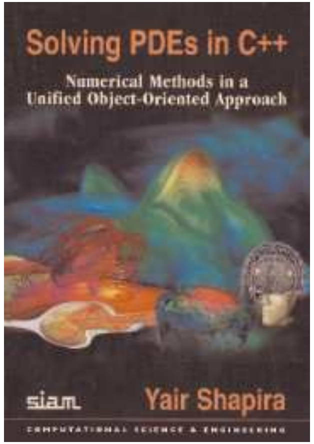
Introduction to C++

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, …)


Solving PDEs in C++
Numerical Methods in a
Unified Object-Oriented Approach
siam
Yair Shapira
COMPUTATIONAL SCIENCE & ENGINEERING

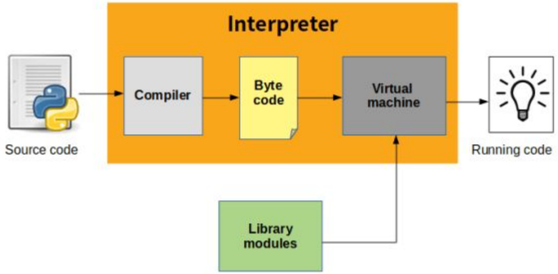
Programming Language Implementation

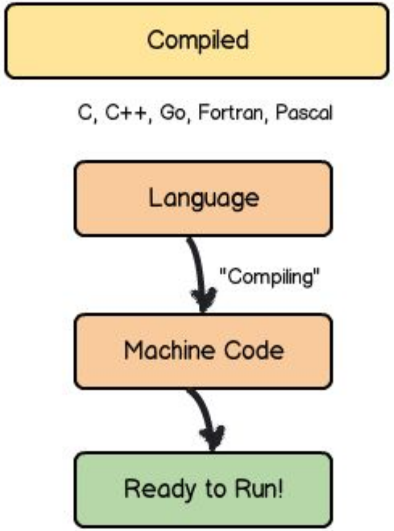
High-level programming languages are translated to machine code via one of the following methods:

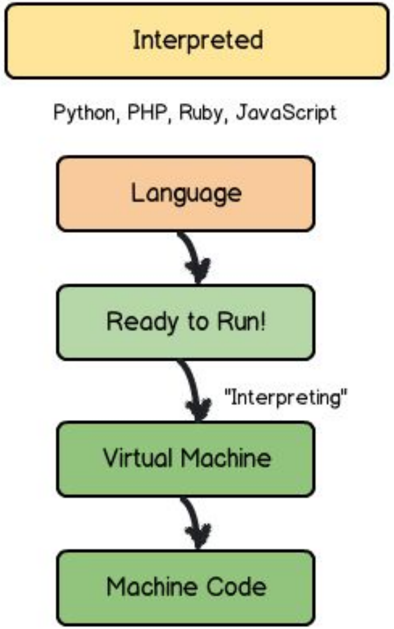
1 - Compilation

2 - Interpretation

3 - Complex combination of both







Compilation vs Interpretation

		Compilation	Interpretation
1	Input	Entire program	Single instruction
2	Portability	Specific to machine architecture	Cross-platform
3	Speed	Faster execution	Slower execution
4	Workload	Compiled once	Interpreted every run
5	Errors	Returned during compile time	Returned during run time
6	Code	Private	Public

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 ...

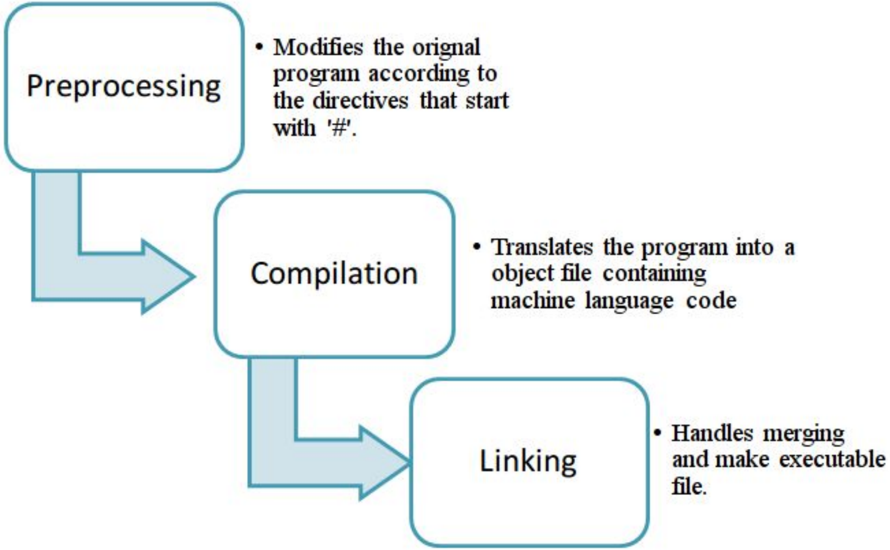
C++11 1st major update in 13 years. Brings many improvements (lambda’s, automatic type inference, …)

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 Feb 2020. Adds more new major features.

Compilation & Linking



Preprocessing

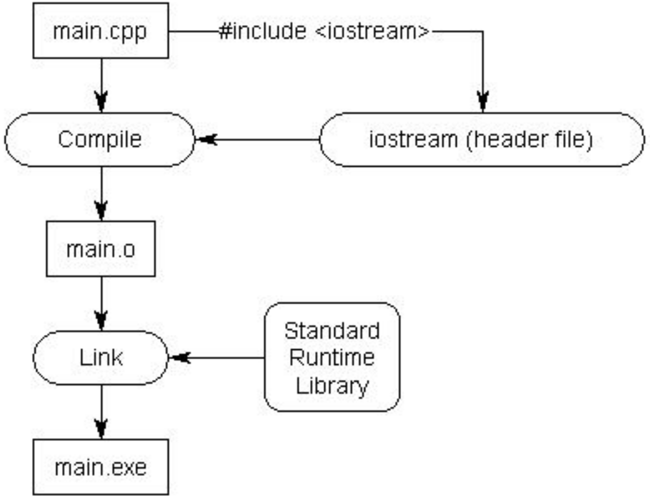
- Modifies the orignal program according to the directives that start with '#'.

Compilation

- Translates the program into a object file containing machine language code

Linking

- Handles merging and make executable file.



Dynamic vs Static Typing

Static typing:

double variable; **variable type declared**
✓ variable = 1.0;
✗ string variable = "Ben"; **variable cannot change type**

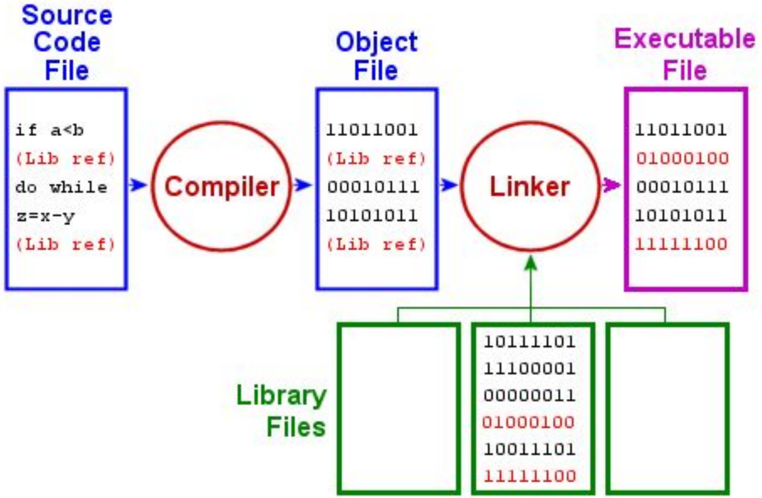
- Easier to read
- References resolved during compile time
- Faster execution

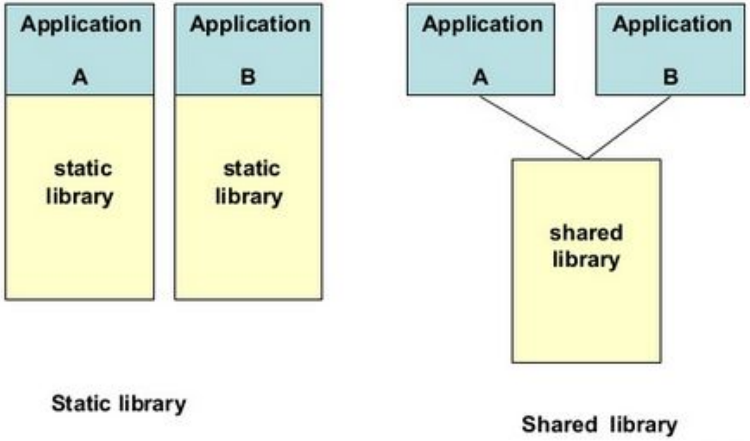
Dynamic typing:

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

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

Libraries & Executables





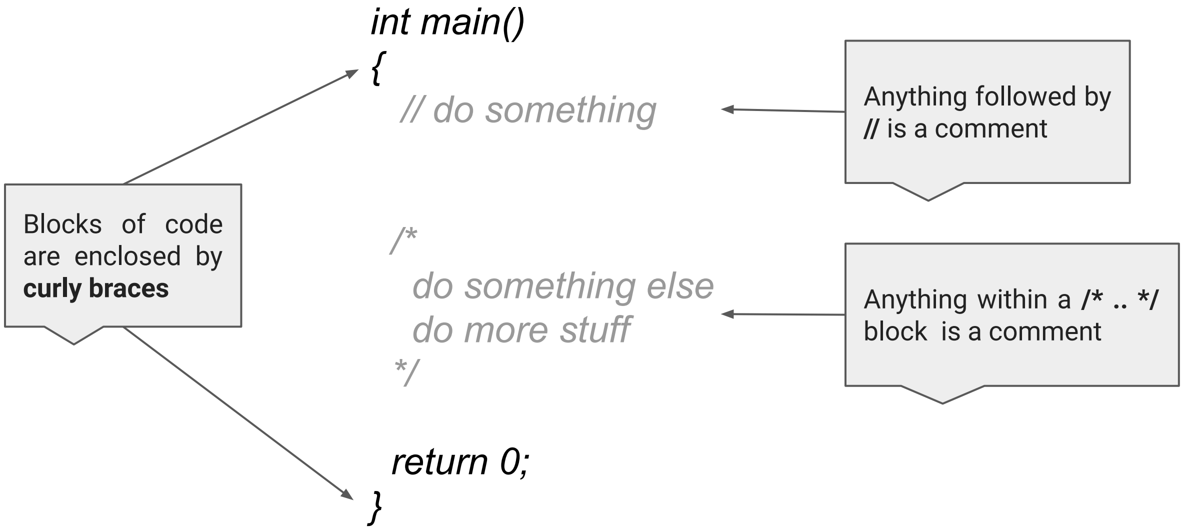
Example 1

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

cd introductory-cpp

cd Part-I/Ex1

clang++ main.cc -o hello_world.a

./hello_world.a
```

Part II

Basic Syntax

Program Structure



```
int main()
{
  // do something


 /*
   do something else
   do more stuff
 */

  return 0;
}
```

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

Anything followed by **//** is a comment

Anything within a **/* .. */** block is a comment

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)
```

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

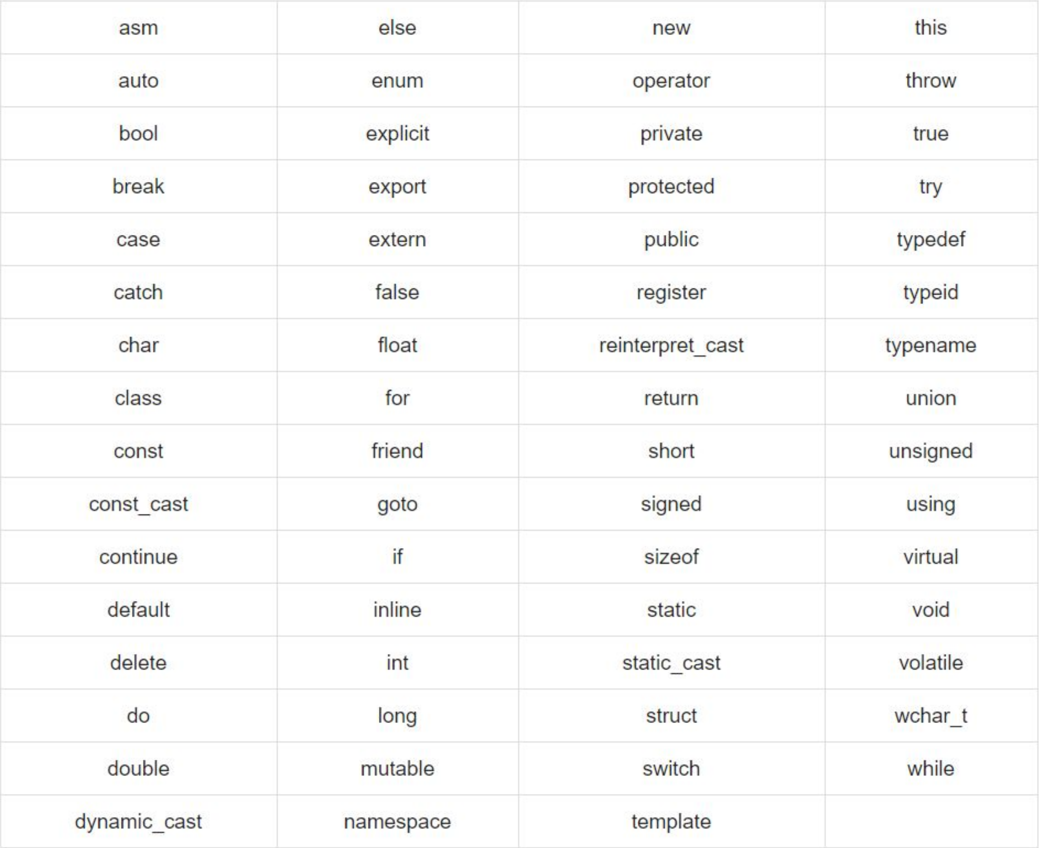
✓

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	

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 1: Declaration vs Definition

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

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:

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

Run:

```
  ./print_int.a
```

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

Exercise 2: Extraction

cd ../Ex2

1 file to edit:

- ❑ **main.cc**: entry point ("main" function)

Compilation:

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

Run:

```
./print_int.a
```

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

Exercise 3: Namespaces

cd ../Ex3

<u>1 file to edit:</u>

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

<u>Compilation:</u>

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

<u>Run:</u>

```
./print_int.a
```

```
namespace some_name
{
    Members
}
```

Part III

Control Flow

If Statements

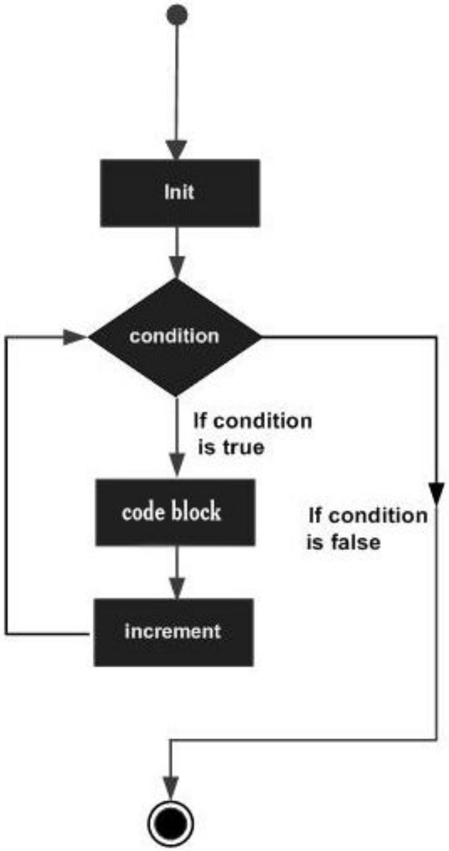
```
if (expression)
        // statement
else
        // statement
```

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

For Loop

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

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



For Loop: Example

```
#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

```
#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:

```
clang++ 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:

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

Run:

```
./factorial.a
```

Factorials:

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

$0! = 1$

No custom header files!

Part IV
Data Types

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.

Derived Data Types: Functions

```
type myFunction() {
  // code to be executed
  return type;
}
```

```
double myFunction(double a, double b = 2)
{
  return a*b;
}
```

```
void myFunction() {
  // code to be executed
  // returns nothing
}
```

```
double myFunction()
{
  double variable;
  // code to be executed

  return variable;
}
```

Polymorphism: Function Overloading

2 or more functions having the same name but different implementation

```
float area(float base, float height)
{
        // computes area of a triangle
        return base * height / 2.0;
}
```

```
float area(float radius)
{
        // computes area of a circle
        return 3.14 * radius * radius;
}
```

Operators (such as +, -, *, etc.) can be overloaded in C++ as well.

Exercise 1: Functions

```
cd ../../Part-IV/Ex1
```

<u>1 file to edit:</u>

- ❑ **equation.h**: function declaration

<u>Compilation:</u>

```
clang++ main.cc equation.cc -o function.a
```

<u>Run:</u>

```
./function.a
```

Default arguments are included in function declaration.

Exercise 2: Function Overloading

cd ../Ex2

1 file to create:

❏ **print.cc**: function definition for the 2 “print” functions declared in **print.h**

Compilation:

```
clang++ main.cc print.cc -o print.a
```

Run:

```
./print.a
```

Derived Data Types: C-style Arrays

A C-style array is a collection of items stored at contiguous memory locations and elements can be accessed randomly using indices (0,1,..)

index	0	1	...	N-1
content	item1	item2	...	itemN

e.g. int numbers[ ] = {10, 20, 30, 40};

char myword[ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword[ ] = "Hello";

Array elements are accessed via the **array subscript operator [ ]** e.g.

numbers[0] -> 10

myword[1] -> ‘e’

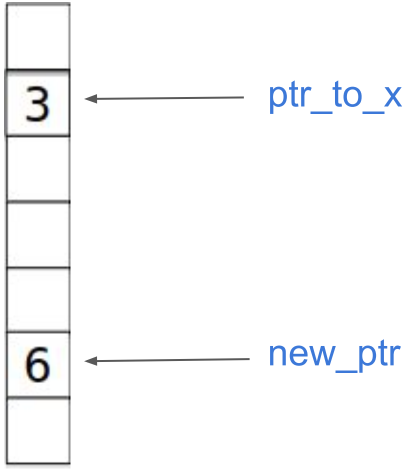
Derived Data Types: Pointers

- For every type, there is also a special type called a pointer, e.g. int* is a pointer to an int.
  - You get pointers by taking the address of objects with &.
  - You access the object being pointed to with the dereference operator *.

- A pointer stores the location of an object in memory. It is not an integer, but you can do some math with it:

```
int x = 3;
int* ptr_to_x = &x;

int* new_ptr = ptr_to_x + 4;
*new_ptr = 6;
```

Dereference operator



Pointers & Memory Allocation

Pointers can be used to allocate memory dynamically with the **new** operator:

```
double* x = new double();
*x = 4.0;
// alternatively: double* x = new double(4.0);
```

Remember to release the memory with the **delete** operator (C++ doesn't have garbage collection): delete x;

A contiguous block of memory can be allocated to create a dynamic array:

```
double* x = new double[size];

for (int i = 0; i < size; i++)
{
        // each element in x can be accessed via *(x + i) or x[i]
        // do something
}

delete[] x;
```

Exercise 3: Arrays & Pointers

```
cd ../Ex3
```

<u>1 file to create:</u>

- ❏ **dot_product.cc**: function definitions for “dot_product_ptr” and “dot_product_arr” declared in **main.cc**

<u>Compilation:</u>

```
clang++ main.cc dot_product.cc -o dot_product.a
```

<u>Run:</u>

```
./dot_product.a
```

type *ptr const is a constant pointer

Exercise 4: Ptrs & Memory Allocation

```
cd ../Ex4
```

1 file to edit:

- ❏ **main.cc**: complete function definition for *main*() and *print*()

Compilation:

```
clang++ main.cc -o main.a
```

Run:

```
./main.a
```

new operator is used to allocate memory

delete operator is used to free memory

Exercise 5: Ptrs & Functions

cd ../Ex5

<u>1 file to edit:</u>

❏ **main.cc**: complete function definition for *main*() and *print*()

<u>Compilation:</u>

```
clang++ main.cc -o main.a
```

<u>Run:</u>

```
./main.a
```

new operator is used to allocate memory

delete operator is used to free memory

Exercise 6: Vectors

cd ../Ex6

1 file to edit:

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

Compilation:

```
clang++ main.cc -o vector.a
```

Run:

```
./vector.a
```

std::vector is a sequence container that encapsulates dynamic size arrays

Exercise 7: Tuples

```
cd ../Ex7
```

<u>1 file to edit:</u>

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

<u>Compilation:</u>

```
clang++ main.cc -o tuples.a -std=c++17
```

<u>Run:</u>

```
./tuples.a
```

std::tuple offer fixed-size collection of heterogeneous values.

std::apply invokes a callable object with a tuple of arguments.

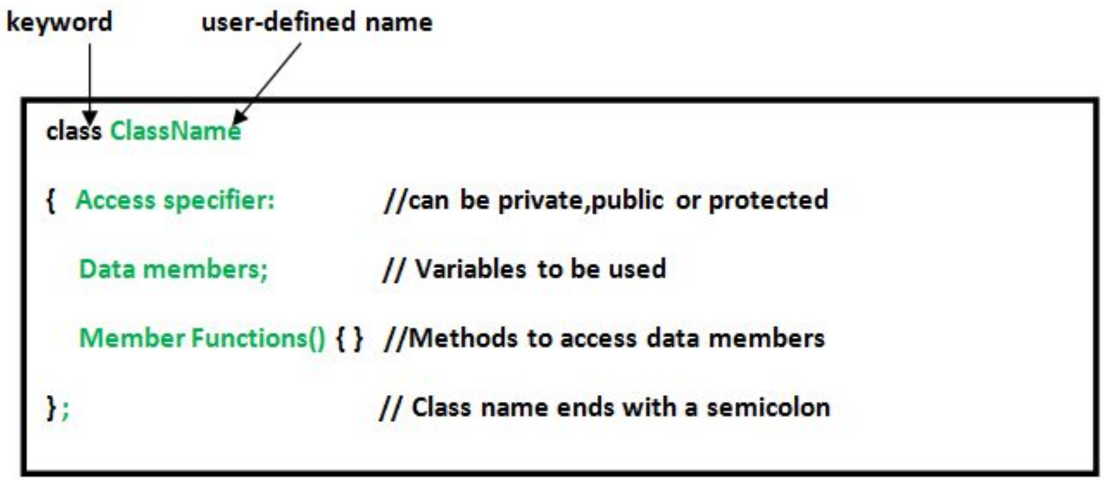
C++17

Part V

Classes & Objects

Classes

- Building blocks of Object-Oriented programming
- User-defined data type, which holds its own data members and member functions



```
class ClassName

{  Access specifier:          //can be private,public or protected

   Data members;              // Variables to be used

   Member Functions() { }     //Methods to access data members

};                            // Class name ends with a semicolon
```

Access Specifiers & Objects

- Modify the access rights for class members
- 3 types: **private**, **public** or **protected**
- By default, all class members are **private**

```
class Person
{
 private:
   float age, height, weight;
   double wage;
   std::string name;
   bool isHealthy;

 public:
   void set_height(float);
   void set_age(float);
};
```

Instantiation →

harry is an object of type **Person**

```
auto harry = Person(...);

harry.set_height(5.8);   ✓

harry.height = 5.8;      ✗
```

Constructors & Destructors

- A constructor is a special member function of a class that is invoked whenever we create new objects of that class

- A destructor is a special member function of a class that is invoked whenever an object of its class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class

```
class Foo
{
   public:
      void setProp(double prop);
      Foo();  // This is the constructor
      ~Foo();  // This is the destructor
   private:
      double prop;
 };
```

```
Foo::Foo(void)
{
   std::cout << "Object is being created";
}

Foo::~Foo(void)
{
   std::cout << "Object is being deleted";
 }
```

Exercise 1: Class demo

```
cd ../../PartV/Ex1
```

2 files to create:

- ❏ **element.h:** defines an “Element” class
- ❏ **element.cc** : defines methods of “Element”

Compilation:

```
clang++ main.cc element.cc -o element.a
```

Run:

```
./element.a
```

RAII: programming idiom (resource acquisition is initialization)

Exercise 2: Classes & Pointers

```
cd ../Ex2
```

<u>2 files to create:</u>

- ❏ **element.h:** defines an “Element” class
- ❏ **element.cc** : defines methods of “Element”

<u>Compilation:</u>

```
clang++ main.cc element.cc -o element.a
```

<u>Run:</u>

```
./element.a
```

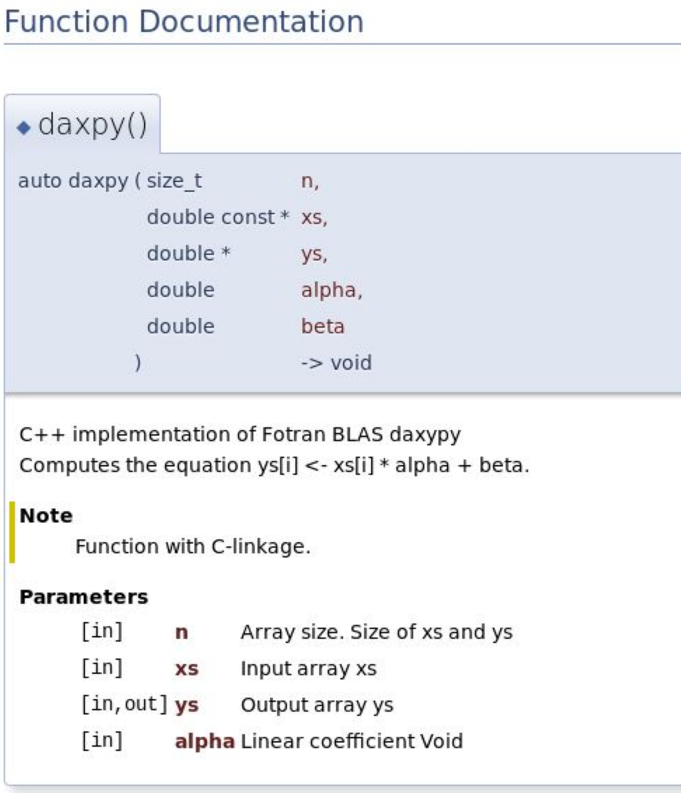
type *ptr const is a constant pointer

Part VI

References

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

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

Thank You