C++ is a systems programming language that, according to its inventor Bjarne Stroustrup, was designed to

- be a "better C"
- support data abstraction
- support object-oriented programming
- support generic programming

Though its syntax can be more difficult or complex than newer languages, it is widely used because it compiles to native instructions that can be directly run by the processor and offers tight control over hardware (like C) while offering high-level features such as generics, exceptions, and classes. This combination of speed and functionality makes C++ one of the most widely-used programming languages.

```
//////////////////
// Comparison to C
//////////////////

// C++ is _almost_ a superset of C and shares its basic syntax for
// variable declarations, primitive types, and functions.

// Just like in C, your program's entry point is a function called
// main with an integer return type.
// This value serves as the program's exit status.
// See http://en.wikipedia.org/wiki/Exit_status for more information.
int main(int argc, char** argv)
{
    // Command line arguments are passed in by argc and argv in the same way
    // they are in C.
    // argc indicates the number of arguments,
    // and argv is an array of C-style strings (char*)
    // representing the arguments.
    // The first argument is the name by which the program was called.
    // argc and argv can be omitted if you do not care about arguments,
    // giving the function signature of int main()

    // An exit status of 0 indicates success.
    return 0;
}

// However, C++ varies in some of the following ways:

// In C++, character literals are chars
sizeof('c') == sizeof(char) == 1

// In C, character literals are ints
sizeof('c') == sizeof(int)


// C++ has strict prototyping
void func(); // function which accepts no arguments

// In C
void func(); // function which may accept any number of arguments
```

```cpp
// Use nullptr instead of NULL in C++
int* ip = nullptr;

// C standard headers are available in C++,
// but are prefixed with "c" and have no .h suffix.
#include <cstdio>

int main()
{
    printf("Hello, world!\n");
    return 0;
}

//////////////////////
// Function overloading
//////////////////////

// C++ supports function overloading
// provided each function takes different parameters.

void print(char const* myString)
{
    printf("String %s\n", myString);
}

void print(int myInt)
{
    printf("My int is %d", myInt);
}

int main()
{
    print("Hello"); // Resolves to void print(const char*)
    print(15); // Resolves to void print(int)
}

/////////////////////////////
// Default function arguments
/////////////////////////////

// You can provide default arguments for a function
// if they are not provided by the caller.

void doSomethingWithInts(int a = 1, int b = 4)
{
    // Do something with the ints here
}

int main()
{
    doSomethingWithInts();      // a = 1,  b = 4
    doSomethingWithInts(20);    // a = 20, b = 4
    doSomethingWithInts(20, 5); // a = 20, b = 5
}
```

```
// Default arguments must be at the end of the arguments list.

void invalidDeclaration(int a = 1, int b) // Error!
{
}


/////////////
// Namespaces
/////////////

// Namespaces provide separate scopes for variable, function,
// and other declarations.
// Namespaces can be nested.

namespace First {
    namespace Nested {
        void foo()
        {
            printf("This is First::Nested::foo\n");
        }
    } // end namespace Nested
} // end namespace First

namespace Second {
    void foo()
    {
        printf("This is Second::foo\n");
    }
}

void foo()
{
    printf("This is global foo\n");
}

int main()
{
    // Includes all symbols from namespace Second into the current scope. Note
    // that simply foo() no longer works, since it is now ambiguous whether
    // we're calling the foo in namespace Second or the top level.
    using namespace Second;

    Second::foo(); // prints "This is Second::foo"
    First::Nested::foo(); // prints "This is First::Nested::foo"
    ::foo(); // prints "This is global foo"
}

///////////////
// Input/Output
///////////////

// C++ input and output uses streams
```

```
// cin, cout, and cerr represent stdin, stdout, and stderr.
// << is the insertion operator and >> is the extraction operator.

#include <iostream> // Include for I/O streams

using namespace std; // Streams are in the std namespace (standard library)

int main()
{
   int myInt;

   // Prints to stdout (or terminal/screen)
   cout << "Enter your favorite number:\n";
   // Takes in input
   cin >> myInt;

   // cout can also be formatted
   cout << "Your favorite number is " << myInt << "\n";
   // prints "Your favorite number is <myInt>"

    cerr << "Used for error messages";
}

//////////
// Strings
//////////

// Strings in C++ are objects and have many member functions
#include <string>

using namespace std; // Strings are also in the namespace std (standard library)

string myString = "Hello";
string myOtherString = " World";

// + is used for concatenation.
cout << myString + myOtherString; // "Hello World"

cout << myString + " You"; // "Hello You"

// C++ strings are mutable and have value semantics.
myString.append(" Dog");
cout << myString; // "Hello Dog"


/////////////
// References
/////////////

// In addition to pointers like the ones in C,
// C++ has _references_.
// These are pointer types that cannot be reassigned once set
// and cannot be null.
// They also have the same syntax as the variable itself:
```

```
// No * is needed for dereferencing and
// & (address of) is not used for assignment.

using namespace std;

string foo = "I am foo";
string bar = "I am bar";


string& fooRef = foo; // This creates a reference to foo.
fooRef += ". Hi!"; // Modifies foo through the reference
cout << fooRef; // Prints "I am foo. Hi!"

// Doesn't reassign "fooRef". This is the same as "foo = bar", and
//   foo == "I am bar"
// after this line.
cout << &fooRef << endl; //Prints the address of foo
fooRef = bar;
cout << &fooRef << endl; //Still prints the address of foo
cout << fooRef;  // Prints "I am bar"

//The address of fooRef remains the same, i.e. it is still referring to foo.


const string& barRef = bar; // Create a const reference to bar.
// Like C, const values (and pointers and references) cannot be modified.
barRef += ". Hi!"; // Error, const references cannot be modified.

// Sidetrack: Before we talk more about references, we must introduce a concept
// called a temporary object. Suppose we have the following code:
string tempObjectFun() { ... }
string retVal = tempObjectFun();

// What happens in the second line is actually:
//   - a string object is returned from tempObjectFun
//   - a new string is constructed with the returned object as argument to the
//     constructor
//   - the returned object is destroyed
// The returned object is called a temporary object. Temporary objects are
// created whenever a function returns an object, and they are destroyed at the
// end of the evaluation of the enclosing expression (Well, this is what the
// standard says, but compilers are allowed to change this behavior. Look up
// "return value optimization" if you're into this kind of details). So in this
// code:
foo(bar(tempObjectFun()))

// assuming foo and bar exist, the object returned from tempObjectFun is
// passed to bar, and it is destroyed before foo is called.

// Now back to references. The exception to the "at the end of the enclosing
// expression" rule is if a temporary object is bound to a const reference, in
// which case its life gets extended to the current scope:

void constReferenceTempObjectFun() {
```

```
  // constRef gets the temporary object, and it is valid until the end of this
  // function.
  const string& constRef = tempObjectFun();
  ...
}

// Another kind of reference introduced in C++11 is specifically for temporary
// objects. You cannot have a variable of its type, but it takes precedence in
// overload resolution:

void someFun(string& s) { ... }  // Regular reference
void someFun(string&& s) { ... }  // Reference to temporary object

string foo;
someFun(foo);  // Calls the version with regular reference
someFun(tempObjectFun());  // Calls the version with temporary reference

// For example, you will see these two versions of constructors for
// std::basic_string:
basic_string(const basic_string& other);
basic_string(basic_string&& other);

// Idea being if we are constructing a new string from a temporary object (which
// is going to be destroyed soon anyway), we can have a more efficient
// constructor that "salvages" parts of that temporary string. You will see this
// concept referred to as "move semantics".

/////////////////////
// Enums
/////////////////////

// Enums are a way to assign a value to a constant most commonly used for
// easier visualization and reading of code
enum ECarTypes
{
  Sedan,
  Hatchback,
  SUV,
  Wagon
};

ECarTypes GetPreferredCarType()
{
    return ECarTypes::Hatchback;
}

// As of C++11 there is an easy way to assign a type to the enum which can be
// useful in serialization of data and converting enums back-and-forth between
// the desired type and their respective constants
enum ECarTypes : uint8_t
{
  Sedan, // 0
  Hatchback, // 1
  SUV = 254, // 254
```

```
  Hybrid // 255
};

void WriteByteToFile(uint8_t InputValue)
{
    // Serialize the InputValue to a file
}

void WritePreferredCarTypeToFile(ECarTypes InputCarType)
{
    // The enum is implicitly converted to a uint8_t due to its declared enum type
    WriteByteToFile(InputCarType);
}

// On the other hand you may not want enums to be accidentally cast to an integer
// type or to other enums so it is instead possible to create an enum class which
// won't be implicitly converted
enum class ECarTypes : uint8_t
{
  Sedan, // 0
  Hatchback, // 1
  SUV = 254, // 254
  Hybrid // 255
};

void WriteByteToFile(uint8_t InputValue)
{
    // Serialize the InputValue to a file
}

void WritePreferredCarTypeToFile(ECarTypes InputCarType)
{
    // Won't compile even though ECarTypes is a uint8_t due to the enum
    // being declared as an "enum class"!
    WriteByteToFile(InputCarType);
}

//////////////////////////////////////////
// Classes and object-oriented programming
//////////////////////////////////////////

// First example of classes
#include <iostream>

// Declare a class.
// Classes are usually declared in header (.h or .hpp) files.
class Dog {
    // Member variables and functions are private by default.
    std::string name;
    int weight;

// All members following this are public
// until "private:" or "protected:" is found.
public:
```

```cpp
    // Default constructor
    Dog();

    // Member function declarations (implementations to follow)
    // Note that we use std::string here instead of placing
    // using namespace std;
    // above.
    // Never put a "using namespace" statement in a header.
    void setName(const std::string& dogsName);

    void setWeight(int dogsWeight);

    // Functions that do not modify the state of the object
    // should be marked as const.
    // This allows you to call them if given a const reference to the object.
    // Also note the functions must be explicitly declared as _virtual_
    // in order to be overridden in derived classes.
    // Functions are not virtual by default for performance reasons.
    virtual void print() const;

    // Functions can also be defined inside the class body.
    // Functions defined as such are automatically inlined.
    void bark() const { std::cout << name << " barks!\n"; }

    // Along with constructors, C++ provides destructors.
    // These are called when an object is deleted or falls out of scope.
    // This enables powerful paradigms such as RAII
    // (see below)
    // The destructor should be virtual if a class is to be derived from;
    // if it is not virtual, then the derived class' destructor will
    // not be called if the object is destroyed through a base-class reference
    // or pointer.
    virtual ~Dog();

}; // A semicolon must follow the class definition.

// Class member functions are usually implemented in .cpp files.
Dog::Dog()
{
    std::cout << "A dog has been constructed\n";
}

// Objects (such as strings) should be passed by reference
// if you are modifying them or const reference if you are not.
void Dog::setName(const std::string& dogsName)
{
    name = dogsName;
}

void Dog::setWeight(int dogsWeight)
{
    weight = dogsWeight;
}
```

```
// Notice that "virtual" is only needed in the declaration, not the definition.
void Dog::print() const
{
    std::cout << "Dog is " << name << " and weighs " << weight << "kg\n";
}

Dog::~Dog()
{
    cout << "Goodbye " << name << "\n";
}

int main() {
    Dog myDog; // prints "A dog has been constructed"
    myDog.setName("Barkley");
    myDog.setWeight(10);
    myDog.print(); // prints "Dog is Barkley and weighs 10 kg"
    return 0;
} // prints "Goodbye Barkley"

// Inheritance:

// This class inherits everything public and protected from the Dog class
// as well as private but may not directly access private members/methods
// without a public or protected method for doing so
class OwnedDog : public Dog {

    void setOwner(const std::string& dogsOwner);

    // Override the behavior of the print function for all OwnedDogs. See
    // http://en.wikipedia.org/wiki/Polymorphism_(computer_science)#Subtyping
    // for a more general introduction if you are unfamiliar with
    // subtype polymorphism.
    // The override keyword is optional but makes sure you are actually
    // overriding the method in a base class.
    void print() const override;

private:
    std::string owner;
};

// Meanwhile, in the corresponding .cpp file:

void OwnedDog::setOwner(const std::string& dogsOwner)
{
    owner = dogsOwner;
}

void OwnedDog::print() const
{
    Dog::print(); // Call the print function in the base Dog class
    std::cout << "Dog is owned by " << owner << "\n";
    // Prints "Dog is <name> and weights <weight>"
    //        "Dog is owned by <owner>"
```

```
}

/////////////////////////////////////////
// Initialization and Operator Overloading
/////////////////////////////////////////

// In C++ you can overload the behavior of operators such as +, -, *, /, etc.
// This is done by defining a function which is called
// whenever the operator is used.

#include <iostream>
using namespace std;

class Point {
public:
    // Member variables can be given default values in this manner.
    double x = 0;
    double y = 0;

    // Define a default constructor which does nothing
    // but initialize the Point to the default value (0, 0)
    Point() { };

    // The following syntax is known as an initialization list
    // and is the proper way to initialize class member values
    Point (double a, double b) :
        x(a),
        y(b)
    { /* Do nothing except initialize the values */ }

    // Overload the + operator.
    Point operator+(const Point& rhs) const;

    // Overload the += operator
    Point& operator+=(const Point& rhs);

    // It would also make sense to add the - and -= operators,
    // but we will skip those for brevity.
};

Point Point::operator+(const Point& rhs) const
{
    // Create a new point that is the sum of this one and rhs.
    return Point(x + rhs.x, y + rhs.y);
}

Point& Point::operator+=(const Point& rhs)
{
    x += rhs.x;
    y += rhs.y;
    return *this;
}

int main () {
```

```
    Point up (0,1);
    Point right (1,0);
    // This calls the Point + operator
    // Point up calls the + (function) with right as its parameter
    Point result = up + right;
    // Prints "Result is upright (1,1)"
    cout << "Result is upright (" << result.x << ',' << result.y << ")\n";
    return 0;
}

/////////////////////
// Templates
/////////////////////

// Templates in C++ are mostly used for generic programming, though they are
// much more powerful than generic constructs in other languages. They also
// support explicit and partial specialization and functional-style type
// classes; in fact, they are a Turing-complete functional language embedded
// in C++!

// We start with the kind of generic programming you might be familiar with. To
// define a class or function that takes a type parameter:
template<class T>
class Box {
public:
    // In this class, T can be used as any other type.
    void insert(const T&) { ... }
};

// During compilation, the compiler actually generates copies of each template
// with parameters substituted, so the full definition of the class must be
// present at each invocation. This is why you will see template classes defined
// entirely in header files.

// To instantiate a template class on the stack:
Box<int> intBox;

// and you can use it as you would expect:
intBox.insert(123);

// You can, of course, nest templates:
Box<Box<int> > boxOfBox;
boxOfBox.insert(intBox);

// Until C++11, you had to place a space between the two '>'s, otherwise '>>'
// would be parsed as the right shift operator.

// You will sometimes see
//   template<typename T>
// instead. The 'class' keyword and 'typename' keywords are _mostly_
// interchangeable in this case. For the full explanation, see
//   http://en.wikipedia.org/wiki/Typename
// (yes, that keyword has its own Wikipedia page).
```

```
// Similarly, a template function:
template<class T>
void barkThreeTimes(const T& input)
{
    input.bark();
    input.bark();
    input.bark();
}

// Notice that nothing is specified about the type parameters here. The compiler
// will generate and then type-check every invocation of the template, so the
// above function works with any type 'T' that has a const 'bark' method!

Dog fluffy;
fluffy.setName("Fluffy")
barkThreeTimes(fluffy); // Prints "Fluffy barks" three times.

// Template parameters don't have to be classes:
template<int Y>
void printMessage() {
  cout << "Learn C++ in " << Y << " minutes!" << endl;
}

// And you can explicitly specialize templates for more efficient code. Of
// course, most real-world uses of specialization are not as trivial as this.
// Note that you still need to declare the function (or class) as a template
// even if you explicitly specified all parameters.
template<>
void printMessage<10>() {
  cout << "Learn C++ faster in only 10 minutes!" << endl;
}

printMessage<20>();  // Prints "Learn C++ in 20 minutes!"
printMessage<10>();  // Prints "Learn C++ faster in only 10 minutes!"


/////////////////////
// Exception Handling
/////////////////////

// The standard library provides a few exception types
// (see http://en.cppreference.com/w/cpp/error/exception)
// but any type can be thrown an as exception
#include <exception>
#include <stdexcept>

// All exceptions thrown inside the _try_ block can be caught by subsequent
// _catch_ handlers.
try {
    // Do not allocate exceptions on the heap using _new_.
    throw std::runtime_error("A problem occurred");
}

// Catch exceptions by const reference if they are objects
```

```cpp
catch (const std::exception& ex)
{
    std::cout << ex.what();
}

// Catches any exception not caught by previous _catch_ blocks
catch (...)
{
    std::cout << "Unknown exception caught";
    throw; // Re-throws the exception
}

///////
// RAII
///////

// RAII stands for "Resource Acquisition Is Initialization".
// It is often considered the most powerful paradigm in C++
// and is the simple concept that a constructor for an object
// acquires that object's resources and the destructor releases them.

// To understand how this is useful,
// consider a function that uses a C file handle:
void doSomethingWithAFile(const char* filename)
{
    // To begin with, assume nothing can fail.

    FILE* fh = fopen(filename, "r"); // Open the file in read mode.

    doSomethingWithTheFile(fh);
    doSomethingElseWithIt(fh);

    fclose(fh); // Close the file handle.
}

// Unfortunately, things are quickly complicated by error handling.
// Suppose fopen can fail, and that doSomethingWithTheFile and
// doSomethingElseWithIt return error codes if they fail.
//  (Exceptions are the preferred way of handling failure,
//   but some programmers, especially those with a C background,
//   disagree on the utility of exceptions).
// We now have to check each call for failure and close the file handle
// if a problem occurred.
bool doSomethingWithAFile(const char* filename)
{
    FILE* fh = fopen(filename, "r"); // Open the file in read mode
    if (fh == nullptr) // The returned pointer is null on failure.
        return false; // Report that failure to the caller.

    // Assume each function returns false if it failed
    if (!doSomethingWithTheFile(fh)) {
        fclose(fh); // Close the file handle so it doesn't leak.
        return false; // Propagate the error.
    }
```

```cpp
    if (!doSomethingElseWithIt(fh)) {
        fclose(fh); // Close the file handle so it doesn't leak.
        return false; // Propagate the error.
    }

    fclose(fh); // Close the file handle so it doesn't leak.
    return true; // Indicate success
}

// C programmers often clean this up a little bit using goto:
bool doSomethingWithAFile(const char* filename)
{
    FILE* fh = fopen(filename, "r");
    if (fh == nullptr)
        return false;

    if (!doSomethingWithTheFile(fh))
        goto failure;

    if (!doSomethingElseWithIt(fh))
        goto failure;

    fclose(fh); // Close the file
    return true; // Indicate success

failure:
    fclose(fh);
    return false; // Propagate the error
}

// If the functions indicate errors using exceptions,
// things are a little cleaner, but still sub-optimal.
void doSomethingWithAFile(const char* filename)
{
    FILE* fh = fopen(filename, "r"); // Open the file in read mode
    if (fh == nullptr)
        throw std::runtime_error("Could not open the file.");

    try {
        doSomethingWithTheFile(fh);
        doSomethingElseWithIt(fh);
    }
    catch (...) {
        fclose(fh); // Be sure to close the file if an error occurs.
        throw; // Then re-throw the exception.
    }

    fclose(fh); // Close the file
    // Everything succeeded
}

// Compare this to the use of C++'s file stream class (fstream)
// fstream uses its destructor to close the file.
// Recall from above that destructors are automatically called
```

```
// whenever an object falls out of scope.
void doSomethingWithAFile(const std::string& filename)
{
    // ifstream is short for input file stream
    std::ifstream fh(filename); // Open the file

    // Do things with the file
    doSomethingWithTheFile(fh);
    doSomethingElseWithIt(fh);

} // The file is automatically closed here by the destructor

// This has _massive_ advantages:
// 1. No matter what happens,
//    the resource (in this case the file handle) will be cleaned up.
//    Once you write the destructor correctly,
//    It is _impossible_ to forget to close the handle and leak the resource.
// 2. Note that the code is much cleaner.
//    The destructor handles closing the file behind the scenes
//    without you having to worry about it.
// 3. The code is exception safe.
//    An exception can be thrown anywhere in the function and cleanup
//    will still occur.

// All idiomatic C++ code uses RAII extensively for all resources.
// Additional examples include
// - Memory using unique_ptr and shared_ptr
// - Containers - the standard library linked list,
//   vector (i.e. self-resizing array), hash maps, and so on
//   all automatically destroy their contents when they fall out of scope.
// - Mutexes using lock_guard and unique_lock

// containers with object keys of non-primitive values (custom classes) require
// compare function in the object itself or as a function pointer. Primitives
// have default comparators, but you can override it.
class Foo {
public:
    int j;
    Foo(int a) : j(a) {}
};
struct compareFunction {
    bool operator()(const Foo& a, const Foo& b) const {
        return a.j < b.j;
    }
};
//this isn't allowed (although it can vary depending on compiler)
//std::map<Foo, int> fooMap;
std::map<Foo, int, compareFunction> fooMap;
fooMap[Foo(1)]  = 1;
fooMap.find(Foo(1)); //true

///////////////////////////////////////
// Lambda Expressions (C++11 and above)
///////////////////////////////////////
```

```
// lambdas are a convenient way of defining an anonymous function
// object right at the location where it is invoked or passed as
// an argument to a function.

// For example, consider sorting a vector of pairs using the second
// value of the pair

vector<pair<int, int> > tester;
tester.push_back(make_pair(3, 6));
tester.push_back(make_pair(1, 9));
tester.push_back(make_pair(5, 0));

// Pass a lambda expression as third argument to the sort function
// sort is from the <algorithm> header

sort(tester.begin(), tester.end(), [](const pair<int, int>& lhs, const pair<int, int>& rhs) {
        return lhs.second < rhs.second;
    });

// Notice the syntax of the lambda expression,
// [] in the lambda is used to "capture" variables
// The "Capture List" defines what from the outside of the lambda should be available inside the functi
// It can be either:
//     1. a value : [x]
//     2. a reference : [&x]
//     3. any variable currently in scope by reference [&]
//     4. same as 3, but by value [=]
// Example:

vector<int> dog_ids;
// number_of_dogs = 3;
for(int i = 0; i < 3; i++) {
    dog_ids.push_back(i);
}

int weight[3] = {30, 50, 10};

// Say you want to sort dog_ids according to the dogs' weights
// So dog_ids should in the end become: [2, 0, 1]

// Here's where lambda expressions come in handy

sort(dog_ids.begin(), dog_ids.end(), [&weight](const int &lhs, const int &rhs) {
        return weight[lhs] < weight[rhs];
    });
// Note we captured "weight" by reference in the above example.
// More on Lambdas in C++ : http://stackoverflow.com/questions/7627098/what-is-a-lambda-expression-in-c

///////////////////////////////
// Range For (C++11 and above)
///////////////////////////////

// You can use a range for loop to iterate over a container
```

```
int arr[] = {1, 10, 3};

for(int elem: arr){
    cout << elem << endl;
}

// You can use "auto" and not worry about the type of the elements of the container
// For example:

for(auto elem: arr) {
    // Do something with each element of arr
}

/////////////////////
// Fun stuff
/////////////////////

// Aspects of C++ that may be surprising to newcomers (and even some veterans).
// This section is, unfortunately, wildly incomplete; C++ is one of the easiest
// languages with which to shoot yourself in the foot.

// You can override private methods!
class Foo {
  virtual void bar();
};
class FooSub : public Foo {
  virtual void bar();  // Overrides Foo::bar!
};


// 0 == false == NULL (most of the time)!
bool* pt = new bool;
*pt = 0; // Sets the value points by 'pt' to false.
pt = 0;  // Sets 'pt' to the null pointer. Both lines compile without warnings.

// nullptr is supposed to fix some of that issue:
int* pt2 = new int;
*pt2 = nullptr; // Doesn't compile
pt2 = nullptr;  // Sets pt2 to null.

// There is an exception made for bools.
// This is to allow you to test for null pointers with if(!ptr),
// but as a consequence you can assign nullptr to a bool directly!
*pt = nullptr;  // This still compiles, even though '*pt' is a bool!


// '=' != '=' != '='!
// Calls Foo::Foo(const Foo&) or some variant (see move semantics) copy
// constructor.
Foo f2;
Foo f1 = f2;

// Calls Foo::Foo(const Foo&) or variant, but only copies the 'Foo' part of
// 'fooSub'. Any extra members of 'fooSub' are discarded. This sometimes
```

```
// horrifying behavior is called "object slicing."
FooSub fooSub;
Foo f1 = fooSub;

// Calls Foo::operator=(Foo&) or variant.
Foo f1;
f1 = f2;


// How to truly clear a container:
class Foo { ... };
vector<Foo> v;
for (int i = 0; i < 10; ++i)
  v.push_back(Foo());

// Following line sets size of v to 0, but destructors don't get called
// and resources aren't released!
v.empty();
v.push_back(Foo());  // New value is copied into the first Foo we inserted

// Truly destroys all values in v. See section about temporary objects for
// explanation of why this works.
v.swap(vector<Foo>());
```

Further Reading:

An up-to-date language reference can be found at http://cppreference.com/w/cpp

Additional resources may be found at http://cplusplus.com