C++

By: Sancho McCann

What is C++?

- Compiled
- Statically-typed
- Supports: procedural, object oriented, and generic programming

Compiled		Interpreted
C++		Ruby
Objective-C		MATLAB
Fortran	Python	Perl
D		Scheme
Java		

Static typing

```
int x = 4;
int y = 10;
int z;
float f;
vector<int> v;

(x + y); // the type of every expression is known at
         // compile time

z = x + y; // assignments can only occur between
           // matching types,

f = x + y; // and types for which conversion is defined;

v = x + y; // attempts to do otherwise will be caught
           // at compile time
```

Multi-paradigm: Procedural

```
int signum(float a) {
  if (a < 0)
    return -1;
  else
    return 1;
}


int main() {
  float a = 4.5;
  int s = signum(a);

  return 0;
}
```

Multi-paradigm: Procedural

```
int str_length(char* str) {
  ...
}

int str_find(char* str, char* substr) {
  ...
}

int main() {
  char str1[10] = “Alpha”;
  char str2[10] = “Beta”;

  int len1 = str_length(str1);
  int len2 = str_length(str2);

  int index = str_find(str1, “a”);

  return 0;
}
```

Multi-paradigm: Object oriented

```
#include <string>

int main() {
  std::string str1(“Alpha”);
  std::string str2(“Beta”);

  int len1 = str1.length();
  int len2 = str2.length();

  int index = str1.find(“a”);

  return 0;
}
```

Multi-paradigm: Generic programming

```
template <typename T>
T max(T x, T y) {
  return x < y ? y : x;
}


int main() {
  float m1 = max(4.0, 5.4);
  int m2 = max(-2, 32);
  string lastInDictionary = max(string(“Alpha”), string(“Beta”);

  return 0;
}
```

What is C++?

- The language (syntax, semantics, and behaviour)
- A standard library (generic containers, algorithms, strings, streams, etc., and the c standard library)

Features

- C syntax
- Operators
- Program components
- Classes
- Inheritance and polymorphism
- Templates

C Syntax

- C++ designed to be as compatible with C as possible
- Some exceptions.. for example:
  - no implicit cast from `void*`
  - new keywords introduced

Operators

Arithmetic

Unary plus	`+a`	
Addition	`a + b`	
Prefix increment (decrement)	`++a`	`--a`
Postfix increment (decrement)	`a++`	`a--`
Assignment by addition	`a += b`	
Unary minus (negation)	`-a`	
Subtraction	`a - b`	
Assignment by subtraction	`a -= b`	
Multiplication	`a * b`	
Assignment by multiplication	`a *= b`	
Division	`a / b`	
Assignment by division	`a /= b`	
Modulus	`a % b`	
Assignment by modulus	`a %= b`	

Comparison

Less than	`a < b`
Less than or equal to	`a <= b`
Greater than	`a > b`
Greater than or equal to	`a >= b`
Not equal to	`a != b`
Equal to	`a == b`

Logical

Logical negation	`!a`
Logical AND	`a && b`
Logical OR	`a || b`

Operators

Other operators

Assignment	`a = b`
Function call	`a()`
Array subscript	`a[b]`
Indirection (dereference)	`*a`
Address-of (reference)	`&a`
Member by pointer	`a->b`
Member	`a.b`
Bind pointer to member by pointer	`a->*b`
Bind pointer to member by reference	`a.*b`
Cast	`(type) a`
Comma	`a , b`
Ternary conditional	`a ? b : c`
Scope resolution	`a::b`
Pointer to member	`a::*b`

size-of	**`sizeof`**`(a)`
	`sizeof``(type)`
Type identification	**`typeid`**`(a)`
	`typeid``(type)`
Allocate storage	**`new`** `type`
Allocate storage (array)	**`new`** `type`**`[`**`n`**`]`**
Deallocate storage	**`delete`** `a`
Deallocate storage (array)	**`delete[]`** `a`

const_cast	
static_cast	
dynamic_cast	
reinterpret_cast	

Program components

- Structure
- Control structures
- Functions
- Memory management

Structure

```
int main() {


  return 0;
}
```

Structure

```
#include <iostream>
using namespace std;

int main() {

  return 0;
}
```

Structure

```
#include <iostream>

int main() {
  std::cout << “Hello world” << std::endl;  // A comment.

  return 0;
}
```

Structure

```
#include <iostream>
using namespace std;

int main() {
  cout << “Hello world” << endl;  // A comment.

  return 0;
}
```

Control structures

```
#include <iostream>
using namespace std;

int main() {
  cout << “Hello world” << endl;  // A comment.

  for (int i = 0; i < 10; ++i) {
    cout << i << endl;
  }
  return 0;
}
```

Control structures

```
#include <iostream>
using namespace std;

int main() {
  cout << “Hello world” << endl;  // A comment.

  for (int i = 0; i < 10; ++i) {
    if (i % 3 == 0) {
      cout << i << endl;
    }
  }
  return 0;
}
```

Functions

```
int multiply(int a, int b) {
  return a * b;
}

int main() {
  int x = 3;
  int y = 5;
  int z = multiply(x, y);

  return 0;
}
```

Scope

```
int multiply(int a, int b) {
  return a * b;
}

int main() {
  int x = 3;
  int y = 5;
  int z = multiply(x, y);

  return 0;
}
```

symbols x, y, and z
are not in scope here

symbols a and b are
not in scope here

Scope

```
int c = 10;

int funky_multiply(int a, int b) {
    return a * (b + c);
}

int main() {
    int x = 3;
    int y = c + 2;
    int z = funky_multiply(x, y);

    return 0;
}
```

c is in scope throughout; it has global scope

Function arguments

```
int multiply(int a, int b) {
  return a * b;
}

int main() {
  int x = 3;
  int y = 5;
  int z = multiply(x, y);

  return 0;
}
```

x and y are passed "by copy" to the multiply function

Function arguments

```
int multiply(int & a, int & b) {
  return a * b;
}

int main() {
  int x = 3;
  int y = 5;
  int z = multiply(x, y);

  return 0;
}
```

x and y are passed "by reference" to the multiply function

Function arguments

```
BigObject multiply(BigObject & a, BigObject & b) {
  return a * b;
}

int main() {
  BigObject x();
  BigObject y();
  BigObject z = multiply(x, y);

  return 0;
}
```

more efficient when the arguments are large objects

Function arguments

```
BigObject multiply_wrong(BigObject & a, BigObject & b) {
  a = a + 15;
  return a * b;
}

int main() {
  BigObject x();
  BigObject y();
  BigObject z = multiply_wrong(x, y);

  return 0;
}
```

problem: a and b are now references to the original objects

Function arguments

```
BigObject multiply(const BigObject & a, const BigObject & b) {
  a = a + 15;  // This line causes a compiler error.

  return a * b;
}

int main() {
  BigObject x();
  BigObject y();
  BigObject z = multiply(x, y);

  return 0;
}
```

const arguments are not modifiable within the function

Function arguments

```
Example: operator arguments


Type1& operator+=(Type1& a, const Type2& b);  // Example: a += b.


Type1 operator*(const Type1& a, const Type2& b);  // Example a * b.
```

Memory management

You are responsible for deallocating (`delete`) memory that you allocate (`new`)

```
template <typename T>
void some_function(std::vector<T> & input) {
  T* temp = new T;  // Allocates memory and calls constructor

  // Do some things.

  delete(temp);  // Calls temp's destructor and deallocates memory.
}
```

Memory management

boost::scoped_ptr<T> makes this easy

```
template <typename T>
void some_function(std::vector<T> & input) {
  boost::scoped_ptr<T> temp;

  // Allocates memory, calls constructor, assigns memory management to
  // the scoped_ptr object.
  temp.reset(new T);

  // Do some things.

  // The scoped_ptr temp automatically calls delete on its pointer
  // when going out of scope.
}
```

Classes

- Class definitions define the object types

Modularity

Class definition declares member variables and functions

```
class Point
{
   int x, y;
public:
   void setLocation(int,int);
   void getX()
   {
      return x;
   }
   void getY()
   {
      return y;
   }
};

void Point::setLocation(int new_x, int new_y)
{
   x = new_x;
   y = new_y;
}

int main()
{
   Point p;
   p.setLocation(5,15);
   cout << “X coordinate: ” << p.getX() << endl;

   return 0;
}
```

Encapsulation

Members and functions are 'private' by default

```
class Point
{
   int x, y;
public:
   void setLocation(int,int);
   void getX()
   {
      return x;
   }
   void getY()
   {
      return y;
   }
};

void Point::setLocation(int new_x, int new_y)
{
   x = new_x;
   y = new_y;
}

int main()
{
   Point p;
   p.x = 5; // this line causes a compiler error
   p.setLocation(5,15);
   cout << "X coordinate: " << p.getX() << endl;

   return 0;
}
```

Encapsulation

The ‘public’ access specifier makes visible only the members you want visible

```
class Point
{
   int x, y;
public:
   void setLocation(int,int);
   void getX()
   {
      return x;
   }
   void getY()
   {
      return y;
   }
};

void Point::setLocation(int new_x, int new_y)
{
   x = new_x;
   y = new_y;
}

int main()
{
   Point p;
   p.setLocation(5,15);
   cout << “X coordinate: ” << p.getX() << endl;

   return 0;
}
```

Syntax details

Function definitions can occur with their delcaration (getX, getY), or be declared later (setLocation)

```
class Point
{
   int x, y;
public:
   void setLocation(int,int);
   void getX()
   {
      return x;
   }
   void getY()
   {
      return y;
   }
};

void Point::setLocation(int new_x, int new_y)
{
   x = new_x;
   y = new_y;
}

int main()
{
   Point p;
   p.setLocation(5,15);
   cout << “X coordinate: ” << p.getX() << endl;

   return 0;
}
```

```
class Point
{
   int x, y;
public:
   void setLocation(int,int);
   void getX()
   {
      return x;
   }
   void getY()
   {
      return y;
   }
};

void Point::setLocation(int new_x, int new_y)
{
   x = new_x;
   y = new_y;
}

int main()
{
   Point p;
   p.setLocation(5,15);
   cout << "X coordinate: " << p.getX() << endl;

   return 0;
}
```

Syntax details

setLocation isn't in the global scope

use the scope resolution operator to refer to Point's setLocation function

```
class Point
{
   int x, y;
public:
   Point(int, int);
   void setLocation(int,int);
   void getX();
   void getY();
};



int main()
{
   Point p(5,10);

   ...
```

Object creation

a constructor is automatically called when a new object of this class is created

Object destruction

```
class Point
{
   int *x, *y;
public:
   Point(int, int);
   ~Point();
   void setLocation(int,int);
   void getX();
   void getY();
};

Point::Point(int a, int b)
{
   x = new int;
   y = new int;
   *x = a;
   *y = b
}

Point::~Point()
{
   delete x;
   delete y;
}
```

You are responsible for deallocating memory that you allocate

Destructors allow you to take care of that responsibility

Called when that object goes out of scope or is deleted itself

```
class Point
{
   int x, y;
public:
   Point();
   Point(int, int);
   Point operator+(Point other);
};

Point Point::operator+(Point other)
{
   Point temp();
   temp.x = x + other.x;
   temp.y = y + other.y;
   return temp;
}


int main()
{
   Point a(2,4);
   Point b(5,-2);
   Point c = a + b;

   return 0;
}
```

Language detail

Most of C++'s operators can be 'overloaded': redefined for your particular class

```
class Point
{
   int x, y;
public:
   Point();
   Point(int, int);
   Point operator+(const Point & other) const;
};

Point Point::operator+(const Point & other) const
{
   Point temp();
   temp.x = x + other.x;
   temp.y = y + other.y;
   return temp;
}


int main()
{
   Point a(2,4);
   Point b(5,-2);
   Point c = a + b;

   return 0;
}
```

Language detail

Just for correctness

point.h

```
#ifndef SANCHO_POINT
#define SANCHO_POINT

namespace sancho {

class Point {
public:
   // Default constructor. This initializes
   // point to (0, 0).
   Point();
   // Constructs point with given location.
   Point(int, int);
   // Adds two points together.
   Point operator+(const Point & other) const;
private:
   int x_;
   int y_;
};

}  // End namespace sancho.
#endif
```

point.cc

```
#include “point.h”

namespace sancho {

Point::Point() {
   x_ = 0;
   y_ = 0;
}

Point::Point(int x, int y) : x_(x), y_(y) {
}

Point Point::operator+(const Point & other) const {
   return Point(x_ + other.x_, y_ + other.y_);
}

}  // End namespace sancho.
```

main.cc

```
#include “point.h”

int main() {
   sancho::Point a(5, 15);
   sancho::Point b(-4, 5);
   sancho::Point c = a + b;

   return 0;
}
```

Inheritance

$$\begin{tikzcd}
 & \text{Point} & \\
\text{NamedPoint} \arrow[ur] & & \text{IndexedPoint} \arrow[ul]
\end{tikzcd}$$

Inheritance

NamedPoint inherits all of the members of Point

```
class NamedPoint : public Point
{
   std::string name;
public:
   NamedPoint(int, int);
   NamedPoint(int, int, const std::string &);
   void print() const;
};

NamedPoint::NamedPoint(int x, int y) : Point(x,y)
{
   name = std::string(“”);
}

NamedPoint::NamedPoint(int x, int y, const std::string & newName) : Point(x,y)
{
   name = newName;
}

void NamedPoint::print() const
{
   std::cout << x << “,” << y << “,” << “name: “ << name << std::endl;
}
```

Inheritance

New members can be added to the derived class

```
class NamedPoint : public Point
{
   std::string name;
public:
   NamedPoint(int, int);
   NamedPoint(int, int, const std::string &);
   void print() const;
};

NamedPoint::NamedPoint(int x, int y) : Point(x,y)
{
   name = std::string(“”);
}

NamedPoint::NamedPoint(int x, int y, const std::string & newName) : Point(x,y)
{
   name = newName;
}

void NamedPoint::print() const
{
   std::cout << x << “,” << y << “,” << “name: “ << name << std::endl;
}
```

Constructors need to be redefined

Derived classes can defer some of the initialization to the base class

```
class NamedPoint : public Point
{
   std::string name;
public:
   NamedPoint(int, int);
   NamedPoint(int, int, const std::string &);
   void print() const;
};

NamedPoint::NamedPoint(int x, int y) : Point(x,y)
{
   name = std::string("");
}

NamedPoint::NamedPoint(int x, int y, const std::string & newName) : Point(x,y)
{
   name = newName;
}

void NamedPoint::print() const
{
   std::cout << x << "," << y << "," << "name: " << name << std::endl;
}
```

Polymorphism

Point

```
class Point
{
   int x, y;
public:
   Point(int, int);
};
```

NamedPoint

```
class NamedPoint : public Point
{
   std::string name;
public:
   NamedPoint(int, int);
   NamedPoint(int, int, const std::string &);
   void print() const; // prints x,y,name
};
```

IndexedPoint

```
class IndexedPoint : public Point
{
   int index;
public:
   IndexedPoint(int, int);
   IndexedPoint(int, int, int);
   void print() const; // prints x,y,id
};
```

```
int main()
{
   Point* p1 = new NamedPoint(0,0,"Zero Point");
   Point* p2 = new IndexedPoint(4,2,5423);
   NamedPoint* n = new NamedPoint(-1,-5,"Negative Point");

   n->print(); // works
   p1->print(); // doesn't work
   p2->print(); // doesn't work
}
```

Polymorphism

Point

```
class Point
{
   int x, y;
public:
   Point(int, int);
   void print() const; // prints x,y
};
```

NamedPoint

```
class NamedPoint : public Point
{
   std::string name;
public:
   NamedPoint(int, int);
   NamedPoint(int, int, const std::string &);
   void print() const; // prints x,y,name
};
```

IndexedPoint

```
class IndexedPont : public Point
{
   int index;
public:
   IndexedPoint(int, int);
   IndexedPoint(int, int, int);
   void print() const; // prints x,y,id
};
```

```
int main()
{
   Point* p1 = new NamedPoint(0,0,”Zero Point”);
   Point* p2 = new IndexedPoint(4,2,5423);
   NamedPoint* n = new NamedPoint(-1,-5,”Negative Point”);

   n->print(); // works
   p1->print(); // works, but only prints x,y
   p2->print(); // works, but only prints x,y
}
```

Polymorphism

Point

```
class Point
{
   int x, y;
public:
   Point(int, int);
   virtual void print() const; // prints x,y
};
```

NamedPoint

```
class NamedPoint : public Point
{
   std::string name;
public:
   NamedPoint(int, int);
   NamedPoint(int, int, const std::string &);
   void print() const; // prints x,y,name
};
```

IndexedPoint

```
class IndexedPont : public Point
{
   int index;
public:
   IndexedPoint(int, int);
   IndexedPoint(int, int, int);
   void print() const; // prints x,y,id
};
```

```
int main()
{
   Point* p1 = new NamedPoint(0,0,”Zero Point”);
   Point* p2 = new IndexedPoint(4,2,5423);
   NamedPoint* n = new NamedPoint(-1,-5,”Negative Point”);

   n->print(); // works
   p1->print(); // works, prints x,y,name
   p2->print(); // works, prints x,y,id
}
```

Classes: Summary

- Modularity
- Encapsulation
- Inheritance
- Polymorphism

Template classes

Allow classes to be written that operate on generic types.

```
template <typename T>
class Point
{
  T x, y;
public:
  typedef T result_type;

  Point(T x, T y) : x(x), y(y) {}

  template <typename U>
  Point<T> operator+(const Point<U> & other)
  {
    Point<T> temp(0,0);
    temp.x = x + other.getX();
    temp.y = y + other.getY();
    return temp;
  }

  T getX() const { return x; }
  T getY() const { return y; }
  void print() const;

};

int main()
{
   Point<int> a(1,2);
   Point<float> b(0.1, 0.2);

   Point< Point<int>::result_type > c = a + b;

   return 0;
}
```

Features

- C syntax
- Operators
- Program components
- Classes
- Inheritance and polymorphism
- Templates

How to use C++

- Only when you need to
- Use the standard template library
- Use the Boost C++ libraries
- Code according to a style guide

Standard Template Library

- Generic containers (queues, vectors, lists, sets, stacks)
- Iterators over those containers
- Generic algorithms

Standard Template Library

```
#include <vector>

int main()
{
   vector<int> a(10,0);
   ...
```

Boost C++ Libraries

- High quality, peer reviewed, generic libraries
- Eventual standardization (10+ libraries will be part of the new C++ standard)

Boost C++ Libraries

Accumulators
Bind
Date Time
Filesystem
Foreach
Generic Image Library
Interval values
Lambda functions
Math
Octonions
Quaterions
Statistical Distributions
Program Options
Random
Smart Pointers
Thread

Style guide

Google's C++ Style Guide: Consistency, Simplicity

In general, every `.cc` file should have an associated `.h` file.

When defining a function, parameter order is: inputs, then outputs.

In *`dir/foo`*`.cc` or *`dir/foo_test`*`.cc`, whose main purpose is to implement or test the stuff in *`dir2/foo2`*`.h`, order your includes as follows:

1. *`dir2/foo2`*`.h` (preferred location — see details below).
2. C system files.
3. C++ system files.
4. Other libraries' `.h` files.
5. Your project's `.h` files.

Use the C++ keyword `explicit` for constructors with one argument.

Composition is often more appropriate than inheritance. When using inheritance, make it `public`.

Questions