C++ Syntax Cheat Sheet

Preface

Since the C++ language varies so heavily between versions (e.g. C++0x, C++11, C++17, etc.), I will preface this cheat sheet by saying that the majority of the examples here target C++0x or c++11, as those are the versions that I am most familiar with. I come from the aerospace industry (embedded flight software) in which we purposefully don't use the latest technologies for safety reasons, so most of the code I write is in C++0x and sometimes C++11. Nevertheless, the basic concepts of C++ and object oriented programming still generally apply to both past and future versions of the language.

Table of Contents

1.0 C++ Classes

1.1 Class Syntax

1.1.1 Class Declaration (`.h` file)

Here's a simple class representing a polygon, a shape with any number of sides.

The class *declaration* typically goes in the header file, which has the extension `.h` (or, less commonly, `.hpp` to distinguish from C headers). The *declaration* gives the class name, any classes it may extend, declares the members and methods, and declares which members/methods are public, private, or protected. You can think of the declaration as sort of saying: "there will be a thing and here's how it will look like". The declaration is used to inform the compiler about the future essence and use of a particular symbol.

```
// File: polygon.h

#include <string>

class Polygon {

// Private members and methods are only accessible via methods in the class definiti
private:
    int num_sides;      // Number of sides

// Protected members and methods are only accessible in the class definition or by c
protected:
    std::string name;   // Name of the polygon

// Public members and methods are accessible to anyone who creates an instance of th
public:
    // Constructors
    Polygon(const int num_sides, const std::string & name); // <--- This constructor

    // Getters and Setters
    int GetNumSides(void) const;
    void SetNumSides(const int num_sides);

    std::string & GetName(void) const;
    void SetName(const std::string & name);

}; // <--- Don't forget the semicolon!
```


1.1.2 Class Definition (`.cpp` file)

The class *definition* typically goes in the `.cpp` file. The *definition* extends the declaration by providing an actual implementation of whatever it is that you're building. Continuing the example from the declaration, the definition can be thought of as saying: "Right, that thing I told you briefly about earlier? Here's how it actually functions". The definition thus provides the compileable implementation.

```cpp
// File: polygon.cpp

#include <string>       // <--- Required for std::string

#include "polygon.h"    // <--- Obtains the class declaration

// Constructor
// You must scope the method definitions with the class name (Polygon::)
// Also, see the section on the 'explicit' keyword for a warning about constructors
Polygon::Polygon(const int num_sides, const std::string & name) {
    this->num_sides = num_sides;        // 'this' is a pointer to the instance of th
    this->name = name;                  // In this case you need to use 'this->...'
}

// Get the number of sides
int Polygon::GetNumSides(void) const {  // The 'const' here tells the compiler that
    return this->num_sides;
}

// Set the number of sides
void Polygon::SetNumSides(const int num_sides) {
    this->num_sides = num_sides;
}

// Get the polygon name
std::string & Polygon::GetName(void) const {
    return this->name;
}

// Set the polygon name
void Polygon::SetName(const std::string & name) {
    this->name = name;
}
```


The getters and setters here don't do much, but you could imagine limiting the number of sides such that it must have at least 3 sides to be a useful polygon, in which case you could enforce that in `Polygon::SetNumSides()`. Of course, you'd also need to modify the constructor, which could then call `SetNumSides()` instead of setting the variable directly.

> NOTE: Regarding the use of `this->` in a class definition, there are places where it's strictly necessary for readability, e.g. when your method parameter shares the exact same name as a member variable, you use `this->` to avoid what's called shadowing. However, some prefer to always use `this->` explicitly regardless of whether it's necessary.

1.1.3 Class Utilization (Another `.cpp` file)

```
// File: main.cpp

#include <string>
#include <iostream>

#include "Polygon.h"    // <--- Obtains the class declaration

int main(int argc, char * argv[]) {
    // Create a polygon with 4 sides and the name "Rectangle"
    Polygon polygon = Polygon(4, "Rectangle");

    // Check number of sides -- Prints "Rectangle has 4 sides"
    std::cout << polygon.GetName() << " has " << polygon.GetNumSides() << " sides"<<

    // Change number of sides to 3 and rename to "Triangle"
    polygon.SetNumSides(3);
    polygon.SetName("Triangle");
}
```

1.1.4 Getters and Setters

A shortcut often used for Getters/Setters is to define them in the class declaration (`.h`) file as follows:

```
// File: car.h

#include <string>

class Car {
private:
    int year;
    std::string make;

public:
    int GetYear(void) const { return this->year; }
    void SetYear(const int year) { this->year = year; }
    std::string & GetMake(void) const { return this->make; }
    void SetMake(const std::string & make) { this->make = make; }
};
```

This is often used for very basic getters and setters, and also for basic constructors. In contrast, you'll nearly always find more complex methods defined in the `.cpp` file. One exception to this is with class templates, in which the entire templated class declaration and definition must reside in the header file.

Another important consideration: If you have getters and setters for all of your members, you may want to reconsider the design of your class. Sometimes having getters and setters for every member is indicative of poor planning of the class design and interface. In particular, setters should be used more thoughtfully. Could a variable be set once in the constructor and left constant thereafter? Does it need to be modified at all? Is it set somewhere else in another method, perhaps even indirectly?

1.2 Inheritance

A class can extend another class, meaning that the new class inherits all of the data from the other class, and can also override its methods, add new members, etc. Inheritance is the key feature required for polymorphism.

It is important to note that this feature is often overused by beginners and sometimes unnecessary hierarchies are created, adding to the overally complexity. There are some good alternatives such as composition and aggregation, although, of course, sometimes inheritance is exactly what is needed.

Example: the class `Rectangle` can inherit from the class `Polygon`. You would then say that a `Rectangle` extends from a `Polygon`, or that class `Rectangle` is a sub-class of `Polygon`. In plain English, this means that a `Rectangle` is a more specialized version of a `Polygon`. Thus, all rectangles are polygons, but not all polygons are rectangles.

1.2.1 `Rectangle` Declaration (`.h` file)


```
// File: rectangle.h

#include <string>       // <--- Explicitly include the string header, even though pc

#include "polygon.h"    // <--- You must include the declaration in order to extend

// We extend from Polygon by using the colon (:) and specifying which type of inheri
// will be used (public inheritance, in this case)

class Rectangle : public Polygon {
private:
    int length;
    int width;
```

```
    // <--- NOTE: The member variables 'num_sides' and 'name' are already inherited
    //            it's as if we sort of get them for free, since we are a sub-class

public:
    // Constructors
    explicit Rectangle(const std::string &name);
    Rectangle(const std::string &name, const int length, const int width);

    // Getters and Setters
    const int GetLength(void) const { return this->length; }
    void SetLength(const int) { this->length = length; }

    const int GetWidth(void) const { return this->width; }
    void SetWidth(const int) { this->width = width; }

    // <--- NOTE: Again, the getters/setters for 'num_sides' and 'name' are already

    // Other Methods
    const int Area(void) const;
};
```

> NOTE: The inheritance access specifier (`public` , `protected` , or `private`) is used to determine the type of inheritance. If this is omitted then `private` inheritance is used by default. **Public inheritance is by far the most common type of inheritance.**

1.2.2 `Rectangle` Definition (`.cpp` file)

```
// File: rectangle.cpp

#include "rectangle.h"  // <--- Only need to include 'Rectangle', since 'Polygon' is

// This constructor calls the superclass (Polygon) constructor and sets the name and
Rectangle::Rectangle(const std::string &name, const int length, const int width) : P
    this->length = length;
    this->width = width;
}

// This constructor calls the superclass (Polygon) constructor, but sets the length
// The explicit keyword is used to restrict the use of the constructor. See section
explicit Rectangle::Rectangle(const std::string &name) : Polygon(4, name) {
    this->length = 1;
    this->width = 1;
}

// Compute the area of the rectangle
int Rectangle::Area(void) const {
```

```
    return length * width;              // <--- Note that you don't explicitly need
}
```

1.2.3 `Rectangle` Utilization (Another `.cpp` file)

```
// File: main.cpp

#include <iostream>

#include "Rectangle.h"

int main(int argc, char *argv[]) {
    Rectangle rectangle = Rectangle("Square", 6, 6);

    // Prints "Square has 4 sides, and an area of 36"
    std::cout << rectangle.GetName() << " has " << rectangle.GetNumSides() << " side
}
```

1.3 Class Polymorphism

Polymorphism describes a system in which a common interface is used to manipulate objects of different types. In essence various classes can inherit from a common interface through which they make certain guarantees about which methods/variables are available for use. By adhering to this common interface, one can use a pointer to an object of the base interface type to call the methods of any number of extending classes. Using polymorphism one can say "I don't care what type this really is; I know it implements `Foo()` and `Bar()` because it inherits from this interface", which is a pretty nifty feature.

The `virtual` keyword is used to ensure runtime polymorphism for class methods. Additionally, an overriding method can be forced by the compiler by not providing a default implementation in the interface, which is done by setting the method to `= 0`, as will be shown later.

1.3.1 Motivation

Let's consider a similar class hierarchy using shapes as previously discussed. Considering a shape to be any 3 or more sided polygon from which we can compute certain attributes (like the shape's area), let's extend from it to create a rectangle class from which we can set the length/width and a circle class in which you can set the radius. **In both cases, we want to be able to compute the area of the shape.** This is a key observation that we will expand upon later.

For now, this (poorly implemented) shape class will suffice:

```
// File: shape.h

#include <cmath>        // needed for M_PI constant

class Shape {
    // We'll leave Shape empty for now... not very interesting yet
};

class Rectangle : public Shape {
private:
    double length;
    double width;

public:
    // Constructor using a member initializer list instead of assignment in the meth
    Rectangle(const double w, const double l) : width(w), length(l) {}

    // Compute the area of a rectangle
    double Area(void) const {
        return length * width;
    }
};

class Circle : public Shape {
private:
    double radius;

public:
    explicit Circle(double r) : radius(r) {}

    // Compute the area of a circle
    double Area(void) const {
        return M_PI * radius * radius;  // pi*r^2
    }
};
```

> NOTE: As shown here, you can put multiple classes in a single header, although in practice unless you have a good reason for doing so it's probably best to use a separate header file per class.

> NOTE: I'm not using default value initialization for member variables (i.e. `double length = 0;`) and I'm using parentheses `()` instead of braces `{}` for the initializer list since older compilers (pre-C++11) may not support the new syntax.

So, we have our two classes, `Rectangle` and `Circle` , but in this case inheriting from `Shape` isn't really buying us anything. To make use of polymorphism we need to pull the common `Area()` method into the base class as follows, by using virtual methods.

1.3.2 Virtual Methods

Imagine you want to have a pointer to a shape with which you want to compute the area of that shape. For example, maybe you want to hold shapes in some sort of data structure, but you don't want to limit yourself to just rectangles or just circles; you want to support all objects that call themselves a 'Shape'. Something like:

```
Rectangle rectangle(2.0, 5.0);
Circle circle(1.0);

// Point to the rectangle
Shape * unknown_shape = &rectangle; // Could point to *any* shape, Rectangle, Circle

unknown_shape->Area();  // Returns 10.0

// Point to the circle
unknown_shape = &circle;
unknown-shape->Area();  // Returns 3.14...
```


The way to achieve this is to use the `virtual` keyword on the base class methods, which specifies that when a pointer to a base class invokes the method of an object that it points to, it should determine, at runtime, the correct method to invoke. That is, when `unknown_shape` points to a `Rectangle` it invokes `Rectangle::Area()` and if `unknown_shape` points to a `Circle` it invokes `Circle::Area()` .

Virtual methods are employed as follows:

```
#include <cmath>

class Shape {
```

```
public:
    // Virtual destructor (VERY IMPORTANT, SEE NOTE BELOW)
    virtual ~Shape() {}

    // Virtual area method
    virtual double Area() const {
        return 0.0;
    }
};

class Rectangle : public Shape {
private:
    double length;
    double width;

public:
    Rectangle(double w, double l) : width(w), length(l) {}

    // Override the Shape::Area() method with an implementation specific to Rectangl
    double Area() const override {
        return length * width;
    }
};

class Circle : public Shape {
private:
    double radius;

public:
    explicit Circle(double t) : radius(r) {}

    // Override the Shape::Area() method with an implementation specific to Circle
    //
    // NOTE: there is an 'override' keyword that was introduced in C++11 and is opti
    // to enforce that the method is indeed an overriding method of a virtual base m
    // and is used as follows:
    double Area() const override {
        return M_PI * radius * radius; // pi*r^2
    }
};
```


> NOTE: It is very important that a default virtual destructor was included after adding the virtual `Area()` method to the base class. Whenever a base class includes even a single virtual method, it must include a virtual destructor so that the correct destructor(s) are called in the correct order when the object is eventually deleted.

This is called runtime polymorphism because the decision of which implementation of the `Area()` method to use is determined during program execution based on the type that the base is pointing at. It is implemented using the virtual table mechanism. In a nutshell: it is a little more expensive to use but it can be immensely useful. There is also compile-time polymorphism. Here is more on the differences between them.

In the example above, if a class extends from `Shape` but does not include an override of `Area()` then calling the `Area()` method will invoke the base class method which (in the implementation above) returns `0.0` .

In some cases, you may want to **enforce** that sub-classes implement this method. This is done by not providing a default implementation, thus making it what is called a *pure virtual* method.

```
class Shape {
public:
    virtual ~Shape() {}
    virtual double Area() const = 0;
};
```

In general a class with only pure virtual methods and a virtual destructor is called an *abstract class* or *interface* and is typically named as such (e.g. `ButtonInterface` , or similar). An interface class guarantees that all extending classes implement a specific method with a specific method signature.

1.4 Special Methods

1.4.1 Constructor and Destructor

All classes have at least one constructor and a destructor, even if they are not explicitly defined. The constructor and destructor assist in managing the lifetime of the object. The constructor is invoked when an object is created and the destructor is invoked when an object is destroyed (either by going out of scope or explicitly using `delete`).

The constructor establishes a class invariant, a set of assertions guaranteed to be true during the lifetime of the object, which is then removed when the destructor is called.

1.4.1.1 Use of `explicit` in Constructors

The `explicit` keyword should be used in single-argument constructors to avoid a situation in which the constructor is implicitly invoked when a single argument is given in place of an object. Consider the following `Array` class:

```
class Array {
private:
    int size;

public:
    // Constructor
    Array(int size) {
        this->size = size;
    }

    // Destructor
    ~Array() {}

    // Print the contents of the array
    Print(const Array & array) {
        // ...
    }
};
```

The following is now legal but ambiguous:

```
Array array = 12345;
```

It ends up being the equivalent of this:

```
Array array = Array(12345);
```

Perhaps that's okay, but what about the following:

```
array.Print(12345);
```

Uh-oh. That's now legal, compileable code, but what does it mean? It is extremely unclear to the user.

To fix this, declare the single-argument `Array` constructor as `explicit`:

```
class Array {
    int size;
public:
    explicit Array(int size) {
        this->size = size;
    }
```

```
    // ...
};
```

Now you can only use the print method as follows:

```
array.Print(Array(12345));
```

and the previous `array.Print(12345)` is now a syntax error.

1.4.1.2 Member Initializer Lists

Member initializer lists allow you to initialize member variables in the definition of a method. This turns out to provide some performance benefits for class-type member variables, since a call to the default constructor is avoided. For POD (plain old data) like ints and floats, though, it is the same as initializing them in the body of the method.

```
class Car {
private:
    int year;
    int miles;
    std::string make;

public:
    Car(const int year, const int miles, const std::string & make) : year(year), mil
};
```

Using the initializer list is basically the same as the following more verbose constructor implementation, notwithstanding the note above regarding performance:

```
Car(const int year, const int miles, const std::string & make) {
    this->year = year;
    this->miles = miles;
    this->make = make;
}
```

Since C++11 initializer lists have some added functionality and curly braces `{}` can be used instead of parentheses `()` in the initializer list, but to maintain compatibility with older compilers you may want to use parentheses. The same applies in general to initialization syntax when creating objects. Many people prefer braces, and in some cases it's necessary (e.g. vector containing [100, 1] or a vector of one hundred 1s?), but to support older compilers you may consider using parentheses.

1.4.2 `new` and `delete`

The `new` and `delete` operators (and their array counterparts, `new[]` and `delete[]`) are operators used to dynamically allocate memory for objects, much like C's `malloc()` and `free()`.

More on these operators can be found here.

When manually allocating memory dynamically, it is the responsibility of the programmer to manage the memory and properly delete objects that have been allocated.

1.4.3 Copy Constructor and Copy Assignment

Copy constructors and copy assigment operators allow one object to be constructed or assigned a copy of another object directly:

```
Foo a(10);
Foo b(a);   // (1): Copy via constructor
Foo c ;
c = a;  // (2): Copy via assignment operator
```

This is accomplished by supplying a copy constructor and an assigment operator overload, both of which have a special syntax where they accept a const reference to an object of their same type.


```
class Foo {
private:
    int data;

public:
    // Default (no argument) constructor
    Foo() : data(0) {}

    // Single argument constructor
    explicit Foo(const int v) : data(v) {}

    // Copy constructor
```

```
    Foo(const Foo & f) : data(f.data) {}

    // Copy assignment operator
    Foo & operator=(const Foo & f) {
        data = f.data;
        return *this;
    }
};
```

Note that the compiler will always provide a default constructor, a default copy constructor, and a default copy assignment operator, so for simple cases (like this trivial example) you will not have to implement them yourself. More info on this can be found here.

1.4.4 Move Constructor and Move Assignment

Sometimes instead of performing a copy you instead wish to completely move data from one object to another. This requires the use of a move constructor and move assignement operator.

```
class Movable {
private:
    Foo * data_ptr;

public:
    Movable(Foo data) : data_ptr(new Foo(data)) {}

    // Move constructor
    Movable(Movable && m) {
        // Point to the other object's data
        data_ptr = m.data_ptr;

        // Remove the other object's data pointer by
        // setting it to nullptr
        m.data_ptr = nullptr;
    }

    // Move assignment operator
    Movable & operator=(Movable && m) {
        data_ptr = m.data_ptr;
        m.data_ptr = nullptr;
        return *this;
    }

    ~Movable() {
        delete data_ptr;
```

```
    }
};
```

The move constructor and assignment operator can be used as follows:

```
Movable Bar() {
    // ...
}

int main() {
    Movable a(Bar());       // Using the move constructor
    Movable b = Bar();          // Using the move assignment operator
}
```

Since `Bar()` creates an object that won't be used elsewhere and is deleted after the call, we can use the move constructor or move assignment operator to move the data to our object.

A programming idiom called 'copy and swap' makes use of the move constructor and can be a useful idiom.

1.5 Operator Overloading

Operators such as `+`, `-`, `*`, etc. are familiar and ubiquitous when working with simple data types like integers and floating point numbers. These operators as well as others can also be overloaded to provide a clear syntactic meaning to your own classes. For example, when working with linear algebra you can overload the `+` operator to perform an element-wise addition of two vectors. Here's a brief example using complex numbers that allows you to use the `+` and `-` operators to easily add and subtract two complex numbers.

There are two main ways to do operator overloading. The first is using normal member functions. The second uses the `friend` keyword and non-member methods that have access to the private member variables of the class.

Using normal member functions (requires a getter method for the member variables):

```
// File: complex.h

class Complex {
private:
    double r = 0.0; // Real part, defaults to 0.0
    double i = 0.0; // Imaginary part, defaults to 0.0
```

```
public:
    Complex(const double r, const double i) : r(r), i(i) {}

    // Accessor methods
    double GetReal(void) const { return r; }
    double GetImaginary(void) const { return i; }

    // + Operator
    Complex operator+(const Complex & a, const Complex & b) {
        return Complex(a.GetReal() + b.GetReal(), a.GetImaginary() + b.GetImaginary(
    }

    // - Operator
    Complex operator-(const Complex& a, const Complex& b) {
        return Complex(a.GetReal() - b.GetReal(), a.GetImaginary() - b.GetImaginary(
    }
};
```

Using `friend` methods:

```
// File: complex.h

class Complex {
private:
    double r = 0.0; // Real part, defaults to 0.0
    double i = 0.0; // Imaginary part, defaults to 0.0

public:
    Complex(const double r, const double i) : r(r), i(i) {}

    // + Operator (declaration only)
    friend Complex operator+(const Complex & a, const Complex & b);

    // - Operator (declaration only)
    friend Complex operator-(const Complex& a, const Complex& b);
};

// These are NOT member functions
// They can also be defined inside the class body but leaving them outside
// is a clearer reminder that they are not part of the class
Complex operator+(const Complex & a, const Complex & b) {
    return Complex(a.r + b.r, a.i + b.i);
}

Complex operator-(const Complex& a, const Complex& b) {
```

```
    return Complex(a.r - b.r, a.i - b.i);
}
```

In either case, the new operators can be used as follows:

```
int main() {
    Complex a(1, 2);    //  1 + 2i
    Complex b(5, 3);    //  5 + 3i

    Complex c = a + b;  //  6 + 5i
    Complex d = a - b;  // -4 - 1i
}
```

It's also often useful to overload the output stream operator to provide a custom output string displaying the object's internal state in a human-readable format. This is done by overloading the `<<` operator and requires using the `<iostream>` functionality.

```
#include <iostream>

class Complex {
private:
    // ...
public:
    // ...

    friend std::ostream & operator<<(std::ostream & os, const Complex & c);
};

// Definition
// Again, this is NOT a member function!
std::ostream & operator<<(std::ostream & os, const Complex & c) {
    os << c.r << " + " << c.i << "i";
    return os;
}

int main() {
    Complex a {1, 2};
    Complex b {5, 3};

    std::cout << a;     // Prints: 1 + 2i
    std::cout << a + b; // Prints: 6 + 5i
}
```


You can also similiarly overload the input stream operator (`>>`), and can read more about the various operators here.

1.6 Templates

Templates are a very powerful abstraction allowing you to generate compile-time methods/classes/etc. for any number of types while writing only one implementation.

Say you have a method that adds two floating point number together, and another to add two integers together:

```
double Add(const double a, const double b) {
    return a + b;
}

int Add(const int a, const int b) {
    return a + b;
}
```

That's great, but since both floating point numbers and integers implement the `+` operator you can use a template to instead write one generic implementation of a method that can operate on doubles, ints, floats, and (in this case) any other type that implements the `+` operator.

A simple templatized version of `Add` would look something like this:

```
template <typename T>   // T becomes whatever type is used at compile-time
T Add(const T & a, const T & b) {
    return a + b;   // The type T must support the + operator
}

// Usages
int main() {
    Add<int>(3, 5);                 // int version
    Add<double>(3.2, 5.8);  // double
    Add(3.45f, 5.0f);       // implicit float version: we leave off the <float> here

    Complex a {1, 2};       // Custom class
    Complex b {5, 3};
    Add(a, b);              // Works because we added support for the + operator!
}
```


In this simple example the compiler would generate four different methods, one for each type. Templating allows you to write more concise and modular code at the expense of generating a larger executable (code bloat).

Templates are especially useful to create class templates. Class templates must be completely defined in a single header file.

```
// File: storage.h

template <class T>      // <--- 'class' is synonymous with 'typename'
class Container {
private:
    T data;
public:
    explicit Container(const T & d) : data(d) {}
};

// Usage
int main() {
    Container<int> a(1);
    Container<float> b(10.0f);
    Container<Container<int>> c(a);
}
```

> NOTE: More coming soon on templates...

Read more about templates here and here.

2.0 General C++ Syntax

2.1 Namespaces

In a large production project you may have thousands of symbols for various types, variables, methods, and so on. To avoid symbol names conflicting with one another you can use namespaces to logically separate symbol names in to broad categories. Namespaces are an inherent feature of C++; when you create a class and refer to a method as `ClassName::Method()` you are essentially using a namespace feature intrinsic to classes.

For a brief namespace example, suppose that you have two data structures, both of which implement a `Node` class. In the following code, namespaces are used to allow the compiler (and the programmer) to distinguish between the two types.

```
// File: list.h

namespace list {
```

```
template <typename T>
struct Node {
    Node * next;
    Node * prev;
    T data;
};

}; // namespace


// File: bst.h

namespace bst {

template <typename T>
struct Node {
    Node * left;
    Node * right;
    T data;
};

}; // namespace


// File: main.cpp
#include "list.h"
#include "bst.h"

int main() {
    list::Node<int> a;
    bst::Node<int> b;
};
```


The standard C++ library uses the namespace `std`, e.g. `std::cout`, `std::string`, `std::endl`, etc. While you can use a `using namespace foo;` directive to address symbols directly in the `foo` namespace without prefixing the `foo::` qualifier, this is generally considered bad practice as it pollutes the global namespace and sort of undermines the point of using namespaces in the first place.

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

cout << "Hello, World" << endl;             // <--- BAD: pollutes the global namespa
```

```
#include <iostream>

std::cout << "Hello, World" << std::endl;   // <--- GOOD: It's clear that you're usi
```

2.2 References and Pointers

Those familiar with C will be very intimately acquainted with pointers. C++ adds the concept of references, which is a powerful way to have *some* of the features of pointers while avoiding some of the pitfalls. Later versions of C++ also add smart pointers, which allow for better memory management and scoping via `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`, as compared to traditional raw pointers.

Raw pointers in C++ behave exactly the same way as they do in C: a pointer variable stores the address of whatever it is pointing to. You can think of pointers as essentially storing a link to another piece of data. You can access the data that the pointer points to with the `->` operator, or dereference it with the `*` operator.

References are more akin to an alias. References cannot be `NULL` or `nullptr`, and references cannot be reassigned to reference something else after they have been created. Additionally, references do not take up extra memory; they share the same address as whatever they reference to. References cannot have multiple levels of indirection (pointers can), and there is no reference arithmetic like there is for pointers. You can access the underlying data of a reference directly by using the reference itself: that is, if it's a reference to an integer it can be used as an integer. If it's a reference to a class you can access the class members directly with the `.` operator.

Although pointers are incredibly powerful, references are generally much safer, especially when passing objects to methods using pass-by-reference. It is very common in C++ code to pass an object as a `const` reference (if the data should be unmutable within the method) or a non-const reference rather than a raw pointer as is required in C.

More on references vs pointers here.

In the following code, assume a 32-bit system, in which case the size of a pointer variable is 4 bytes, and that the stack grows towards higher memory addresses.

```
// Pointers
int a = 10;                         // Ends up at memory address '0x2A000084', for e
int b = 20;                         // Ends up at memory address '0x2A000088'

int * ptr = nullptr;                // ptr is a separate variable whose type is 'poi
```

```
printf("ptr = %p\n");               // Prints: 0x0

ptr = &a;                           // The value of ptr is now the address of the va
std::cout << ptr << std::endl;      // Prints: 0x2a000084
std::cout << *ptr << std::endl;     // Prints: 10

ptr = &b;                           // The value of ptr is now the address of the va
std::cout << ptr << std::endl;      // Prints: 0x2a000088
std::cout << *ptr << std::endl;     // Prints: 20
```

```
// References
int a = 10;                         // Ends up at memory address '0x2A000084', for e
int b = 20;                         // Ends up at memory address '0x2A000088'

int & ref_a = a;                    // ref_a is an alias of (reference to) the varia
int & ref_b = b;                    // ref_b is an alias of (reference to) the varia

std::cout << ref_a << std::endl;    // Prints: 10
std::cout << ref_b << std::endl;    // Prints: 20

std::cout << &ref_a << std::endl;   // Prints: 0x2a000084
std::cout << &ref_b << std::endl;   // Prints: 0x2a000088

ref_a = b;                          // SETS THE VALUE OF 'a' TO THE VALUE OF 'b'!

std::cout << ref_a << std::endl;    // Prints: 20
std::cout << a << std::endl;        // ALSO PRINTS: 20 !

int & ref_c;                        // ERROR! References must be initialized at thei
```


Perhaps the most widely used aspect of references is to pass objects by reference (sometimes constant reference) to a method. To avoid hammering the stack with large objects when you pass them by value it is nearly always preferrable to pass by reference, which is the term used when using either a reference *or* a pointer. Using a reference allows you to pass any size object by reference, while still allowing you to access the object directly.

```
// Pass by reference using a const reference
void Foo(const Bar & bar) {
    int a = bar.GetValue();

    if (bar.SomeMethod()) {
        // ...
```

```
    }

    bar.SetValue(10);   // ERROR! Cannot modify a const reference!
}

// Pass by reference using a non-const reference
void Foo(Bar & bar) {
    int a = bar.GetValue();

    if (bar.SomeMethod()) {
        // ...
    }

    bar.SetValue(10);   // Modifies 'bar' and thus whatever 'bar' references
}
```

By passing an object by reference using a reference instead of a pointer you:

- Don't need to check for `NULL` or `nullptr` since references cannot be null
- Can access the referenced object's data directly instead of using the `->` operator or dereferencing a pointer
- Make it clearer which parameters are meant to be *input* parameters and which are meant to be *output* parameters by using `const` to denote strictly input parameters
- Gain the benefits of both passing by value and passing by reference since you don't need to use a lot of memory on the stack for your object

Thus, passing by reference using a `const` reference is essentially the same as passing by value, but you avoid copying the object onto the stack. Passing by reference using a non-const reference is essentially the same as passing by reference using a pointer, but you are guaranteed that it is not null and it's as if the pointer is effectively dereferenced.

2.3 Keywords

Reference

2.3.1 General Keywords

`asm` `auto` `const` `constexpr` (*since C++11*) `explicit` `export` (*until C++11*) `extern` (*language linkage*) `friend` `inline` `mutable` `noexcept` (*operator*) `noexcept` (*function specifier*) `nullptr` `override` `static` (*class member specifier*) `template` `this` `virtual` (*function specifier*) `virtual` (*base class specifier*) `volatile`

2.3.2 Storage Class Specifiers

Reference

- `auto` (*until C++11*)
- `register` (*until C++17*)
- `static`
- `extern`
- `thread_local` (*since C++11*)

2.3.3 `const` and `dynamic` Cast Conversion

- `const_cast`
- `dynamic_cast`

2.4 Preprocessor Tokens

- `#if` : Preprocessor version of `if(...)`
- `#elif` : Preprocessor version of `else if(...)`
- `#else` : Preprocessor version of `else`
- `#endif` : Used to end an `#if` , `#ifdef` , or `#ifndef`
- `defined()` : Returns true if the macro is defined
- `#ifdef` : Same as `#if defined(...)`
- `#ifndef` : Same as `#if !defined(...)`
- `#define` : Defines a text macro. See here for full explanation, including macro functions and predefined macros.
- `#undef` : Un-defines a text macro
- `#include` : Includes a source file
- `#line` : Changes the current file name and line number in the preprocessor
- `#error` : Prints an error message and stops compilation
- `#pragma` : Non-standard, can be used instead of header guards (`#ifndef HEADER_H` ...)

2.5 Strings (`std::string`)

Reference

2.6 Iterators (`std::iterator<...>`)

Reference

2.7 Exceptions

Reference

2.8 Lambdas

Reference