CIS 330: UNIX and C/C++
(Re)Introduction to C++

Prof. Kevin Butler
Winter 2012
Today’s goals

• An introduction to C++
  ‣ some shortcomings of C that C++ addresses
  ‣ give perspective on how to learn C++
  ‣ kick the tires and see some code
We had to work hard to mimic encapsulation, abstraction

- **encapsulation**: hiding implementation details
  - used header file conventions and the “static” specifier to separate private functions from public functions
  - cast structures to (void *) to hide implementation-specific details

- **abstraction**: associating behavior with encapsulated state
  - the functions that operate on a LinkedList were not really tied to the linked list structure
  - we passed a linked list to a function, rather than invoking a method on a linked list instance
• A major addition is its support for object-orientedness!
  ‣ classes
    • public, private, and protected methods and instance variables
    • (multiple!) inheritance
  ‣ polymorphism
    • static polymorphism: multiple functions or methods with the same name, but different argument types
    • dynamic polymorphism: derived classes can overload of methods of parents, and methods will be dispatched correctly
• We had to emulate generic data structures
  ▸ customer passes a (void *) as a payload to a linked list
  ▸ customer had to pass in function pointers so that the linked list could operate on payloads correctly
    • comparisons, deallocation, pickling up state, etc.
• Supports **templates** to facilitate generic data types!

  ‣ to declare that `x` is a vector of ints:
    * `vector<int> x;`
  
  ‣ to declare that `x` is a vector of floats:
    * `vector<float> x;`
  
  ‣ to declare that `x` is a vector of (vectors of floats):
    * `vector<vector<float>> x;`
• We had to be careful about namespace collisions
  ‣ C distinguishes between external and internal linkage
    • use “static” to prevent a name from being visible outside a module
    • otherwise, a name is global -- visible everywhere
  ‣ we used naming conventions to help avoid collisions in the global namespace
    • LLIteratorNext, HTIteratorNext, etc.
• Permits a module to define its own namespace!
  ‣ the linked list module could define an “LL” namespace
  ‣ the hashtable module could define an “HT” namespace
  ‣ both modules could define an Iterator class
    • one would be globally named `LL::Iterator`
    • the other would be globally named `HT::Iterator`
• C does not provide any standard data structures
  ‣ we had to implement our own linked list and hash table
  ‣ as a C programmer, you often re-invent the wheel badly
    • maybe if you’re clever you’ll use somebody else’s libraries
    • but, C’s lack of abstraction, encapsulation, and generics means you’ll probably have to tweak them, or tweak your code to use them
• The C++ standard library is rich!

  ‣ **generic containers**: bitset, queue, list, associative array (including hash table), queue, set, stack, and vector

  • and, iterators for most of these

  ‣ **a string class**: hides the implementation of strings

  ‣ **streams**: allows you to stream data to and from objects, consoles, files, strings, and so on

  • and more...
• Error handling is a pain
  ▸ have to define error codes and return them
  ▸ customers have to understand error code conventions, and need to constantly test return values
  ▸ if \( a( ) \) calls \( b( ) \) calls \( c( ) \)
    • \( a \) depends on \( b \) to propagate an error in \( c \) back to it
• Supports exceptions!
  ‣ try / throw / catch

• if used with discipline, can simplify error processing
  ‣ but, if used carelessly, can complicate memory management
  ‣ consider: a() calls b() calls c()
    • if c() throws an exception that b() doesn’t catch, you might not get a chance to clean up resources allocated inside b()
Some tasks still hurt in C++

• Memory management
  ‣ C++ has no garbage collector
    • you have to manage memory allocation and deallocation, and track ownership of memory
    • it’s still possible to have leaks, double frees, and so on
  ‣ but, there are some things that help
    • “smart pointers”
      ‣ classes that encapsulate pointers and track reference counts
      ‣ deallocate memory when the reference count goes to zero
Some tasks still hurt in C++

• C++ doesn’t guarantee type or memory safety
  ‣ You can still...
    • forcibly cast pointers between incompatible types
    • walk off the end of an array and smash the stack
    • have dangling pointers
    • conjure up a pointer to an address of your choosing
C++ has many, many features

- Operator overloading
  - your class can define methods for handling “+”, “->”, etc!
- Object constructors, destructors
  - particularly handy for stack-allocated objects
- Reference types
  - truly pass-by-reference instead of pass-by-value
- Advanced OO
  - multiple inheritance, virtual base classes, dynamic dispatch
How to think about C++

set of styles and ways to use C++

good styles and robust engineering practices

style guides

set of styles and ways to use C
Another Analogy

• C++ can be a powerful weapon in your arsenal

• But use it wrong and it can backfire on you
Hello, world!

• Looks simple enough...
  ‣ compile with g++ instead of gcc:
    • g++ -Wall -o helloworld helloworld.cc
  ‣ let’s walk through the program step by step

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

helloworld.cc
Hello, world!

- `iostream.h` is part of the C++ standard library
  - note you don’t need to include the “.h” when you include standard library headers
    - but you do for local headers (e.g., `#include "ll.h"`)
  - `iostream` declares stream object instances, including `std::cin`, `std::cout`, `std::cerr`, in the “std” namespace
Hello, world!

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

- **cstdlib** is the C standard library’s stdlib.h header
  - all C standard library functions are available to you
- for header `foo.h`, you should `#include <cfoo>`
  - we need it for EXIT_SUCCESS, as usual
Hello, world!

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

• `std::cout` is the “cout” object instance declared by `iostream.h`, living within the “std” namespace
  ‣ `std::cout` is an object of class `ostream`
  ‣ used to format and write output to the console
Hello, world!

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

- C++ distinguishes between objects and primitive types
  - primitive types include all the familiar ones from C
    - char, short, unsigned long, float, double, long double, etc.
    - and, C++ defines “bool” as a primitive type (woohoo!)
Hello, world!

```c++
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

- “<<” is an operator defined by the C++ language
  - it’s defined by C as well; in C/C++, it bitshifts integers
  - but, C++ allows **classes** to overload operators
    - the ostream class overloads “<<”
    - i.e., it defines methods that are invoked when an ostream is the LHS of the << operator
Hello, world!

- `#include <iostream>`
- `#include <cstdlib>`

```cpp
int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

- `ostream` has many different methods to handle `<<`
  - The methods differ in the type of the RHS of `<<`
  - If you do `std::cout << "foo";`
    - C++ invokes `cout`'s method to handle `"<<"` with RHS `"char *"`
Hello, world!

helloworld.cc

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

- the ostream class’s methods that handle “<<” return (a reference to) themselves
  - so, when (std::cout << “Hello, World!”) is evaluated:
    - a method of the std::cout object is invoked
    - it buffers the string “Hello, World!” for the console
    - and, it returns (a reference to) std::cout
Hello, world!

```
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

• next, a method on std::cout to handle “<<” is invoked
  ‣ this time, the RHS is `std::endl`
  ‣ turns out this is a pointer to a “manipulator” function
    • this manipulator function writes newline to the ostream it is invoked on, and then flushes the ostream’s buffer
    • so, something is printed on the console at this point
Complexity of C++

You should be surprised and scared at this point

- C++ makes it easy to hide a significant amount of complexity
  - it’s powerful, but really dangerous
  - once you mix together templates, operator overloading, method overloading, generics, and multiple inheritance, and it gets really hard to know what’s actually happening!
Refining it a bit...

**C++’s standard library has a `std::string` class!**

- include the string.h header to use it
Refining it a bit...

• The “using” keyword introduces part of a namespace, or an entire namespace, into the current region

  ‣ using namespace std;  -- imports all names from std::
  ‣ using std::cout;  -- imports only std::cout

```cpp
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;

int main(int argc, char **argv) {
    string hello("Hello, World!");
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```
• We’re instantiating a `std::string` object on the stack
  ‣ passing the C string “Hello, World!” to its constructor method
    • “hello” is deallocated (and its destructor invoked) when `main` returns
Refining it a bit...

• The C++ string library overloads the `<<` operator as well
  ‣ defines a function (not an object method) that is invoked when the LHS is an ostream and the RHS is a `std::string`

```c++
#include <iostream>
#include <cstdlib>
#include <string>

using namespace std;

int main(int argc, char **argv) {
    string hello("Hello, World!");
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```
Refining it a bit...

```cpp
#include <iostream>
#include <cstdlib>
#include <string>

using namespace std;

int main(int argc, char **argv) {
    string hello("Hello, World!");
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```

• Note the side-effect of `using namespace std;`
  ‣ can now refer to `std::string` by `string`, `std::cout` by `cout`, and `std::endl` by `endl`
The string class overloads the “+” operator

- creates and returns a new string that is the concatenation of LHS and RHS
string assignment

- The string class overloads the “=” operator
  - copies the RHS and replaces the string’s contents with it
    - so, the full statement (a) “+” creates a string that is the concatenation of hello’s current contents and “ there”, and (b) “=” creates a copy of the concatenation to store in hello

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

int main(int argc, char **argv) {
    string hello("Hello");
    hello = hello + " there";
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```
Stream manipulators

- iomanip.h defines a set of stream manipulator functions
  - pass them to a stream to affect formatting
Stream manipulators

- `setw(x)` sets the width of the next field to `x`
  - only affects the next thing sent to the output stream

```cpp
#include <iostream>
#include <cstdlib>
#include <iomanip>
using namespace std;

int main(int argc, char **argv) {
    cout << "Hi! " << setw(4) << 5 << " " << 5 << endl;
    cout << hex << 16 << " " << 13 << endl;
    cout << dec << 16 << " " << 13 << endl;
    return EXIT_SUCCESS;
}
```
Stream manipulators

- **hex** sets the stream to output integers in hexadecimal
  - stays in effect until you set the stream to some other base
  - **hex, dec, oct** are your choices

```cpp
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

int main(int argc, char **argv) {
    cout << "Hi! " << setw(4) << 5 << " " << 5 << endl;
    cout << hex << 16 << " " << 13 << endl;
    cout << dec << 16 << " " << 13 << endl;
    return EXIT_SUCCESS;
}
```
You can still use printf, though

- C is (roughly) a subset of C++

```c
#include <cstdio>
#include <cstdlib>

int main(int argc, char **argv) {
    printf("hello from C\n");
    return EXIT_SUCCESS;
}
```
std::cin is an object instance of class istream

- supports the >> operator for “extraction”
- cin also has a getline( ) method
• C: a pointer is a variable containing an address
  ‣ you can change its value to change what it is pointing to
  ‣ a pointer can contain the address of a different variable

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int *z = &x;
    *z += 1;  // sets x to 6
    x += 1;   // sets x (and therefore *z) to 7
    z = &y;   // sets z to the address of y
    *z += 1;  // sets y (and therefore *z) to 11
    return EXIT_SUCCESS;
}
```
Reminder: pointers

- **C**: A pointer is a variable containing an address
  - you can change its value to change what it is pointing to
  - a pointer can contain the address of a different variable

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int *z = &x;

    *z += 1; // sets x to 6
    x += 1;  // sets x (and therefore *z) to 7

    z = &y;  // sets z to the address of y
    *z += 1; // sets y (and therefore *z) to 11

    return EXIT_SUCCESS;
}
```

In the above code, `z` is a pointer to an integer variable. Initially, `z` points to `x` (at address `0xbfff2d4`). After the assignment `z = &y;`, `z` points to `y` (address `0xbfff347`). Changing the value of `y` via `*z` also changes the value of `x` since they share the same memory location.
Reminder: pointers

- **C**: a pointer is a variable containing an address
  - you can change its value to change what it is pointing to
  - a pointer can contain the address of a different variable

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int *z = &x;

    *z += 1; // sets x to 6
    x += 1; // sets x (and therefore *z) to 7

    z = &y; // sets z to the address of y
    *z += 1; // sets y (and therefore *z) to 11

    return EXIT_SUCCESS;
}
```

pointer.cc
Reminder: pointers

- **C:** a pointer is a variable containing an address
  - you can change its value to change what it is pointing to
  - a pointer can contain the address of a different variable

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int *z = &x;

    *z += 1;  // sets x to 6
    x += 1;   // sets x (and therefore *z) to 7

    z = &y;   // sets z to the address of y
    *z += 1;  // sets y (and therefore *z) to 11

    return EXIT_SUCCESS;
}
```
Reminder: pointers

- **C**: A pointer is a variable containing an address
  - you can change its value to change what it is pointing to
  - a pointer can contain the address of a different variable

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int *z = &x;

    *z += 1; // sets x to 6
    x += 1; // sets x (and therefore *z) to 7

    z = &y; // sets z to the address of y
    *z += 1; // sets y (and therefore *z) to 11

    return EXIT_SUCCESS;
}
```
Reminder: pointers

- C: a pointer is a variable containing an address
  - you can change its value to change what it is pointing to
  - a pointer can contain the address of a different variable

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int *z = &x;

    *z += 1; // sets x to 6
    x += 1; // sets x (and therefore *z) to 7

    z = &y; // sets z to the address of y
    *z += 1; // sets y (and therefore *z) to 11

    return EXIT_SUCCESS;
}
```
• C++: introduces references as *part of the language*
  ‣ a reference is **an alias** for some other variable
    • **alias:** another name that is bound to the aliased variable
    • mutating a reference **is** mutating the referenced variable

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int &z = x; // binds the name "z" to variable x
    z += 1;    // sets z (and thus x) to 6
    x += 1;    // sets x (and thus z) to 7
    z = y;     // sets z (and thus x) to the value of y
    z += 1;    // sets z (and thus x) to 11
    return EXIT_SUCCESS;
}
```

reference1.cc
• C++: introduces references as part of the language
  ‣ a reference is an alias for some other variable
    • alias: another name that is bound to the aliased variable
    • mutating a reference is mutating the referenced variable

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int &z = x; // binds the name "z" to variable x
    z += 1; // sets z (and thus x) to 6
    x += 1; // sets x (and thus z) to 7
    z = y; // sets z (and thus x) to the value of y
    z += 1; // sets z (and thus x) to 11
    return EXIT_SUCCESS;
}
```
References

• C++: introduces references as part of the language
  ‣ a reference is **an alias** for some other variable
  • **alias**: another name that is bound to the aliased variable
  • mutating a reference **is** mutating the referenced variable

```c
int main(int argc, char **argv) {
  int x = 5, y = 10;
  int &z = x;  // binds the name "z" to variable x
  z += 1;    // sets z (and thus x) to 6
  x += 1;    // sets x (and thus z) to 7
  z = y;     // sets z (and thus x) to the value of y
  z += 1;    // sets z (and thus x) to 11
  return EXIT_SUCCESS;
}
```

x, z  6

y  10

reference1.cc
C++: introduces references as part of the language

- a reference is an alias for some other variable
  - alias: another name that is bound to the aliased variable
  - mutating a reference is mutating the referenced variable

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int &z = x; // binds the name "z" to variable x

    z += 1; // sets z (and thus x) to 6
    x += 1; // sets x (and thus z) to 7

    z = y;  // sets z (and thus x) to the value of y
    z += 1; // sets z (and thus x) to 11

    return EXIT_SUCCESS;
}
```
References

- C++: introduces references as part of the language
  - A reference is an alias for some other variable
    - Alias: another name that is bound to the aliased variable
    - Mutating a reference is mutating the referenced variable

```c
int main(int argc, char **argv) {
  int x = 5, y = 10;
  int &z = x; // binds the name "z" to variable x

  z += 1;   // sets z (and thus x) to 6
  x += 1;   // sets x (and thus z) to 7

  z = y;    // sets z (and thus x) to the value of y
  z += 1;   // sets z (and thus x) to 11

  return EXIT_SUCCESS;
}
```

```
x, z | 10
---+----
y   | 10
```

```
reference1.cc
```
• C++: introduces references as part of the language
  ‣ a reference is **an alias** for some other variable
  • **alias**: another name that is bound to the aliased variable
  • mutating a reference **is** mutating the referenced variable

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int &z = x; // binds the name "z" to variable x
    z += 1; // sets z (and thus x) to 6
    x += 1; // sets x (and thus z) to 7
    z = y; // sets z (and thus x) to the value of y
    z += 1; // sets z (and thus x) to 11

    return EXIT_SUCCESS;
}
```
Pass by reference

• C++ allows you to truly pass-by-reference
  ‣ client passes in an argument with normal syntax
    • function uses reference parameters with normal syntax
    • modifying a reference parameter modifies the caller’s argument

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

int main(int argc, char **argv) {
    int a = 5, b = 10;
    swap(a, b);
    cout << "a: " << a << " b: " << b << endl;
    return EXIT_SUCCESS;
}
```
Pass by reference

- C++ allows you to truly pass-by-reference
  - client passes in an argument with normal syntax
    - function uses reference parameters with normal syntax
    - modifying a reference parameter modifies the caller’s argument

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

int main(int argc, char **argv) { 
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << " b: " << b << endl;
  return EXIT_SUCCESS;
}
```
Pass by reference

- C++ allows you to truly pass-by-reference
  - client passes in an argument with normal syntax
    - function uses reference parameters with normal syntax
    - modifying a reference parameter modifies the caller’s argument

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

int main(int argc, char **argv) {
    int a = 5, b = 10;
    
    swap(a, b);
    cout << "a: " << a << " b: " << b << endl;
    return EXIT_SUCCESS;
}
```
Pass by reference

- C++ allows you to truly pass-by-reference
  - client passes in an argument with normal syntax
  - function uses reference parameters with normal syntax
  - modifying a reference parameter modifies the caller’s argument

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

int main(int argc, char **argv) {
    int a = 5, b = 10;

    swap(a, b);
    cout << "a: " << a << " b: " << b << endl;
    return EXIT_SUCCESS;
}
```
Pass by reference

- C++ allows you to truly pass-by-reference
  - client passes in an argument with normal syntax
  - function uses reference parameters with normal syntax
  - modifying a reference parameter modifies the caller’s argument

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

int main(int argc, char **argv) {
    int a = 5, b = 10;
    swap(a, b);
    cout << "a: " << a << " b: " << b << endl;
    return EXIT_SUCCESS;
}
```
Pass by reference

• C++ allows you to truly pass-by-reference
  ‣ client passes in an argument with normal syntax
    • function uses reference parameters with normal syntax
    • modifying a reference parameter modifies the caller’s argument

```cpp
global void swap(int &x, int &y) { 
    int tmp = x;
    x = y;
    y = tmp;
}
global int main(int argc, char **argv) { 
    int a = 5, b = 10;

    swap(a, b);
    cout << "a: " << a << " b: " << b << endl;
    return EXIT_SUCCESS;
}
```
Pass by reference

• C++ allows you to truly pass-by-reference
  ‣ client passes in an argument with normal syntax
    • function uses reference parameters with normal syntax
    • modifying a reference parameter modifies the caller’s argument

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

int main(int argc, char **argv) {
    int a = 5, b = 10;
    swap(a, b);
    cout << "a: " << a << " b: " << b << endl;
    return EXIT_SUCCESS;
}
```
• **const**: cannot be changed
  - used much more in C++ than in C

```cpp
void BrokenPrintSquare(const int &i) {
  i = i*i;  // Compiler error here!
  std::cout << i << std::endl;
}

int main(int argc, char **argv) {
  int j = 2;
  BrokenPrintSquare(j);
  return EXIT_SUCCESS;
}
```

brokenpassbyrefconst.cc
• **const’s** syntax is confusing

```c
int main(int argc, char **argv) {
    int x = 5;      // x is an int
    const int y = 6; // y is a (const int)
y++;              // compiler error

    const int *z = &y;    // z is a (variable pointer) to a (const int)
    *z += 1;             // compiler error
    z++;                 // ok

    int *const w = &x;    // w is a (const pointer) to a (variable int)
    *w += 1;             // ok
    w++;                 // compiler error

    const int *const v = &x; // v is a (const pointer) to a (const int)
    *v += 1;             // compiler error
    v++;                 // compiler error

    return EXIT_SUCCESS;
}
```

`constmadness.cc`
• use const reference parameters to pass input
• use pointers to pass output parameters
  ‣ input parameters first, then output parameters last

```c
#include <cstdlib>

void CalcArea(const int &width, const int &height, int *area) {
    *area = width * height;
}

int main(int argc, char **argv) {
    int w = 10, h = 20, a;

    CalcArea(w, h, &a);
    return EXIT_SUCCESS;
}
```
Classes

• **class declaration syntax** (in a .h file)

```cpp
class Name {
  private:
    members;
  public:
    members;
};
```

• **class member definition syntax** (in a .cc file)

```cpp
returntype classname::methodname(parameters) {
  statements;
}
```

You can name your .cc, .h file anything (unlike Java)

- typically name them Classname.cc, Classname.h
#ifndef _POINT_H_
#define _POINT_H_

class Point {
    public:
        Point(const int x, const int y); // constructor
        int get_x() const { return x_; } // inline member function
        int get_y() const { return y_; } // inline member function
        double Distance(const Point &p) const; // member function
        void SetLocation(const int x, const int y); // member functn

    private:
        int x_; // data member
        int y_; // data member
}; // class Point

#endif // _POINT_H_
#include <cmath>
#include "Point.h"

Point::Point(const int x, const int y) {
    x_ = x;
    this->y_ = y; // "this->" is optional, unless names conflict
}

double Point::Distance(const Point &p) const {
    // We can access p’s x_ and y_ variables either through the
    // get_x(), get_y() accessor functions, or the x_, y_ private
    // member variables directly, since we’re in a member
    // function of the same class.
    double distance = (x_ - p.get_x()) * (x_ - p.get_x());
    distance += (y_ - p.y_) * (y_ - p.y_);
    return sqrt(distance);
}

void Point::SetLocation(const int x, const int y) {
    x_ = x;
    y_ = y;
}
```cpp
#include <iostream>
#include "Point.h"

using namespace std;

int main(int argc, char **argv) {
    Point p1(1, 2);  // stack allocate a new Point
    Point p2(4, 6);  // stack allocate a new Point

    cout << "p1 is: (" << p1.get_x() << ", ";
    cout << p1.get_y() << ")" << endl;

    cout << "p2 is: (" << p2.get_x() << ", ";
    cout << p2.get_y() << ")" << endl;

    cout << "dist : " << p1.Distance(p2) << endl;
    return 0;
}
```