C++ Primer
What is C++?

C++ is a general-purpose programming language with a bias towards systems programming that:

• Is a better C
• Supports data abstraction (e.g., classes)
• Supports object-oriented programming (e.g., inheritance)
• Supports generic programming (e.g., reusable generic containers and algorithms)
• Supports functional programming (e.g., template metaprogramming, lambda functions)

(see the C++ Super-FAQ at https://isocpp.org/faq)
History of C++

• Extension of C
• Early 1980s: Bjarne Stroustrup
• Supports OOP (Object Oriented Programming)
  • Objects are reusable software components (attempt to model items in the real world)
  • Object-oriented programs are easier to understand, correct, and modify
• C++ is a hybrid language
  • C-style programming
  • OOP-style

• Standardized
  • Current standard: C++14
  • Working draft: C++17
Object-oriented programming (OOP)

• OOP is a methodology for organizing data and functions
• In OOP, functions (called methods) are attached/associated with the data (objects) (whereas in procedural-based programming, functions act on data)
• In OOP, functions can only be invoked through an object
• Note: C++ allows both object-oriented and procedural programming

• OOP provides a clean interface between programmer and user
• OOP facilitates code reuse through composition/aggregation, inheritance, and polymorphism
  • Aggregation: a whole is made out of parts (but does not own the parts)
  • Composition: a whole is made out of parts (and owns the parts)
  • Inheritance: new classes inherit some of the properties and behavior of existing classes
  • Polymorphism: code/operation behaves differently in different contexts
Roles in OOP

• Design (architect)
  Think how to solve a problem using objects (language agnostic)

• Implement
  Code C++ classes, functions, etc. (requires detailed understanding of design)

• Use
  Make use of C++ classes in user code (requires high-level understanding of design)
C and C++ concepts
And no, I'm not a walking C++ dictionary. I do not keep every technical detail in my head at all times. If I did that, I would be a much poorer programmer. I do keep the main points straight in my head most of the time, and I do know where to find the details when I need them.

Bjarne Stroustrup
Scope

• A scope is a region of program text
  • Global scope (outside any language construct)
  • Class scope (within a class)
  • Local scope (between { ... } braces)
  • Statement scope (e.g. in a for-statement)

• A name in a scope can be seen from within its scope and within scopes nested within that scope
  • Only after the declaration of the name ("can’t look ahead" rule)
  • Class members can be used within the class before they are declared

• A scope keeps “things” local
  • Prevents my variables, functions, etc., from interfering with yours
  • Remember: real programs have many thousands of entities
  • Locality is good!
    • Keep names as local as possible
Scope

```cpp
// no r, i, or v here
class My_vector {
    vector<int> v; // v is in class scope

public:
    int largest() // largest is in class scope
    {
        int r = 0; // r is local
        for (int i = 0; i < v.size(); ++i) // i is in statement scope
            r = max(r, abs(v[i]));
        // no i here
        return r;
    }
    // no r here
}; // no v here
```

www.stroustrup.com/Programming
Namespaces

- Address the problem of naming conflicts between different parts of the code.
- Namespaces define the context (scope) in which names (types, functions, variables) are defined:

  ```
  // namespace.h
  namespace myscope {
    void foo();
  }
  ``

  ```
  // namespace.cpp
  include <iostream>
  namespace myscope {
    void foo() {
      std::cout << "calling my foo()" << std::endl;
    }
  }
  ```

- Calling foo() from the mycode namespace:

  ```
  myscope::foo();
  ```

- Multiple namespace blocks with the same name are allowed.
- Nested namespaces are allowed (e.g., `chrono::vehicle::ChVehicle`)
Namespaces

- using-directive: avoid explicitly prepending the namespace for all declared names:

```cpp
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
    foo(); // equivalent to calling myscope::foo()
}
```

- using-declaration: avoid explicitly prepending the namespace for a single name:

```cpp
#include <iostream>
using std::cout;
using std::endl;
int main(int argc, char* argv[]) {
    cout << “Hello World!” << endl;
}
```

- Do not put ‘using namespace’ directives in header files!
  - It forces all includes of that header to use that namespace, potentially resulting in ambiguities.
Constants

• C-style constants (using macros)

  \#define PI 3.1415926

• C++ style constants (using const)

  const double PI = 3.1415926;

• New style: provides type and scope
Pointers

- A pointer is an object whose value is the address in memory where another object is stored.
- A pointer to an object of type T is denoted by T*.
- A null pointer does not refer to a valid address location; null pointer value provided by the keyword nullptr.
- Accessing the object to which a pointer refers is called dereferencing.
  - Dereferencing a pointer is done with the indirection operator *.
  - If p is a pointer, then *p is the object to which the pointer refers.
- If x is an object of type T, then &x is the address of x (a pointer of type T*).

```c
int a;
int* p = nullptr; // p is a pointer to an int
int* p1 = &a;    // p1 is a pointer to an int (and points to the address of a)
```
References

• References are aliases (for an already existing object):

```c
int var;
int& ref = var;
```

• From here on, ref is an alias for var. You cannot make ref an alias for another variable.

• References are **not** pointers.

• Note:
  • Above are so-called lvalue references
  • There is also the concept of rvalue references (used in the context of move constructors and move assignment operators)
Parameter passing by reference

- Avoids (potentially expensive) copying

```c
void swap(int& x, int& y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

- Const reference parameters

```c
int compare(const MyType& x, const MyType& y);
```

- Guarantee that a function does not modify parameters passed as const references
- Compiler-time check
Pointers vs. references

• Both can be used to refer to some other entity (e.g., an object or a function)

• Two key differences:
  • References must refer to something; pointers can have null value (nullptr)
  • References cannot be rebound; pointers can be modified to point to some other entity

• References have cleaner syntax; to be used, pointers must be dereferenced

• Pointers typically require memory management (new/delete)

• Prefer using references instead of pointers, unless:
  • You need to refer to “nothing” (nullptr)
  • You need to change what you refer to
Classes and objects
Classes

• A class is a **user-defined type**

• A class specifies:
  • How objects of that type are represented (through its member variables)
  • What operations can be performed on such objects (through its member functions)

• A class can have zero or more **members**:
  • **Data** members (define the representation of objects of the class)
  • **Function** members (define operations on objects of the class)
  • **Type** members (define types associated with the class)

• The **interface** is the part of a class accessible to users

• The **implementation** of a class is the internal part of a class (accessible to users only indirectly, through the class interface)
Class access specifiers

• Control the access level that users have to the class members

• There are three levels of access:
  • **public**: these members can be accessed by any code
  • **protected**: these members can be accessed by derived classes (related to inheritance)
  • **private**: these members can only be accessed by other members of the class (also by friends of the class)

• The public members constitute the class interface

• The private and protected members constitute the class implementation
Objects

• Classes are “first-class citizens”! They have the **same standing** as all built-in types
• Objects are variables of a certain class type (instances of that class)
• Objects can be passed to functions
• The return value of a function can be an object
• You can implement type-conversion operations to automatically convert objects from one class to another
• All rules for resolving overloaded functions also apply to functions with object arguments
Example of a class

```cpp
#include "Date.h"

class Date {
public:
    Date();
    Date(int year, int month, int day);
    void SetDate(int year, int month, int day);
    void PrintDate() const;
    int GetYear() const {return m_year;}
    int GetMonth() const {return m_month;}
    int GetDay() const {return m_day;}
private:
    int m_year;
    int m_month;
    int m_day;
};
#endif

// Default constructor
Date::Date() : m_year(2016), m_month(1), m_day(1) {
}

// Constructor
Date::Date(int year, int month, int day)
    : m_year(year), m_month(month), m_day(day)
{
}

// Member functions
void Date::SetDate(int year, int month, int day) {
    m_year = year;
    m_month = month;
    m_day = day;
}
void Date::PrintDate() const {
    // ...
}
```

Date.h (header file)

Date.cpp (implementation file)
The Date class

• **Date** is a class. It is a new **data type**
• Entities such as **today** or **election_day** are **instances** of the **Date** class and each one represents an **object** of type **Date**
• Note: class ≠ object
• **m_year, m_month, m_day** are member variables (data members)
• **SetDate()** is a member function (method)

```cpp
#include "Date.h"

int main(int argc, char* argv[]) {
    Date today(2016, 11, 14);
    today.Print();
}
Constructors and destructors

• A **constructor** is a member function which initializes the class.

• A constructor has
  • the same name as the class itself
  • no return type

• A class can have more than one constructor, as long as the argument lists differ.

• A constructor is called automatically whenever a new instance of a class is created.

• You must supply the arguments to the constructor when a new instance is created.

• If no constructor is specified, the compiler generates a default constructor for you.
  • may not be what you want!
Constructors and destructors

• A destructor is a member function which deletes an object.

• A destructor function is called automatically when the object goes out of scope:
  1. the function ends
  2. the program ends
  3. a block containing temporary variables ends
  4. a delete operator is called

• A destructor has:
  • the same name as the class but is preceded by a tilde (~)
  • no arguments and no return value
Constructors and destructors

/// Geometric object representing a piecewise cubic
/// Bezier curve in 3D.
class ChApi ChLineBezier : public ChLine {

    public:
    ChLineBezier() : m_own_data(false), m_path(NULL) {} 
    ChLineBezier(ChBezierCurve* path);
    ChLineBezier(const std::string& filename);
    ChLineBezier(const ChLineBezier& source);
    ChLineBezier(const ChLineBezier* source);
    ~ChLineBezier();
    
    // ...
    // ...

    private:
    bool m_own_data; ///< owns the data?
    ChBezierCurve* m_path; ///< pointer to Bezier curve
};

ChLineBezier::ChLineBezier(ChBezierCurve* path)
: m_own_data(false), m_path(path) {
    complexityU = static_cast<int>(m_path->getNumPoints());
}

ChLineBezier::ChLineBezier(const std::string& filename) {
    m_path = ChBezierCurve::read(filename);
    m_own_data = true;
    complexityU = static_cast<int>(m_path->getNumPoints());
}

ChLineBezier::ChLineBezier(const ChLineBezier& source) : ChLine(source) {
    m_path = source.m_path;
    m_own_data = false;
    complexityU = source.complexityU;
}

ChLineBezier::ChLineBezier(const ChLineBezier* source) : ChLine(*source) {
    m_path = source->m_path;
    m_own_data = false;
    complexityU = source->complexityU;
}

ChLineBezier::~ChLineBezier() {
    if (m_own_data)
        delete m_path;
}
Smart pointers
C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, you blow away your whole leg!

Bjarne Stroustrup
Dynamic memory in C++

• Dynamic memory allocated using operator `new`
  • `new` is followed by a data type specifier and, if needed, the number of elements (within `[]`)
  • `new` returns a pointer to the beginning of the new block of memory allocated
  • `new` can use any variable value for size (since memory is assigned at run time)

• Dynamic memory no longer needed can be freed with the operator `delete`
  • The value passed to `delete` must be a pointer previously allocated with `new` or `nullptr`
  • `delete` releases memory of a single element allocated using `new`
  • `delete[]` releases memory allocated for arrays of elements using `new` and size in brackets

```c++
ChBody* body = new ChBody(ChMaterialSurfaceBase::DVI);
ChBody* body_array = new ChBody[5];
// ...
delte body;
delte[] body_array;
```

can only use the default constructor
Smart pointers

• In C/C++ programming, pointers are the main source of errors and bugs
  • Memory leaks, due to how pointers interact with memory (allocation/deallocation)
  • Dangling pointer (result of failing to delete a pointer to dynamically allocated memory)
  • Corrupted free store (result of “deleting” the same memory location twice)

• Solution: use smart pointers

• RAII – Resource Acquisition Is Initialization
  • Holding a resource is tied to the object lifetime
  • Resource allocation (acquisition) is done during object creation (initialization), by the constructor
  • Resource deallocation (release) is done during object destruction, by the destructor
  • If objects are destructed properly, no resource leaks occur
Shared pointers

• Smart pointers are essential to the RAII programming idiom
• Smart pointers are class objects that behave like built-in pointers
• Smart pointers support pointer operations:
  • dereferencing (operator *)
  • member operator (operator ->)

• Smart pointers do additional things that regular pointers do not: automatic memory management

• C++11 introduced comprehensive implementation of smart pointers
  • std::auto_ptr
  • std::shared_ptr
  • std::unique_ptr
  • std::weak_ptr
Common construct

```cpp
void foo() {
    myClass* p(new myClass);
    p->DoSomething();

    // ...

    delete p;
}
```

- This code will work fine (most of the time). What if somewhere in the function DoSomething() an exception gets thrown?
  - delete never gets called $\rightarrow$ memory leak

- Use of a smart pointer solves this issue because the smart pointer will be cleaned up whenever it gets out of scope (whether through normal execution or during an exception)
std::auto_ptr

• auto_ptr is a class template (available through the C++ Standard Library header <memory>) that provides basic RAII features for C++ raw pointers.

• The auto_ptr<T> template class describes an object that stores (wraps) a pointer to a single allocated object of type T* and ensures that the object to which it points is destroyed automatically when control leaves a scope.

```cpp
void foo() {
    std::auto_ptr<myClass> p(new myClass);
    p->DoSomething();

    // ...

    // delete p;
    // p's destructor called automatically as it goes out of scope
}
```
std::shared_ptr

- Introduced in C++11 (together with std::unique_ptr and std::weak_ptr)
- std::shared_ptr is a smart pointer; i.e., a C++ object with overloaded dereference and indirection operators
- std::shared_ptr is a reference-counted object; i.e., it holds (wraps) a pointer to an object and a pointer to a shared reference counter
- Every time a copy of the smart pointer is made, the reference counter is incremented
- When a shared pointer is destroyed, the reference counter is decremented
- When the counter reaches zero, the managed object (the wrapped raw pointer) is deleted (its destructor is called)
std::shared_ptr

x wraps a pointer to an int
here, the reference count is 1
new scope
make a copy of x.
y and x now share the same pointer to an int
here, the reference count is 2
y fell out of scope and was destroyed.
the reference count, previously shared by both x and y, is now decremented to 1.
on exit from main, x is destroyed.
the reference count is decremented to 0 and the wrapped pointer is deleted.

#include <memory>

int main(int argc, char** argv) {
    std::shared_ptr<int> x(new int(10));
    {
        std::shared_ptr<int> y = x;
    }
    return 0;
}
Prefer using std::make_shared

- When creating std::shared_ptr objects, prefer to use std::make_shared over explicitly using new with shared_ptr

```cpp
auto ball = std::make_shared<ChBody>(ChMaterialSurfaceBase::DEM);  // ✔️
auto ball = std::shared_ptr<ChBody>(new ChBody(ChMaterialSurfaceBase::DEM));  // ✗
```

- More efficient
- Control block (reference count) and owned block (wrapped pointer) can be allocated together
- One memory allocation instead of two (better cache efficiency)
- Better exception safety (avoids resource leaks)
std::weak_ptr & std::unique_ptr

• There are some situations where std::shared_ptr has problems (if the sharing graph has cycles, the reference counter cannot reach zero)
• std::weak_ptr can be used to break such a cycle

• std::unique_ptr is a smart pointer that models unique ownership, meaning that at any time in your program there shall be only one (owning) pointer to the pointed object
• std::unique_ptr is non-copyable

• std::weak_ptr and std::unique_ptr introduced in C++11
Inheritance and polymorphism
Inheritance

• Inheritance implements the “is a” relationship

• Example: Circle – Shape relationship
  • Circle is “a kind of a” Shape
  • Circle is “derived from” Shape
  • Circle is “a specialized” Shape
  • Circle is a “subclass” of Shape
  • Circle is a “derived class” of Shape
  • Shape is the “base class” of Circle

• Circle inherits properties and methods of Shape and adds its own behavior

• In C++, expressed through public inheritance:

```cpp
class Circle : public Shape {
    public:
        // ...
};
```
Polymorphism

- Polymorphism: a call to a member function will cause a different function to be executed depending on the object type

- **Inheritance** polymorphism
  - Public inheritance creates sub-types
  - Hinges crucially on the fact that a pointer to a derived class is *type-compatible* with a pointer to its base class
  - Typically refers to using *virtual methods*

- **Interface** polymorphism
  - Template parameters also induce a subtype relation
Virtual functions

• Inheritance polymorphism depends on public virtual member functions
  • Base class declares a member function virtual
  • Derived classes override the base class definition of that function

• Overriding happens only if the function signatures are the same
  • Otherwise, it just overloads the function or operator name

• Without virtual: you get “early binding”
  • which method gets called is decided at compile time, based on type of pointer you call through

• With virtual: you get “late binding”
  • which method gets called is decided at run time, based on type of pointed-to object

• Use final (C++11) to prevent (further) overriding of a virtual method
• Use override (C++11) in the derived class to ensure that the signatures match
  • compiler error otherwise
Virtual functions example

```cpp
class Animal {
public:
    void eat() { std::cout << "I'm eating generic food."; }
}

class Cat : public Animal {
public:
    void eat() { std::cout << "I'm eating a rat."; }
}

class Animal {
public:
    virtual void eat() { std::cout << "I'm eating generic food."; }
}

class Cat : public Animal {
public:
    virtual void eat() override { std::cout << "I'm eating a rat."; }
}

void func(Animal* xyz) {
    xyz->eat();
}

void func(Animal* xyz) {
    xyz->eat();
}

Animal* animal = new Animal;
Cat* cat = new Cat;

animal->eat(); // outputs: "I'm eating generic food."
cat->eat(); // outputs: "I'm eating a rat."

Animal* animal = new Animal;
Cat* cat = new Cat;

animal->eat(); // outputs: "I'm eating generic food."
cat->eat(); // outputs: "I'm eating a rat."

Animal* animal = new Animal;
Cat* cat = new Cat;

func(animal); // outputs: "I'm eating generic food." ✗
func(cat); // outputs: "I'm eating generic food." ✗

Animal* animal = new Animal;
Cat* cat = new Cat;

func(animal); // outputs: "I'm eating generic food." ✓
func(cat); // outputs: "I'm eating a rat." ✓
```

Abstract Base Classes (ABCs)

- Used to implement interfaces (and cleanly separate interface from implementation)
  - At design level, an ABC corresponds to an abstract concept
  - At programming level, an ABC is a base class that contains one or more pure virtual member functions

- An ABC cannot be instantiated
  - Cannot instantiate a class that declares pure virtual functions
  - Cannot instantiate a class that inherits pure virtual functions that are not overridden

- A pure virtual function is declared with =0

```cpp
class A {
public:
    virtual void foo() = 0;
};
```
The C++ Standard Template Library

• The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provides general-purpose templatized classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

• Components

**Containers**

Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.

**Algorithms**

Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.

**Iterators**

Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

http://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm
```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // create a vector to store int
    vector<int> vec;
    int i;

    // display the original size of vec
    cout << "vector size = " << vec.size() << endl;

    // push 5 values into the vector
    for (i = 0; i < 5; i++){
        vec.push_back(i);
    }

    // display extended size of vec
    cout << "extended vector size = " << vec.size() << endl;

    // access 5 values from the vector
    for (i = 0; i < 5; i++){
        cout << "value of vec [" << i << "] = " << vec[i] << endl;
    }

    // use iterator to access the values
    vector<int>::iterator v = vec.begin();
    while (v != vec.end()) {
        cout << "value of v = " << *v << endl;
        v++;
    }
    return 0;
}
```

- `push_back()` – inserts value at the end of the vector, expanding its size as needed
- `size()` – returns the size of the vector
- `begin()` – returns an iterator to the start of the vector
- `end()` – returns an iterator at the end of the vector

http://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm
References and Resources

• Bjarne Stroustrup, *The C++ Programming Language (fourth edition)*
• Bjarne Stroustrup, *A Tour of C++*
• Bjarne Stroustrup, *Programming – Principles and Practice Using C++*
• Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*
• Andrei Alexandrescu & Herb Sutter, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*
• Scott Meyer, *Overview of the New C++ (C++11/14)*

• The C++ super-FAQ – [https://isocpp.org/faq](https://isocpp.org/faq)