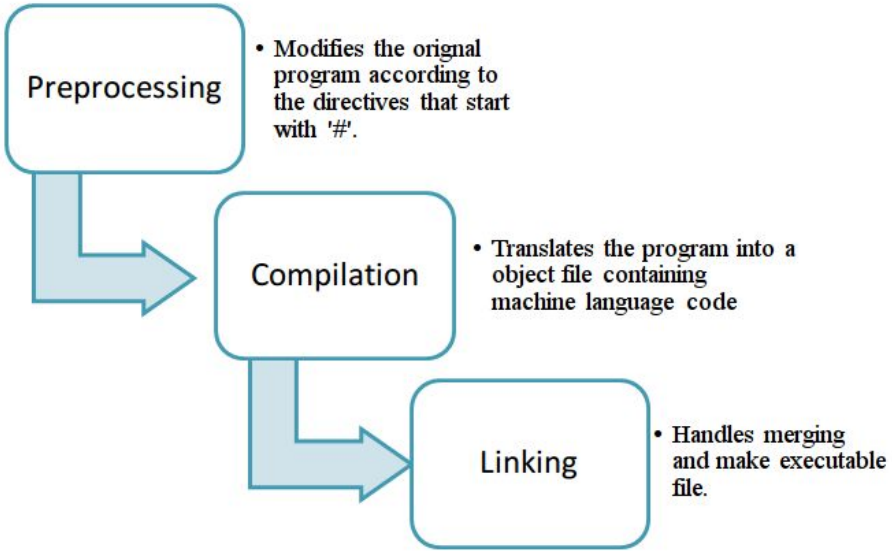
Example 1: Hello World

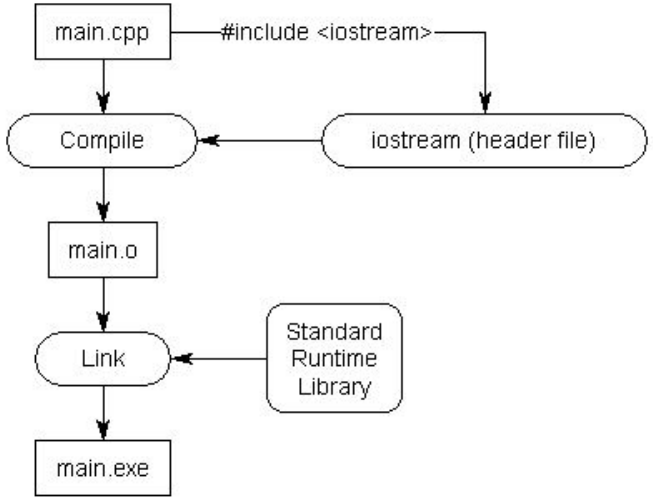
cd Part-I/Ex1

Method 1:
Compilation & linking: **g++ main.cc**
Execution: **./a.out**

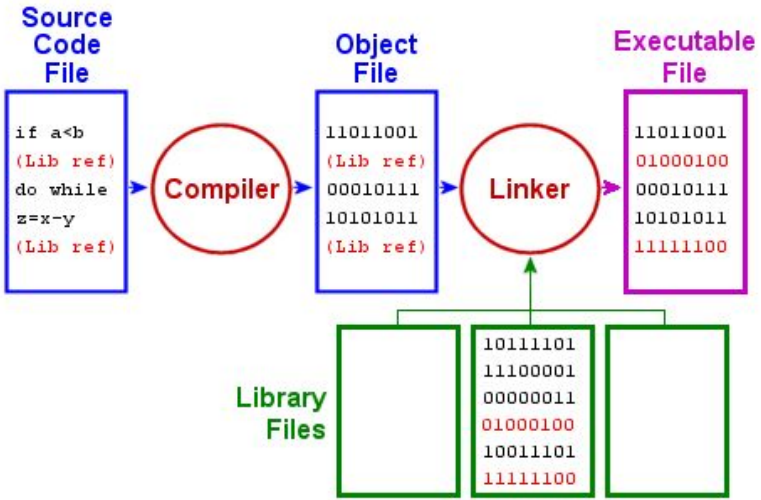
Method 2:
Compilation: **g++ main.cc -c**
Linking: **g++ main.o**
Execution: **./a.out**

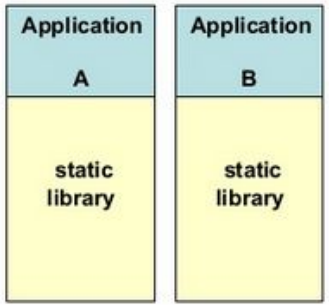
Compilation & Linking



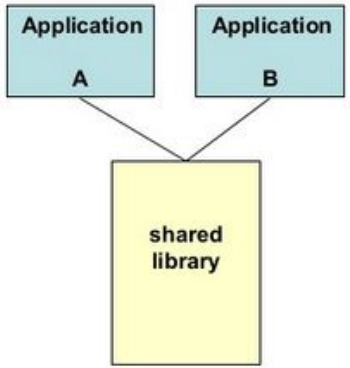


Libraries & Executables





Static library



Shared library

Example 2: Library Linking

cd ../Ex2

Static library compilation:

g++ library.cc -c

ar rv libname.a library.o

Executable compilation & linking:

g++ main.cc -lname.a -L.

Dynamic vs Static Typing

Static typing:

double variable; **variable type declared**
✔ variable = 1.0;
✘ string variable = "Ben"; **variable cannot change type**

- Easier to read
- References resolved during compile time
- Faster execution

Dynamic typing:

variable = 1.0; **variable type not declared**
✔ variable = "Ben"; **variable can change type**

- Harder to read
- References resolved during runtime
- Slower execution

Fundamental Data Types

Category	Type	Contents
Integral	char	Type char is an integral type that usually contains members of the basic execution character set.
	bool	Type bool is an integral type that can have one of the two values true or false. Its size is unspecified.
	int	Type int is an integral type that is larger than or equal to the size of type short int, and shorter than or equal to the size of type long.
Floating point	float	Type float is the smallest floating point type.
	double	Type double is a floating point type that is larger than or equal to type float, but shorter than or equal to the size of type long double.

Example 3: BMI Calculator

cd ../Ex3

1 file to edit:

- **bmi.cc**: entry point (“main” function)

Compilation:

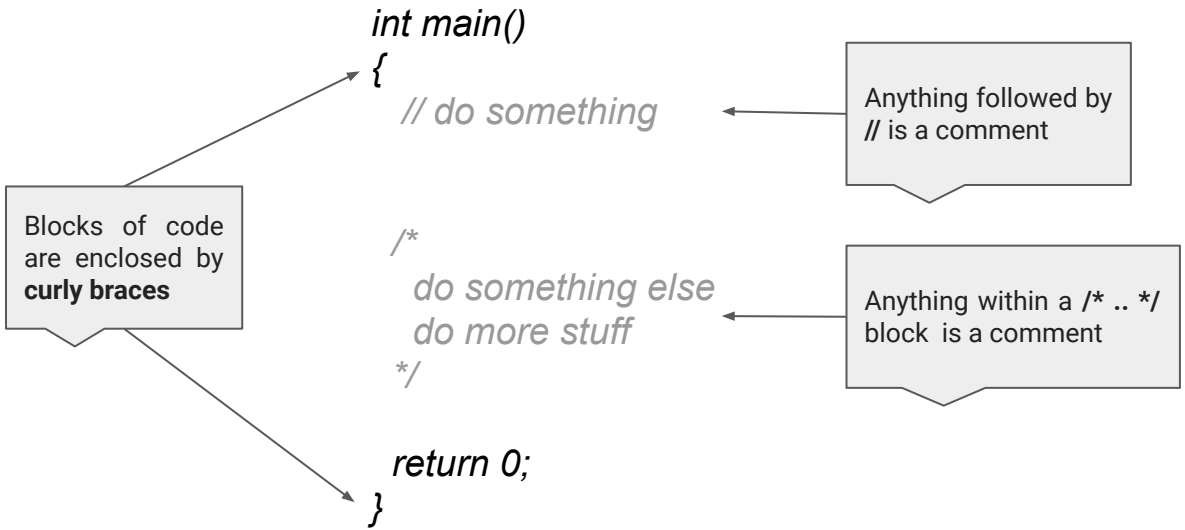
g++ main.cc -o bmi.out

Run:

./bmi.out

Part II
Basic Syntax

Program Structure



C++ vs Python: Syntax

C++ **for** loop:

```
for (int i = 0; i < 10; i++)
{
    // do something
    // do something else
}
```

Blocks of code are enclosed by **curly braces**

Python **for** loop:

```
for i in range(10):
    # do something
    # do something else
```

Blocks of code are denoted by **whitespace**

C++ vs Python: Scope

C++ function

```
#include <iostream>

void foo(bool isPositive)
{

    int variable;

    if(isPositive)
        variable = 1;
    else
        variable = -1;

    std::cout << "variable = " <<  variable;
}
```

An expression followed by a semicolon is a **statement**

Variables must be declared before they can be used

Python function

```
def foo(isPositive):

    # no need to declare variable

    if(isPositive):
        variable = 1
    else:
        variable = -1

    print("variable =", variable)
```

Exercise 1: Scope

cd ../../Part-II/Ex1

Fix the logical bug to get result = 448.

1 file to edit:

- **main.cc**: entry point (“main” function)

Compilation:

```
g++ main.cc print_int.cc -o print_int.a
```

Run:

```
./print_int.a
```

Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

An identifier cannot be a reserved keyword.

Acceptable identifiers:

- John
- john
- _temp
- Pi22
- foo_123

Unacceptable identifiers:

- @John
- $john
- -temp
- 22Pi
- **double**

Some Reserved Words

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

C++ Files

Function “foo” **declaration**

```
// my_file.h
#ifndef MY_FILE_H // include guard
#define MY_FILE_H

void foo(int);

#endif
```

Function “foo” **definition**

```
// my_file.cc
#include “my_file.h”
#include <iostream>

void foo(int integer)
{
    std::cout << integer << std::endl;
}
```

Exercise 2: Declaration vs Definition

cd ../Ex2

3 files to edit:

- **main.cc**: entry point (“main” function)
- **print_int.cc**: user-defined function definition
- **print_int.h**: user-defined function declaration

Compilation:

```
g++ main.cc print_int.cc -o print_int.a
```

Run:

```
./print_int.a
```

<<: insertion operator
e.g. cout << “hello world!”;

>>: extraction operator
e.g. cin >> input;

Namespaces

Unnamed namespaces are typically used to shield global data

```
namespace
{
        // members such as functions &
        // classes go here
}
```

```
namespace nameSpaceA
{
        // memberA
}
```

```
namespace nameSpaceB
{
        namespace subNameSpace
        {
                // memberB
        }
}
```

Members accessible via the ***scope operator ::***

e.g.
nameSpace::memberA
nameSpaceB::subNameSpace::memberB

This provides accessibility similar to that of Python modules

e.g.
module.memberA
module.submodule.memberB

C++ Files

Function “foo” **declaration**

```
// my_file.h
#ifndef MY_FILE_H // include guard
#define MY_FILE_H

namespace name
{
    void foo(int);
}

#endif
```

Function “foo” **definition**

```
// my_file.cc
#include “my_file.h”
#include <iostream>

void name::foo(int integer)
{
    std::cout << integer << std::endl;
}
```

Exercise 3: Namespaces

cd ../Ex3

1 file to edit:

- **main.cc**: entry point (“main” function)
- **print_int.cc**: define user-defined functions

Compilation:

```
g++ main.cc print_int.cc -o print_int.a
```

Run:

```
./print_int.a
```

```
namespace some_name
{
    Members
}
```

Part III
Control Flow

If Statements

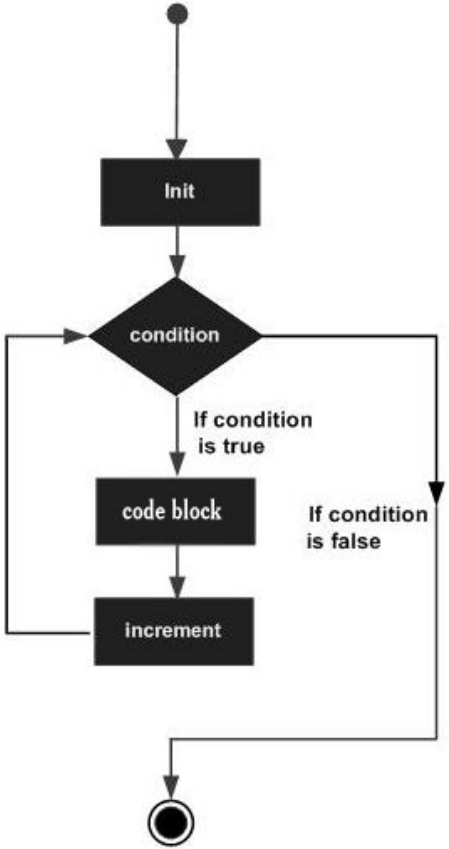
```
if (expression)
        // statement
else
        // statement
```

```
if (expression)
{
        // statement1
        // statement2
        // ...
}
else
{
        // statement1
        // statement2
        // ...
}
```

For Loop

```
for (initialization; condition; increment)
        // statement
```

```
for (initialization; condition; increment)
{
        // statement1
        // statement2
        // ...
}
```



For Loop: Example

```cpp
#include <iostream>

int main ()
{
   // for loop execution
   for( int a = 10; a < 20; a = a + 1 )
      std::cout << "value of a: " << a << std::endl;

   return 0;
}
```

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Range-based For Loop

```cpp
#include <iostream>

int main ()
{
   int array[] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};

   // Range-based for loop execution
   for( int a: array )
      std::cout << "value of a: " << a << std::endl;

   return 0;
}
```

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Exercise 1: Recursive Factorial

cd ../../Part-III/Ex1

1 file to edit:

- **main.cc**: entry point (“main” function)

Compilation:

```
g++ main.cc -o factorial.a
```

Run:

```
./factorial.a
```

Factorials:

N! = N(N-1)(N-2)...(1)

0! = 1

No custom header files!

Exercise 2: Iterative Factorial

cd ../Ex2

1 file to edit:

- **main.cc**: entry point (“main” function)

Compilation:

```
g++ main.cc -o factorial.a
```

Run:

```
./factorial.a
```

Factorials:

N! = N(N-1)(N-2)...(1)

0! = 1

No custom header files!

Extras

Documentation: Doxygen

```
/*  @brief C++ implementation of Fortran BLAS daxypy
    Computes the equation ys[i] <- xs[i] * alpha + beta

    @note Function with C-linkage.

    @param[in]      n      Array size. Size of xs and ys
    @param[in]      xs     Input  array xs
    @param[in, out] ys     Output array ys
    @param[in]      alpha  Linear coefficient
    @return         Void
*/

auto daxpy(size_t n, double const* xs, double* ys,
           double alpha, double beta) ->  void;
```

Function Documentation

◆ daxpy()

```
auto daxpy ( size_t          n,
             double const *  xs,
             double *        ys,
             double          alpha,
             double          beta
           )                 -> void
```

C++ implementation of Fotran BLAS daxypy
Computes the equation ys[i] <- xs[i] * alpha + beta.

Note

Function with C-linkage.

Parameters

[in]	**n**	Array size. Size of xs and ys
[in]	**xs**	Input array xs
[in, out]	**ys**	Output array ys
[in]	**alpha**	Linear coefficient Void

Testing Frameworks

CMake has support for adding tests to a project:

```
enable_testing()
```

This adds another build target, which is `test` for Makefile generators, or `RUN_TESTS` for integrated development environments (like Visual Studio).

From that point on, you can use the `add_test` command to add tests to the project:

```
add_test(testname Exename arg1 arg2 ...)
```

Or, in its longer form:

```
add_test(NAME <name> [CONFIGURATIONS [Debug|Release|...]]
         [WORKING_DIRECTORY dir]
         COMMAND <command> [arg1 [arg2 ...]])
```

Once you have built the project, you can execute all tests via

```
make test
```

with Makefile generators, or by rebuilding the `RUN_TESTS` target in your IDE. Internally this runs CTest to actually perform the testing; you could just as well execute

```
ctest
```

in the binary directory of your build.

In some projects you will want to set `*_POSTFIX` properties on executables that will be executed for testing, e.g. to make executables compiled with debug information distinguishable ("-debug"). Note that the shorthand version of `add_test` does *not* automatically append these postfixes to the commands it calls for the test target, i.e. your test will want to call "" but the executable is "-debug", resulting in an error message. Use the long version of the `add_test()` in this case, which adds the appropriate `_POSTFIX` to the command name.

For more information, check the CMake Documentation or run:

```
cmake --help-command enable_testing
cmake --help-command add_test
cmake --help-property "<CONFIG>_POSTFIX"
cmake --help-command set_property
```

ctest CMake

```
1  TEST_CASE("Test positives", "[classic]")
2  {
3     SECTION("Test all up to 10") {
4        REQUIRE(fizzbuzz(1) == "1");
5        REQUIRE(fizzbuzz(2) == "2");
6        REQUIRE(fizzbuzz(3) == "fizz");
7        REQUIRE(fizzbuzz(4) == "4");
8        REQUIRE(fizzbuzz(5) == "buzz");
9        REQUIRE(fizzbuzz(6) == "fizz");
10       REQUIRE(fizzbuzz(7) == "7");
11       REQUIRE(fizzbuzz(8) == "8");
12       REQUIRE(fizzbuzz(9) == "fizz");
13       REQUIRE(fizzbuzz(10) == "buzz");
14    }
15
16    SECTION("Test all multiples of 3 only up to 100") {
17       for (int i = 3; i <= 100; i+=3) {
18          if (i % 5) REQUIRE(fizzbuzz(i) == "fizz");
19       }
20    }
21
22    SECTION("Test all multiples of 5 only up to 100") {
23       for (int i = 5; i <= 100; i += 5) {
24          if (i % 3) REQUIRE(fizzbuzz(i) == "buzz");
25       }
26    }
27
28    SECTION("Test all multiples of 3 and 5 up to 100") {
29       for (int i = 15; i <= 100; i += 15) {
30          REQUIRE(fizzbuzz(i) == "fizzbuzz");
31       }
32    }
33 }
```

Catch 2

References

BOOKS

A Tour of C++ (2nd Edition) (C++ In-Depth Series)

Bjarne Stroustrup

Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14

Scott Meyers

C++ Primer Plus (6th Edition) (Developer's Library)

Stephen Prata

C++ Coding Standards: 101 Rules, Guidelines, and Best Practices

Herb Sutter

WEBSITES

https://isocpp.org/get-started

http://www.learncpp.com

http://www.cplusplus.com/doc/tutorial

http://cppreference.com
The definitive reference on the C++ standard library

Questions?

Email me:

andrewabimansour@vt.edu

Andrew Abi-Mansour