```
//////////////////
// 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

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

// C standard headers are available in C++.
// C headers end in .h, while
// C++ headers are prefixed with "c" and have no ".h" suffix.

// The C++ standard version:
#include <cstdio>

// The C standard version:
#include <stdio.h>

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.
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 enu
m type
        WriteByteToFile(InputCarType);
}

// On the other hand you may not want enums to be accidentally cast to an intege
r
// type or to other enums so it is instead possible to create an enum class whic
h
// 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:

    // 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()
{
    std::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 {

public:
    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);
}

// It's good practice to return a reference to the leftmost variable of
// an assignment. `(a += b) == c` will work this way.
Point& Point::operator+=(const Point& rhs)
{
    x += rhs.x;
    y += rhs.y;

    // `this` is a pointer to the object, on which a method is called.
    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 as an 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
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.
    }
    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 shared_ptrread 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


////////////////////
// Smart Pointer
////////////////////

// Generally a smart pointer is a class which wraps a "raw pointer" (usage of "n
ew"
// respectively malloc/calloc in C). The goal is to be able to
// manage the lifetime of the object being pointed to without ever needing to ex
plicitly delete
// the object. The term itself simply describes a set of pointers with the
// mentioned abstraction.
// Smart pointers should preferred over raw pointers, to prevent
// risky memory leaks, which happen if you forget to delete an object.

// Usage of a raw pointer:
Dog* ptr = new Dog();
ptr->bark();
delete ptr;

// By using a smart pointer, you don't have to worry about the deletion
// of the object anymore.
// A smart pointer describes a policy, to count the references to the
// pointer. The object gets destroyed when the last
// reference to the object gets destroyed.

// Usage of "std::shared_ptr":
void foo()
{
// It's no longer necessary to delete the Dog.
std::shared_ptr<Dog> doggo(new Dog());
doggo->bark();
}

// Beware of possible circular references!!!
// There will be always a reference, so it will be never destroyed!
std::shared_ptr<Dog> doggo_one(new Dog());
std::shared_ptr<Dog> doggo_two(new Dog());
doggo_one = doggo_two; // p1 references p2
```

```
doggo_two = doggo_one; // p2 references p1

// There are several kinds of smart pointers.
// The way you have to use them is always the same.
// This leads us to the question: when should we use each kind of smart pointer?
// std::unique_ptr - use it when you just want to hold one reference to
// the object.
// std::shared_ptr - use it when you want to hold multiple references to the
// same object and want to make sure that it's deallocated
// when all references are gone.
// std::weak_ptr - use it when you want to access
// the underlying object of a std::shared_ptr without causing that object to sta
y allocated.
// Weak pointers are used to prevent circular referencing.


//////////////////////
// Containers
//////////////////////

// Containers or the Standard Template Library are some predefined templates.
// They manage the storage space for its elements and provide
// member functions to access and manipulate them.

// Few containers are as follows:

// Vector (Dynamic array)
// Allow us to Define the Array or list of objects at run time
#include <vector>
string val;
vector<string> my_vector; // initialize the vector
cin >> val;
my_vector.push_back(val); // will push the value of 'val' into vector ("array")
my_vector
my_vector.push_back(val); // will push the value into the vector again (now havi
ng two elements)

// To iterate through a vector we have 2 choices:
// Either classic looping (iterating through the vector from index 0 to its last
 index):
for (int i = 0; i < my_vector.size(); i++) {
        cout << my_vector[i] << endl; // for accessing a vector's element we can
 use the operator []
}

// or using an iterator:
vector<string>::iterator it; // initialize the iterator for vector
for (it = my_vector.begin(); it != my_vector.end(); ++it) {
        cout << *it  << endl;
}

// Set
// Sets are containers that store unique elements following a specific order.
// Set is a very useful container to store unique values in sorted order
// without any other functions or code.

#include<set>
set<int> ST;    // Will initialize the set of int data type
ST.insert(30);  // Will insert the value 30 in set ST
ST.insert(10);  // Will insert the value 10 in set ST
ST.insert(20);  // Will insert the value 20 in set ST
ST.insert(30);  // Will insert the value 30 in set ST
// Now elements of sets are as follows
//  10 20 30

// To erase an element
ST.erase(20);  // Will erase element with value 20
// Set ST: 10 30
// To iterate through Set we use iterators
```



```
set<int>::iterator it;
for(it=ST.begin();it!=ST.end();it++) {
        cout << *it << endl;
}
// Output:
// 10
// 30

// To clear the complete container we use Container_name.clear()
ST.clear();
cout << ST.size();  // will print the size of set ST
// Output: 0

// NOTE: for duplicate elements we can use multiset
// NOTE: For hash sets, use unordered_set. They are more efficient but
// do not preserve order. unordered_set is available since C++11

// Map
// Maps store elements formed by a combination of a key value
// and a mapped value, following a specific order.

#include<map>
map<char, int> mymap;  // Will initialize the map with key as char and value as
int

mymap.insert(pair<char,int>('A',1));
// Will insert value 1 for key A
mymap.insert(pair<char,int>('Z',26));
// Will insert value 26 for key Z

// To iterate
map<char,int>::iterator it;
for (it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << "->" << it->second << std::cout;
// Output:
// A->1
// Z->26

// To find the value corresponding to a key
it = mymap.find('Z');
cout << it->second;

// Output: 26

// NOTE: For hash maps, use unordered_map. They are more efficient but do
// not preserve order. unordered_map is available since C++11.

// 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 avai
lable inside the function body and how.
// 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-c11

////////////////////////////////
// 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 contai
ner
// 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;


//////////////////////////////////////
// Tuples (C++11 and above)
//////////////////////////////////////

#include<tuple>

// Conceptually, Tuples are similar to  old data structures (C-like structs) but
 instead of having named data members,
// its elements are accessed by their order in the tuple.

// We start with constructing a tuple.
// Packing values into tuple
auto first = make_tuple(10, 'A');
const int maxN = 1e9;
const int maxL = 15;
auto second = make_tuple(maxN, maxL);
```

```
// Printing elements of 'first' tuple
cout << get<0>(first) << " " << get<1>(first) << '\n'; //prints : 10 A

// Printing elements of 'second' tuple
cout << get<0>(second) << " " << get<1>(second) << '\n'; // prints: 1000000000 1
5

// Unpacking tuple into variables

int first_int;
char first_char;
tie(first_int, first_char) = first;
cout << first_int << " " << first_char << '\n';  // prints : 10 A

// We can also create tuple like this.

tuple<int, char, double> third(11, 'A', 3.14141);
// tuple_size returns number of elements in a tuple (as a constexpr)

cout << tuple_size<decltype(third)>::value << '\n'; // prints: 3

// tuple_cat concatenates the elements of all the tuples in the same order.

auto concatenated_tuple = tuple_cat(first, second, third);
// concatenated_tuple becomes = (10, 'A', 1e9, 15, 11, 'A', 3.14141)

cout << get<0>(concatenated_tuple) << '\n'; // prints: 10
cout << get<3>(concatenated_tuple) << '\n'; // prints: 15
cout << get<5>(concatenated_tuple) << '\n'; // prints: 'A'


///////////////////////////////////
// Logical and Bitwise operators
///////////////////////////////////

// Most of the operators in C++ are same as in other languages

// Logical operators

// C++ uses Short-circuit evaluation for boolean expressions, i.e, the second ar
gument is executed or
// evaluated only if the first argument does not suffice to determine the value
of the expression

true && false // Performs **logical and** to yield false
true || false // Performs **logical or** to yield true
! true        // Performs **logical not** to yield false

// Instead of using symbols equivalent keywords can be used
true and false // Performs **logical and** to yield false
true or false  // Performs **logical or** to yield true
not true       // Performs **logical not** to yield false

// Bitwise operators

// **<<** Left Shift Operator
// << shifts bits to the left
4 << 1 // Shifts bits of 4 to left by 1 to give 8
// x << n can be thought as x * 2^n


// **>>** Right Shift Operator
// >> shifts bits to the right
4 >> 1 // Shifts bits of 4 to right by 1 to give 2
// x >> n can be thought as x / 2^n

~4    // Performs a bitwise not
4 | 3 // Performs bitwise or
```



```
4 & 3 // Performs bitwise and
4 ^ 3 // Performs bitwise xor

// Equivalent keywords are
compl 4    // Performs a bitwise not
4 bitor 3  // Performs bitwise or
4 bitand 3 // Performs bitwise and
4 xor 3    // Performs bitwise xor
// Single-line comments start with // - only available in C99 and later.

/*
Multi-line comments look like this. They work in C89 as well.
*/

/*
Multi-line comments don't nest /* Be careful */  // comment ends on this line...
*/ // ...not this one!

// Constants: #define <keyword>
// Constants are written in all-caps out of convention, not requirement
#define DAYS_IN_YEAR 365

// Enumeration constants are also ways to declare constants.
// All statements must end with a semicolon
enum days {SUN = 1, MON, TUE, WED, THU, FRI, SAT};
// MON gets 2 automatically, TUE gets 3, etc.

// Import headers with #include
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// (File names between <angle brackets> are headers from the C standard library.
)
// For your own headers, use double quotes instead of angle brackets:
//#include "my_header.h"

// Declare function signatures in advance in a .h file, or at the top of
// your .c file.
void function_1();
int function_2(void);

// Must declare a 'function prototype' before main() when functions occur after
// your main() function.
int add_two_ints(int x1, int x2); // function prototype
// although `int add_two_ints(int, int);` is also valid (no need to name the arg
s),
// it is recommended to name arguments in the prototype as well for easier inspe
ction

// Your program's entry point is a function called
// main with an integer return type.
int main(void) {
  // your program
}

// The command line arguments used to run your program are also passed to main
// argc being the number of arguments - your program's name counts as 1
// argv is an array of character arrays - containing the arguments themselves
// argv[0] = name of your program, argv[1] = first argument, etc.
int main (int argc, char** argv)
{
  // print output using printf, for "print formatted"
  // %d is an integer, \n is a newline
  printf("%d\n", 0); // => Prints 0

  ///////////////////////////////////////
  // Types
  ///////////////////////////////////////
```

```
  // Compilers that are not C99-compliant require that variables MUST be
  // declared at the top of the current block scope.
  // Compilers that ARE C99-compliant allow declarations near the point where
  // the value is used.
  // For the sake of the tutorial, variables are declared dynamically under
  // C99-compliant standards.

  // ints are usually 4 bytes
  int x_int = 0;

  // shorts are usually 2 bytes
  short x_short = 0;

  // chars are guaranteed to be 1 byte
  char x_char = 0;
  char y_char = 'y'; // Char literals are quoted with ''

  // longs are often 4 to 8 bytes; long longs are guaranteed to be at least
  // 8 bytes
  long x_long = 0;
  long long x_long_long = 0;

  // floats are usually 32-bit floating point numbers
  float x_float = 0.0f; // 'f' suffix here denotes floating point literal

  // doubles are usually 64-bit floating-point numbers
  double x_double = 0.0; // real numbers without any suffix are doubles

  // integer types may be unsigned (greater than or equal to zero)
  unsigned short ux_short;
  unsigned int ux_int;
  unsigned long long ux_long_long;

  // chars inside single quotes are integers in machine's character set.
  '0'; // => 48 in the ASCII character set.
  'A'; // => 65 in the ASCII character set.

  // sizeof(T) gives you the size of a variable with type T in bytes
  // sizeof(obj) yields the size of the expression (variable, literal, etc.).
  printf("%zu\n", sizeof(int)); // => 4 (on most machines with 4-byte words)

  // If the argument of the `sizeof` operator is an expression, then its argumen
t
  // is not evaluated (except VLAs (see below)).
  // The value it yields in this case is a compile-time constant.
  int a = 1;
  // size_t is an unsigned integer type of at least 2 bytes used to represent
  // the size of an object.
  size_t size = sizeof(a++); // a++ is not evaluated
  printf("sizeof(a++) = %zu where a = %d\n", size, a);
  // prints "sizeof(a++) = 4 where a = 1" (on a 32-bit architecture)

  // Arrays must be initialized with a concrete size.
  char my_char_array[20]; // This array occupies 1 * 20 = 20 bytes
  int my_int_array[20]; // This array occupies 4 * 20 = 80 bytes
  // (assuming 4-byte words)

  // You can initialize an array to 0 thusly:
  char my_array[20] = {0};
  // where the "{0}" part is called an "array initializer".
  // NOTE that you get away without explicitly declaring the size of the array,
  // IF you initialize the array on the same line. So, the following declaration
  // is equivalent:
  char my_array[] = {0};
  // BUT, then you have to evaluate the size of the array at run-time, like this
:
  size_t my_array_size = sizeof(my_array) / sizeof(my_array[0]);
  // WARNING If you adopt this approach, you should evaluate the size *before*
```



```
  // you begin passing the array to function (see later discussion), because
  // arrays get "downgraded" to raw pointers when they are passed to functions
  // (so the statement above will produce the wrong result inside the function).

  // Indexing an array is like other languages -- or,
  // rather, other languages are like C
  my_array[0]; // => 0

  // Arrays are mutable; it's just memory!
  my_array[1] = 2;
  printf("%d\n", my_array[1]); // => 2

  // In C99 (and as an optional feature in C11), variable-length arrays (VLAs)
  // can be declared as well. The size of such an array need not be a compile
  // time constant:
  printf("Enter the array size: "); // ask the user for an array size
  int array_size;
  fscanf(stdin, "%d", &array_size);
  int var_length_array[array_size]; // declare the VLA
  printf("sizeof array = %zu\n", sizeof var_length_array);

  // Example:
  // > Enter the array size: 10
  // > sizeof array = 40

  // Strings are just arrays of chars terminated by a NULL (0x00) byte,
  // represented in strings as the special character '\0'.
  // (We don't have to include the NULL byte in string literals; the compiler
  //  inserts it at the end of the array for us.)
  char a_string[20] = "This is a string";
  printf("%s\n", a_string); // %s formats a string

  printf("%d\n", a_string[16]); // => 0
  // i.e., byte #17 is 0 (as are 18, 19, and 20)

  // If we have characters between single quotes, that's a character literal.
  // It's of type `int`, and *not* `char` (for historical reasons).
  int cha = 'a'; // fine
  char chb = 'a'; // fine too (implicit conversion from int to char)

  // Multi-dimensional arrays:
  int multi_array[2][5] = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 0}
  };
  // access elements:
  int array_int = multi_array[0][2]; // => 3

  //////////////////////////////////////
  // Operators
  //////////////////////////////////////

  // Shorthands for multiple declarations:
  int i1 = 1, i2 = 2;
  float f1 = 1.0, f2 = 2.0;

  int b, c;
  b = c = 0;

  // Arithmetic is straightforward
  i1 + i2; // => 3
  i2 - i1; // => 1
  i2 * i1; // => 2
  i1 / i2; // => 0 (0.5, but truncated towards 0)

  // You need to cast at least one integer to float to get a floating-point resu
lt
  (float)i1 / i2; // => 0.5f
  i1 / (double)i2; // => 0.5 // Same with double
```

```
  f1 / f2; // => 0.5, plus or minus epsilon
  // Floating-point numbers and calculations are not exact

  // Modulo is there as well
  11 % 3; // => 2

  // Comparison operators are probably familiar, but
  // there is no Boolean type in C. We use ints instead.
  // (Or _Bool or bool in C99.)
  // 0 is false, anything else is true. (The comparison
  // operators always yield 0 or 1.)
  3 == 2; // => 0 (false)
  3 != 2; // => 1 (true)
  3 > 2; // => 1
  3 < 2; // => 0
  2 <= 2; // => 1
  2 >= 2; // => 1

  // C is not Python - comparisons don't chain.
  // Warning: The line below will compile, but it means `(0 < a) < 2`.
  // This expression is always true, because (0 < a) could be either 1 or 0.
  // In this case it's 1, because (0 < 1).
  int between_0_and_2 = 0 < a < 2;
  // Instead use:
  int between_0_and_2 = 0 < a && a < 2;

  // Logic works on ints
  !3; // => 0 (Logical not)
  !0; // => 1
  1 && 1; // => 1 (Logical and)
  0 && 1; // => 0
  0 || 1; // => 1 (Logical or)
  0 || 0; // => 0

  // Conditional ternary expression ( ? : )
  int e = 5;
  int f = 10;
  int z;
  z = (e > f) ? e : f; // => 10 "if e > f return e, else return f."

  // Increment and decrement operators:
  int j = 0;
  int s = j++; // Return j THEN increase j. (s = 0, j = 1)
  s = ++j; // Increase j THEN return j. (s = 2, j = 2)
  // same with j-- and --j

  // Bitwise operators!
  ~0x0F; // => 0xFFFFFFF0 (bitwise negation, "1's complement", example result fo
r 32-bit int)
  0x0F & 0xF0; // => 0x00 (bitwise AND)
  0x0F | 0xF0; // => 0xFF (bitwise OR)
  0x04 ^ 0x0F; // => 0x0B (bitwise XOR)
  0x01 << 1; // => 0x02 (bitwise left shift (by 1))
  0x02 >> 1; // => 0x01 (bitwise right shift (by 1))

  // Be careful when shifting signed integers - the following are undefined:
  // - shifting into the sign bit of a signed integer (int a = 1 << 31)
  // - left-shifting a negative number (int a = -1 << 2)
  // - shifting by an offset which is >= the width of the type of the LHS:
  //   int a = 1 << 32; // UB if int is 32 bits wide

  ///////////////////////////////////////
  // Control Structures
  ///////////////////////////////////////

  if (0) {
    printf("I am never run\n");
  } else if (0) {
    printf("I am also never run\n");
```



```
  } else {
    printf("I print\n");
  }

  // While loops exist
  int ii = 0;
  while (ii < 10) { //ANY value less than ten is true.
    printf("%d, ", ii++); // ii++ increments ii AFTER using its current value.
  } // => prints "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

  printf("\n");

  int kk = 0;
  do {
    printf("%d, ", kk);
  } while (++kk < 10); // ++kk increments kk BEFORE using its current value.
  // => prints "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

  printf("\n");

  // For loops too
  int jj;
  for (jj=0; jj < 10; jj++) {
    printf("%d, ", jj);
  } // => prints "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

  printf("\n");

  // *****NOTES*****:
  // Loops and Functions MUST have a body. If no body is needed:
  int i;
  for (i = 0; i <= 5; i++) {
    ; // use semicolon to act as the body (null statement)
  }
  // Or
  for (i = 0; i <= 5; i++);

  // branching with multiple choices: switch()
  switch (a) {
  case 0: // labels need to be integral *constant* expressions (such as enums)
    printf("Hey, 'a' equals 0!\n");
    break; // if you don't break, control flow falls over labels
  case 1:
    printf("Huh, 'a' equals 1!\n");
    break;
    // Be careful - without a "break", execution continues until the
    // next "break" is reached.
  case 3:
  case 4:
    printf("Look at that.. 'a' is either 3, or 4\n");
    break;
  default:
    // if `some_integral_expression` didn't match any of the labels
    fputs("Error!\n", stderr);
    exit(-1);
    break;
  }
  /*
  using "goto" in C
  */
  typedef enum { false, true } bool;
  // for C don't have bool as data type before C99 :(
  bool disaster = false;
  int i, j;
  for(i=0;i<100;++i)
  for(j=0;j<100;++j)
  {
    if((i + j) >= 150)
        disaster = true;
```

```
    if(disaster)
        goto error;
  }
  error :
  printf("Error occurred at i = %d & j = %d.\n", i, j);
  /*
  https://ideone.com/GuPhd6
  this will print out "Error occurred at i = 51 & j = 99."
  */

  ////////////////////////////////////
  // Typecasting
  ////////////////////////////////////

  // Every value in C has a type, but you can cast one value into another type
  // if you want (with some constraints).

  int x_hex = 0x01; // You can assign vars with hex literals

  // Casting between types will attempt to preserve their numeric values
  printf("%d\n", x_hex); // => Prints 1
  printf("%d\n", (short) x_hex); // => Prints 1
  printf("%d\n", (char) x_hex); // => Prints 1

  // Types will overflow without warning
  printf("%d\n", (unsigned char) 257); // => 1 (Max char = 255 if char is 8 bits
 long)

  // For determining the max value of a `char`, a `signed char` and an `unsigned
 char`,
  // respectively, use the CHAR_MAX, SCHAR_MAX and UCHAR_MAX macros from <limits
.h>

  // Integral types can be cast to floating-point types, and vice-versa.
  printf("%f\n", (double) 100); // %f always formats a double...
  printf("%f\n", (float)  100); // ...even with a float.
  printf("%d\n", (char)100.0);

  ////////////////////////////////////
  // Pointers
  ////////////////////////////////////

  // A pointer is a variable declared to store a memory address. Its declaration
 will
  // also tell you the type of data it points to. You can retrieve the memory ad
dress
  // of your variables, then mess with them.

  int x = 0;
  printf("%p\n", (void *)&x); // Use & to retrieve the address of a variable
  // (%p formats an object pointer of type void *)
  // => Prints some address in memory;

  // Pointers start with * in their declaration
  int *px, not_a_pointer; // px is a pointer to an int
  px = &x; // Stores the address of x in px
  printf("%p\n", (void *)px); // => Prints some address in memory
  printf("%zu, %zu\n", sizeof(px), sizeof(not_a_pointer));
  // => Prints "8, 4" on a typical 64-bit system

  // To retrieve the value at the address a pointer is pointing to,
  // put * in front to dereference it.
  // Note: yes, it may be confusing that '*' is used for _both_ declaring a
  // pointer and dereferencing it.
  printf("%d\n", *px); // => Prints 0, the value of x

  // You can also change the value the pointer is pointing to.
  // We'll have to wrap the dereference in parenthesis because
  // ++ has a higher precedence than *.
```



```
  (*px)++; // Increment the value px is pointing to by 1
  printf("%d\n", *px); // => Prints 1
  printf("%d\n", x); // => Prints 1

  // Arrays are a good way to allocate a contiguous block of memory
  int x_array[20]; //declares array of size 20 (cannot change size)
  int xx;
  for (xx = 0; xx < 20; xx++) {
    x_array[xx] = 20 - xx;
  } // Initialize x_array to 20, 19, 18,... 2, 1

  // Declare a pointer of type int and initialize it to point to x_array
  int* x_ptr = x_array;
  // x_ptr now points to the first element in the array (the integer 20).
  // This works because arrays often decay into pointers to their first element.
  // For example, when an array is passed to a function or is assigned to a poin
ter,
  // it decays into (implicitly converted to) a pointer.
  // Exceptions: when the array is the argument of the `&` (address-of) operator
:
  int arr[10];
  int (*ptr_to_arr)[10] = &arr; // &arr is NOT of type `int *`!
  // It's of type "pointer to array" (of ten `int`s).
  // or when the array is a string literal used for initializing a char array:
  char otherarr[] = "foobarbazquirk";
  // or when it's the argument of the `sizeof` or `alignof` operator:
  int arraythethird[10];
  int *ptr = arraythethird; // equivalent with int *ptr = &arr[0];
  printf("%zu, %zu\n", sizeof(arraythethird), sizeof(ptr));
  // probably prints "40, 4" or "40, 8"

  // Pointers are incremented and decremented based on their type
  // (this is called pointer arithmetic)
  printf("%d\n", *(x_ptr + 1)); // => Prints 19
  printf("%d\n", x_array[1]); // => Prints 19

  // You can also dynamically allocate contiguous blocks of memory with the
  // standard library function malloc, which takes one argument of type size_t
  // representing the number of bytes to allocate (usually from the heap, althou
gh this
  // may not be true on e.g. embedded systems - the C standard says nothing abou
t it).
  int *my_ptr = malloc(sizeof(*my_ptr) * 20);
  for (xx = 0; xx < 20; xx++) {
    *(my_ptr + xx) = 20 - xx; // my_ptr[xx] = 20-xx
  } // Initialize memory to 20, 19, 18, 17... 2, 1 (as ints)

  // Be careful passing user-provided values to malloc! If you want
  // to be safe, you can use calloc instead (which, unlike malloc, also zeros ou
t the memory)
  int* my_other_ptr = calloc(20, sizeof(int));

  // Note that there is no standard way to get the length of a
  // dynamically allocated array in C. Because of this, if your arrays are
  // going to be passed around your program a lot, you need another variable
  // to keep track of the number of elements (size) of an array. See the
  // functions section for more info.
  size_t size = 10;
  int *my_arr = calloc(size, sizeof(int));
  // Add an element to the array
  size++;
  my_arr = realloc(my_arr, sizeof(int) * size);
  if (my_arr == NULL) {
    //Remember to check for realloc failure!
    return
  }
  my_arr[10] = 5;

  // Dereferencing memory that you haven't allocated gives
```

```
  // "unpredictable results" - the program is said to invoke "undefined behavior
"
  printf("%d\n", *(my_ptr + 21)); // => Prints who-knows-what? It may even crash
.

  // When you're done with a malloc'd block of memory, you need to free it,
  // or else no one else can use it until your program terminates
  // (this is called a "memory leak"):
  free(my_ptr);

  // Strings are arrays of char, but they are usually represented as a
  // pointer-to-char (which is a pointer to the first element of the array).
  // It's good practice to use `const char *' when referring to a string literal
,
  // since string literals shall not be modified (i.e. "foo"[0] = 'a' is ILLEGAL
.)
  const char *my_str = "This is my very own string literal";
  printf("%c\n", *my_str); // => 'T'

  // This is not the case if the string is an array
  // (potentially initialized with a string literal)
  // that resides in writable memory, as in:
  char foo[] = "foo";
  foo[0] = 'a'; // this is legal, foo now contains "aoo"

  function_1();
} // end main function

///////////////////////////////////////
// Functions
///////////////////////////////////////

// Function declaration syntax:
// <return type> <function name>(<args>)

int add_two_ints(int x1, int x2)
{
  return x1 + x2; // Use return to return a value
}

/*
Functions are call by value. When a function is called, the arguments passed to
the function are copies of the original arguments (except arrays). Anything you
do to the arguments in the function do not change the value of the original
argument where the function was called.

Use pointers if you need to edit the original argument values.

Example: in-place string reversal
*/

// A void function returns no value
void str_reverse(char *str_in)
{
  char tmp;
  size_t ii = 0;
  size_t len = strlen(str_in); // `strlen()` is part of the c standard library
                               // NOTE: length returned by `strlen` DOESN'T incl
ude the
                               //       terminating NULL byte ('\0')
  for (ii = 0; ii < len / 2; ii++) { // in C99 you can directly declare type of
`ii` here
    tmp = str_in[ii];
    str_in[ii] = str_in[len - ii - 1]; // ii-th char from end
    str_in[len - ii - 1] = tmp;
  }
}
//NOTE: string.h header file needs to be included to use strlen()
```



```
/*
char c[] = "This is a test.";
str_reverse(c);
printf("%s\n", c); // => ".tset a si sihT"
*/
/*
as we can return only one variable
to change values of more than one variables we use call by reference
*/
void swapTwoNumbers(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
/*
int first = 10;
int second = 20;
printf("first: %d\nsecond: %d\n", first, second);
swapTwoNumbers(&first, &second);
printf("first: %d\nsecond: %d\n", first, second);
// values will be swapped
*/

/*
With regards to arrays, they will always be passed to functions
as pointers. Even if you statically allocate an array like `arr[10]`,
it still gets passed as a pointer to the first element in any function calls.
Again, there is no standard way to get the size of a dynamically allocated
array in C.
*/
// Size must be passed!
// Otherwise, this function has no way of knowing how big the array is.
void printIntArray(int *arr, size_t size) {
    int i;
    for (i = 0; i < size; i++) {
        printf("arr[%d] is: %d\n", i, arr[i]);
    }
}
/*
int my_arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int size = 10;
printIntArray(my_arr, size);
// will print "arr[0] is: 1" etc
*/

// if referring to external variables outside function, you should use the exter
n keyword.
int i = 0;
void testFunc() {
  extern int i; //i here is now using external variable i
}

// make external variables private to source file with static:
static int j = 0; //other files using testFunc2() cannot access variable j
void testFunc2() {
  extern int j;
}
// The static keyword makes a variable inaccessible to code outside the
// compilation unit. (On almost all systems, a "compilation unit" is a .c
// file.) static can apply both to global (to the compilation unit) variables,
// functions, and function-local variables. When using static with
// function-local variables, the variable is effectively global and retains its
// value across function calls, but is only accessible within the function it
// is declared in. Additionally, static variables are initialized to 0 if not
// declared with some other starting value.
//**You may also declare functions as static to make them private**

///////////////////////////////////////
```

```
// User-defined types and structs
//////////////////////////////////////

// Typedefs can be used to create type aliases
typedef int my_type;
my_type my_type_var = 0;

// Structs are just collections of data, the members are allocated sequentially,
// in the order they are written:
struct rectangle {
  int width;
  int height;
};

// It's not generally true that
// sizeof(struct rectangle) == sizeof(int) + sizeof(int)
// due to potential padding between the structure members (this is for alignment
// reasons). [1]

void function_1()
{
  struct rectangle my_rec;

  // Access struct members with .
  my_rec.width = 10;
  my_rec.height = 20;

  // You can declare pointers to structs
  struct rectangle *my_rec_ptr = &my_rec;

  // Use dereferencing to set struct pointer members...
  (*my_rec_ptr).width = 30;

  // ... or even better: prefer the -> shorthand for the sake of readability
  my_rec_ptr->height = 10; // Same as (*my_rec_ptr).height = 10;
}

// You can apply a typedef to a struct for convenience
typedef struct rectangle rect;

int area(rect r)
{
  return r.width * r.height;
}

// if you have large structs, you can pass them "by pointer" to avoid copying
// the whole struct:
int areaptr(const rect *r)
{
  return r->width * r->height;
}

//////////////////////////////////////
// Function pointers
//////////////////////////////////////
/*
At run time, functions are located at known memory addresses. Function pointers
are
much like any other pointer (they just store a memory address), but can be used
to invoke functions directly, and to pass handlers (or callback functions) aroun
d.
However, definition syntax may be initially confusing.

Example: use str_reverse from a pointer
*/
void str_reverse_through_pointer(char *str_in) {
  // Define a function pointer variable, named f.
  void (*f)(char *); // Signature should exactly match the target function.
  f = &str_reverse; // Assign the address for the actual function (determined at
```



```
 run time)
  // f = str_reverse; would work as well - functions decay into pointers, simila
r to arrays
  (*f)(str_in); // Just calling the function through the pointer
  // f(str_in); // That's an alternative but equally valid syntax for calling it
.
}

/*
As long as function signatures match, you can assign any function to the same po
inter.
Function pointers are usually typedef'd for simplicity and readability, as follo
ws:
*/

typedef void (*my_fnp_type)(char *);

// Then used when declaring the actual pointer variable:
// ...
// my_fnp_type f;


//Special characters:
/*
'\a'; // alert (bell) character
'\n'; // newline character
'\t'; // tab character (left justifies text)
'\v'; // vertical tab
'\f'; // new page (form feed)
'\r'; // carriage return
'\b'; // backspace character
'\0'; // NULL character. Usually put at end of strings in C.
//   hello\n\0. \0 used by convention to mark end of string.
'\\'; // backslash
'\?'; // question mark
'\''; // single quote
'\"'; // double quote
'\xhh'; // hexadecimal number. Example: '\xb' = vertical tab character
'\0oo'; // octal number. Example: '\013' = vertical tab character

//print formatting:
"%d";    // integer
"%3d";   // integer with minimum of length 3 digits (right justifies text)
"%s";    // string
"%f";    // float
"%ld";   // long
"%3.2f"; // minimum 3 digits left and 2 digits right decimal float
"%7.4s"; // (can do with strings too)
"%c";    // char
"%p";    // pointer. NOTE: need to (void *)-cast the pointer, before passing
         //                it as an argument to `printf`.
"%x";    // hexadecimal
"%o";    // octal
"%%";    // prints %
*/

//////////////////////////////////////
// Order of Evaluation
//////////////////////////////////////

//---------------------------------------------------//
//        Operators                  | Associativity //
//---------------------------------------------------//
// () [] -> .                        | left to right //
// ! ~ ++ -- + = *(type)sizeof       | right to left //
// * / %                             | left to right //
// + -                               | left to right //
// << >>                             | left to right //
// < <= > >=                         | left to right //
```

```
// == !=                                  | left to right //
// &                                      | left to right //
// ^                                      | left to right //
// |                                      | left to right //
// &&                                     | left to right //
// ||                                     | left to right //
// ?:                                     | right to left //
// = += -= *= /= %= &= ^= |= <<= >>=     | right to left //
// ,                                      | left to right //
//-------------------------------------------------------//

/****************************** Header Files ********************************

Header files are an important part of C as they allow for the connection of C
source files and can simplify code and definitions by separating them into
separate files.

Header files are syntactically similar to C source files but reside in ".h"
files. They can be included in your C source file by using the precompiler
command #include "example.h", given that example.h exists in the same directory
as the C file.
*/

/* A safe guard to prevent the header from being defined too many times. This */
/* happens in the case of circle dependency, the contents of the header is    */
/* already defined.                                                           */
#ifndef EXAMPLE_H /* if EXAMPLE_H is not yet defined. */
#define EXAMPLE_H /* Define the macro EXAMPLE_H. */

/* Other headers can be included in headers and therefore transitively */
/* included into files that include this header.                       */
#include <string.h>

/* Like c source files macros can be defined in headers and used in files */
/* that include this header file.                                         */
#define EXAMPLE_NAME "Dennis Ritchie"

/* Function macros can also be defined.  */
#define ADD(a, b) ((a) + (b))

/* Notice the parenthesis surrounding the arguments -- this is important to   */
/* ensure that a and b don't get expanded in an unexpected way (e.g. consider */
/* MUL(x, y) (x * y); MUL(1 + 2, 3) would expand to (1 + 2 * 3), yielding an  */
/* incorrect result)                                                          */

/* Structs and typedefs can be used for consistency between files. */
typedef struct Node
{
    int val;
    struct Node *next;
} Node;

/* So can enumerations. */
enum traffic_light_state {GREEN, YELLOW, RED};

/* Function prototypes can also be defined here for use in multiple files,  */
/* but it is bad practice to define the function in the header. Definitions */
/* should instead be put in a C file.                                       */
Node createLinkedList(int *vals, int len);

/* Beyond the above elements, other definitions should be left to a C source */
/* file. Excessive includes or definitions should, also not be contained in */
/* a header file but instead put into separate headers or a C file.          */

#endif /* End of the if precompiler directive. */
//////////////////
// 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

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

// C standard headers are available in C++.
// C headers end in .h, while
// C++ headers are prefixed with "c" and have no ".h" suffix.

// The C++ standard version:
#include <cstdio>

// The C standard version:
#include <stdio.h>

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.
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 enu
m type
        WriteByteToFile(InputCarType);
}

// On the other hand you may not want enums to be accidentally cast to an intege
r
// type or to other enums so it is instead possible to create an enum class whic
h
// 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:

    // 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()
{
    std::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 {

public:
    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);
}

// It's good practice to return a reference to the leftmost variable of
// an assignment. `(a += b) == c` will work this way.
Point& Point::operator+=(const Point& rhs)
{
    x += rhs.x;
    y += rhs.y;

    // `this` is a pointer to the object, on which a method is called.
    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 as an 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
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.
    }
    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 shared_ptrread 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


////////////////////
// Smart Pointer
////////////////////

// Generally a smart pointer is a class which wraps a "raw pointer" (usage of "n
ew"
// respectively malloc/calloc in C). The goal is to be able to
// manage the lifetime of the object being pointed to without ever needing to ex
plicitly delete
// the object. The term itself simply describes a set of pointers with the
// mentioned abstraction.
// Smart pointers should preferred over raw pointers, to prevent
// risky memory leaks, which happen if you forget to delete an object.

// Usage of a raw pointer:
Dog* ptr = new Dog();
ptr->bark();
delete ptr;

// By using a smart pointer, you don't have to worry about the deletion
// of the object anymore.
// A smart pointer describes a policy, to count the references to the
// pointer. The object gets destroyed when the last
// reference to the object gets destroyed.

// Usage of "std::shared_ptr":
void foo()
{
// It's no longer necessary to delete the Dog.
std::shared_ptr<Dog> doggo(new Dog());
doggo->bark();
}

// Beware of possible circular references!!!
// There will be always a reference, so it will be never destroyed!
std::shared_ptr<Dog> doggo_one(new Dog());
std::shared_ptr<Dog> doggo_two(new Dog());
doggo_one = doggo_two; // p1 references p2
doggo_two = doggo_one; // p2 references p1

// There are several kinds of smart pointers.
// The way you have to use them is always the same.
```



```
// This leads us to the question: when should we use each kind of smart pointer?
// std::unique_ptr - use it when you just want to hold one reference to
// the object.
// std::shared_ptr - use it when you want to hold multiple references to the
// same object and want to make sure that it's deallocated
// when all references are gone.
// std::weak_ptr - use it when you want to access
// the underlying object of a std::shared_ptr without causing that object to sta
y allocated.
// Weak pointers are used to prevent circular referencing.


////////////////////
// Containers
////////////////////

// Containers or the Standard Template Library are some predefined templates.
// They manage the storage space for its elements and provide
// member functions to access and manipulate them.

// Few containers are as follows:

// Vector (Dynamic array)
// Allow us to Define the Array or list of objects at run time
#include <vector>
string val;
vector<string> my_vector; // initialize the vector
cin >> val;
my_vector.push_back(val); // will push the value of 'val' into vector ("array")
my_vector
my_vector.push_back(val); // will push the value into the vector again (now havi
ng two elements)

// To iterate through a vector we have 2 choices:
// Either classic looping (iterating through the vector from index 0 to its last
 index):
for (int i = 0; i < my_vector.size(); i++) {
        cout << my_vector[i] << endl; // for accessing a vector's element we can
 use the operator []
}

// or using an iterator:
vector<string>::iterator it; // initialize the iterator for vector
for (it = my_vector.begin(); it != my_vector.end(); ++it) {
        cout << *it  << endl;
}

// Set
// Sets are containers that store unique elements following a specific order.
// Set is a very useful container to store unique values in sorted order
// without any other functions or code.

#include<set>
set<int> ST;    // Will initialize the set of int data type
ST.insert(30);  // Will insert the value 30 in set ST
ST.insert(10);  // Will insert the value 10 in set ST
ST.insert(20);  // Will insert the value 20 in set ST
ST.insert(30);  // Will insert the value 30 in set ST
// Now elements of sets are as follows
//  10 20 30

// To erase an element
ST.erase(20);  // Will erase element with value 20
// Set ST: 10 30
// To iterate through Set we use iterators
set<int>::iterator it;
for(it=ST.begin();it!=ST.end();it++) {
        cout << *it << endl;
}
```

```
// Output:
// 10
// 30

// To clear the complete container we use Container_name.clear()
ST.clear();
cout << ST.size();  // will print the size of set ST
// Output: 0

// NOTE: for duplicate elements we can use multiset
// NOTE: For hash sets, use unordered_set. They are more efficient but
// do not preserve order. unordered_set is available since C++11

// Map
// Maps store elements formed by a combination of a key value
// and a mapped value, following a specific order.

#include<map>
map<char, int> mymap;  // Will initialize the map with key as char and value as
int

mymap.insert(pair<char,int>('A',1));
// Will insert value 1 for key A
mymap.insert(pair<char,int>('Z',26));
// Will insert value 26 for key Z

// To iterate
map<char,int>::iterator it;
for (it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << "->" << it->second << std::cout;
// Output:
// A->1
// Z->26

// To find the value corresponding to a key
it = mymap.find('Z');
cout << it->second;

// Output: 26

// NOTE: For hash maps, use unordered_map. They are more efficient but do
// not preserve order. unordered_map is available since C++11.

// 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 avai
lable inside the function body and how.
// 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-c11

////////////////////////////////
// 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 contai
ner
// 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;


///////////////////////////////////////
// Tuples (C++11 and above)
///////////////////////////////////////

#include<tuple>

// Conceptually, Tuples are similar to  old data structures (C-like structs) but
 instead of having named data members,
// its elements are accessed by their order in the tuple.

// We start with constructing a tuple.
// Packing values into tuple
auto first = make_tuple(10, 'A');
const int maxN = 1e9;
const int maxL = 15;
auto second = make_tuple(maxN, maxL);

// Printing elements of 'first' tuple
cout << get<0>(first) << " " << get<1>(first) << '\n'; //prints : 10 A
```



```
// Printing elements of 'second' tuple
cout << get<0>(second) << " " << get<1>(second) << '\n'; // prints: 1000000000 1
5

// Unpacking tuple into variables

int first_int;
char first_char;
tie(first_int, first_char) = first;
cout << first_int << " " << first_char << '\n';  // prints : 10 A

// We can also create tuple like this.

tuple<int, char, double> third(11, 'A', 3.14141);
// tuple_size returns number of elements in a tuple (as a constexpr)

cout << tuple_size<decltype(third)>::value << '\n'; // prints: 3

// tuple_cat concatenates the elements of all the tuples in the same order.

auto concatenated_tuple = tuple_cat(first, second, third);
// concatenated_tuple becomes = (10, 'A', 1e9, 15, 11, 'A', 3.14141)

cout << get<0>(concatenated_tuple) << '\n'; // prints: 10
cout << get<3>(concatenated_tuple) << '\n'; // prints: 15
cout << get<5>(concatenated_tuple) << '\n'; // prints: 'A'


///////////////////////////////////
// Logical and Bitwise operators
///////////////////////////////////

// Most of the operators in C++ are same as in other languages

// Logical operators

// C++ uses Short-circuit evaluation for boolean expressions, i.e, the second ar
gument is executed or
// evaluated only if the first argument does not suffice to determine the value
of the expression

true && false // Performs **logical and** to yield false
true || false // Performs **logical or** to yield true
! true        // Performs **logical not** to yield false

// Instead of using symbols equivalent keywords can be used
true and false // Performs **logical and** to yield false
true or false  // Performs **logical or** to yield true
not true       // Performs **logical not** to yield false

// Bitwise operators

// **<<** Left Shift Operator
// << shifts bits to the left
4 << 1 // Shifts bits of 4 to left by 1 to give 8
// x << n can be thought as x * 2^n


// **>>** Right Shift Operator
// >> shifts bits to the right
4 >> 1 // Shifts bits of 4 to right by 1 to give 2
// x >> n can be thought as x / 2^n

~4    // Performs a bitwise not
4 | 3 // Performs bitwise or
4 & 3 // Performs bitwise and
4 ^ 3 // Performs bitwise xor

// Equivalent keywords are
```

```
compl 4    // Performs a bitwise not
4 bitor 3  // Performs bitwise or
4 bitand 3 // Performs bitwise and
4 xor 3    // Performs bitwise xor
```