

C++ References, Const, Classes

CSE 333 Autumn 2025

Instructors: Naomi Alterman, Chris Thachuk

Teaching Assistants:

Ann Baturytski	Derek de Leuw	Blake Diaz
Rishabh Jain	Chendur Jel Jayavelu	Lucas Kwan
Irene Xin Jie Lau	Nathan Li	Maya Odenheim
Advay Patil	Selim Saridede	Deeksha Vatwani
Angela Wu	Jiexiao Xu	

Administrivia

- ❖ Homework 2 due next Thursday (10/23)
 - File system crawler, indexer, and search engine
 - Demo in section this week
 - Note: `libhw1.a` (yours or ours) and the `.h` files from hw1 need to be in right directory (`~yourgit/hw1/`)
 - Note: use Ctrl-D to exit `searchshell`, test on directory of small self-made files

- ❖ A bit of a breather on exercises
 - No new exercise out today!

Lecture Outline

- ❖ **C++ References**
- ❖ `const` in C++
- ❖ C++ Classes Intro

Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

```
int main(int argc, char** argv) {  
    int x = 5, y = 10;  
    int* z = &x;  
  
    *z += 1;  
    x += 1;  
  
    z = &y;  
    *z += 1;  
  
    return EXIT_SUCCESS;  
}
```



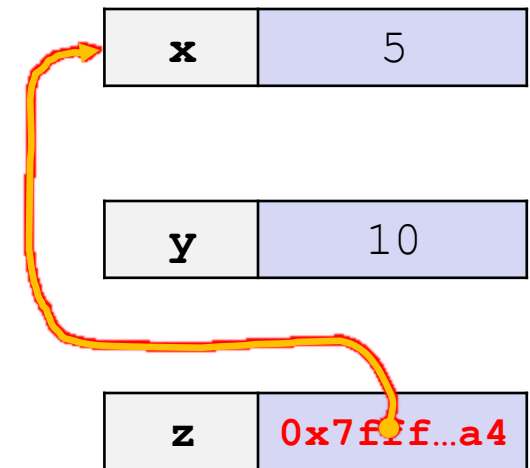
pointer.cc

Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

```
int main(int argc, char** argv) {  
    int x = 5, y = 10;  
    int* z = &x;  
  
    *z += 1;  
    x += 1;  
  
    z = &y;  
    *z += 1;  
  
    return EXIT_SUCCESS;  
}
```



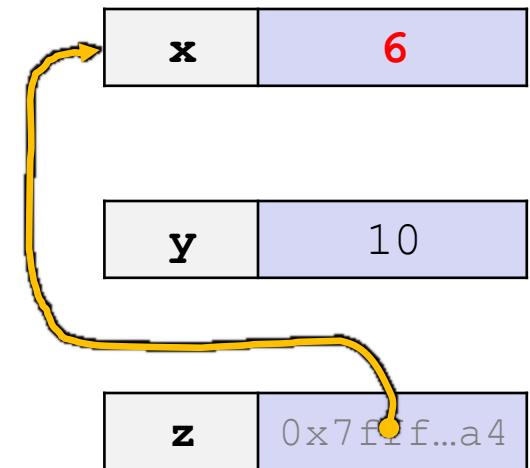
pointer.cc

Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

```
int main(int argc, char** argv) {  
    int x = 5, y = 10;  
    int* z = &x;  
  
    *z += 1;    // sets x to 6  
    x += 1;  
  
    z = &y;  
    *z += 1;  
  
    return EXIT_SUCCESS;  
}
```



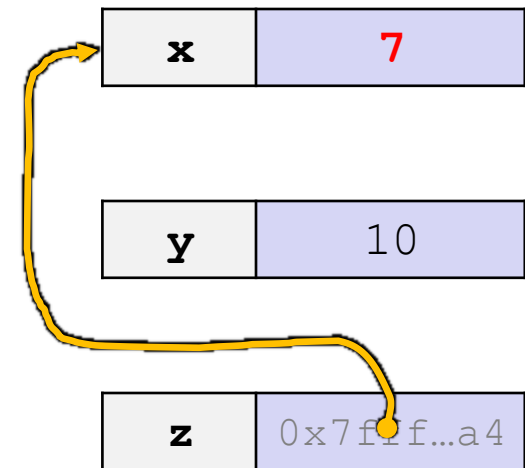
pointer.cc

Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and 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 *z) to 7  
  
    z = &y;  
    *z += 1;  
  
    return EXIT_SUCCESS;  
}
```



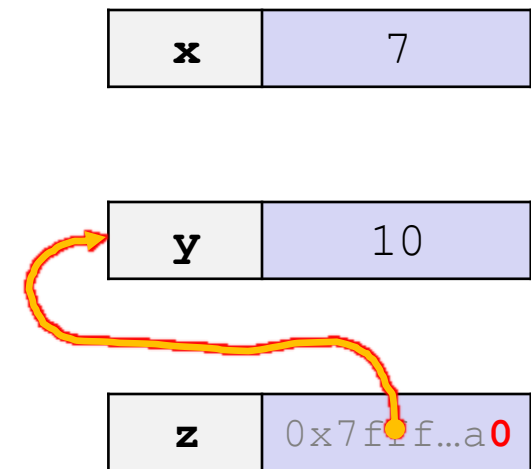
pointer.cc

Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and 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 *z) to 7  
  
    z = &y;    // sets z to the address of y  
    *z += 1;  
  
    return EXIT_SUCCESS;  
}
```



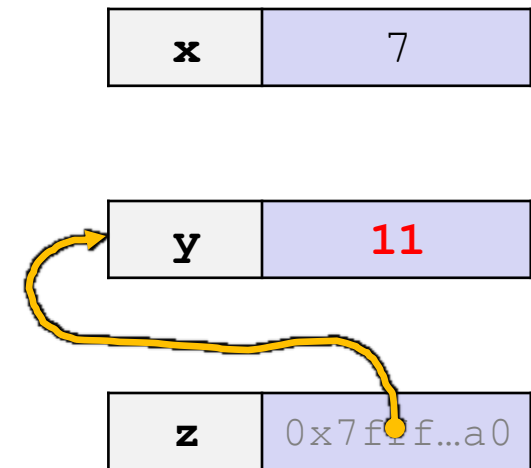
pointer.cc

Pointers Reminder

Note: Arrow points to *next* instruction.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and 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 *z) to 7  
  
    z = &y;    // sets z to the address of y  
    *z += 1;    // sets y (and *z) to 11  
  
→ return EXIT_SUCCESS;  
}
```



pointer.cc

References

Note: Arrow points
to *next* instruction.

- ❖ A **reference** is an alias for another variable
 - *Alias*: another name that is bound to the aliased variable
 - Mutating a reference *is* mutating the aliased variable
 - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {  
    int x = 5, y = 10;  
    int& z = x;  
  
    z += 1;  
    x += 1;  
  
    z = y;  
    z += 1;  
  
    return EXIT_SUCCESS;  
}
```

x	5
----------	---

y	10
----------	----

reference.cc

References

Note: Arrow points
to *next* instruction.

- ❖ A **reference** is an alias for another variable
 - *Alias*: another name that is bound to the aliased variable
 - Mutating a reference *is* mutating the aliased variable
 - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {  
    int x = 5, y = 10;  
    int& z = x;    // binds the name "z" to x  
    z += 1;  
    x += 1;  
  
    z = y;  
    z += 1;  
  
    return EXIT_SUCCESS;  
}
```

x, z	5
-------------	---

y	10
----------	----

reference.cc

References

Note: Arrow points
to *next* instruction.

- ❖ A **reference** is an alias for another variable
 - *Alias*: another name that is bound to the aliased variable
 - Mutating a reference *is* mutating the aliased variable
 - Introduced in C++ as part of the language

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

x, z	6
-------------	----------

y	10
----------	----

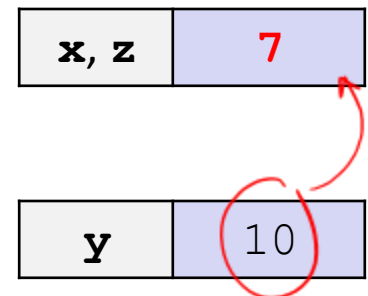
reference.cc

References

Note: Arrow points to *next* instruction.

- ❖ A **reference** is an alias for another variable
 - *Alias*: another name that is bound to the aliased variable
 - Mutating a reference *is* mutating the aliased variable
 - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {  
    int x = 5, y = 10;  
    int& z = x;    // binds the name "z" to x  
  
    z += 1;    // sets z (and x) to 6  
    x += 1;    // sets x (and z) to 7  
  
    z = y;    // normal assignment!  
    z += 1;  
  
    return EXIT_SUCCESS;  
}
```



reference.cc

References

Note: Arrow points
to *next* instruction.

- ❖ A **reference** is an alias for another variable
 - *Alias*: another name that is bound to the aliased variable
 - Mutating a reference *is* mutating the aliased variable
 - Introduced in C++ as part of the language

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

x, z	10
-------------	-----------

y	10
----------	----

reference.cc

References

Note: Arrow points
to *next* instruction.

- ❖ A **reference** is an alias for another variable
 - *Alias*: another name that is bound to the aliased variable
 - Mutating a reference *is* mutating the aliased variable
 - Introduced in C++ as part of the language

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

x, z	11
-------------	-----------

y	10
----------	----

reference.cc

Pass-By-Reference

Note: Arrow points
to *next* instruction.

- ❖ C++ allows you to use real *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!

```
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;  
}
```

aliases that are bound to arguments

(main) a	5
-----------------	---

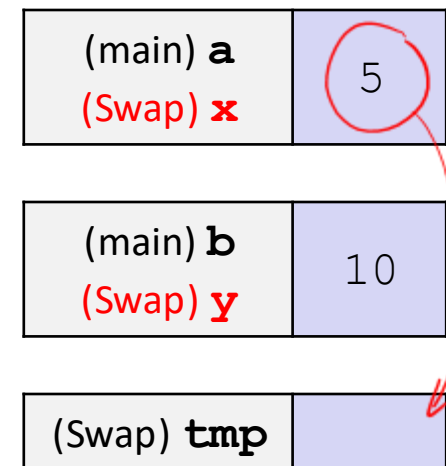
(main) b	10
-----------------	----

Pass-By-Reference

Note: Arrow points to *next* instruction.

- ❖ C++ allows you to use real *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!

```
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;  
}
```

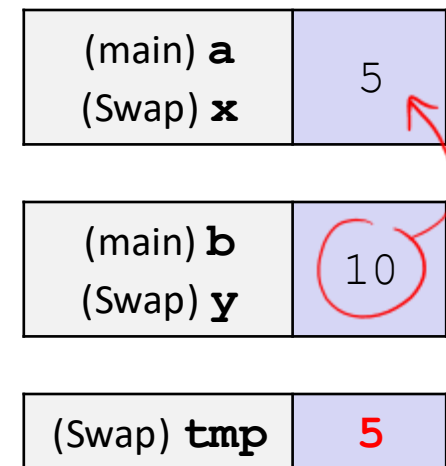


Pass-By-Reference

Note: Arrow points to *next* instruction.

- ❖ C++ allows you to use real *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!

```
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

Note: Arrow points to *next* instruction.

- ❖ C++ allows you to use real *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!

```
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;  
}
```

(main) a	10
(Swap) x	

(main) b	10
(Swap) y	

(Swap) tmp	5
-------------------	---

Pass-By-Reference

Note: Arrow points
to *next* instruction.

- ❖ C++ allows you to use real *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!



```
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;  
}
```

(main) a	10
(Swap) x	

(main) b	5
(Swap) y	

(Swap) tmp	5
-------------------	---

Pass-By-Reference

Note: Arrow points to *next* instruction.

- ❖ C++ allows you to use real *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!

```
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;  
}
```

(main) a	10
-----------------	----

(main) b	5
-----------------	---

pollev.com/cse333

What will happen when we try to compile and run this code?

poll1.cc

A. Output "(1,2,3)"

B. Output "(3,2,3)"

C. Compiler error about arguments to Foo (in main)

D. Compiler error about body of Foo

E. We're lost...

```
void Foo(int& x, int* y, int z) {  
    z = *y;  
    x += 2;  
    y = &x;  
}
```

```
int main(int argc, char** argv) {  
    int a = 1;  
    int b = 2;  
    int& c = a;  
  
    Foo(a, &b, c);  
    std::cout << "(" << a << ", " << b  
    << ", " << c << ")" << std::endl;  
  
    return EXIT_SUCCESS;  
}
```

Diagram illustrating the state of variables and pointers during the execution of the code:

- In `main`: `a` is 1, `b` is 2, and `c` is a reference to `a`.
- In `Foo`: `x` is a reference to `a`, `y` is a pointer to `b`, and `z` is a reference to `x`.
- Inside `Foo`, `z = *y` sets `z` to the value of `b` (2).
- Then `x += 2` increments `a` to 3.
- Finally, `y = &x` updates `y` to point to `x` (which points to `a`).
- When `Foo` returns, `main` prints the values of `a`, `b`, and `c`, which are 3, 2, and 3 respectively.

Lecture Outline

- ❖ C++ References
- ❖ **const in C++**
- ❖ C++ Classes Intro

const

- ❖ `const`: this cannot be changed/mutated
 - Used *much* more in C++ than in C
 - ★ Signal of intent to compiler; meaningless at hardware level
 - Results in compile-time errors

```
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 and Pointers

❖ Pointers can change data in two different contexts:

- 1) You can change the value of the pointer
- 2) You can change the thing the pointer points to (via dereference)

`int x;`
`int *p = &x;`



❖ `const` can be used to prevent either/both of these behaviors!

- `const` next to pointer name means you can't change the value of the pointer

`int *const p;` X ✓

- `const` next to data type pointed to means you can't use this pointer to change the thing being pointed to

`const int *p;` ✓ X

- Tip: read variable declaration from *right-to-left*

const and Pointers

❖ The syntax with pointers is confusing:

```
int main(int argc, char** argv) {  
    int x = 5; // int  
    const int y = 6; // (const int)  
    y++;  
  
    const int* z = &y; // pointer to a (const int)  
    *z += 1;  
    z++;  
  
    int* const w = &x; // (const pointer) to a (variable int)  
    *w += 1;  
    w++;  
  
    const int* const v = &x; // (const pointer) to a (const int)  
    *v += 1;  
    v++;  
  
    return EXIT_SUCCESS;  
}
```

const and Pointers

❖ The syntax with pointers is confusing:

```
int main(int argc, char** argv) {  
    int x = 5;                // int  
    const int y = 6;          // (const int)  
    ✗ y++;                     // compiler error  
  
    const int* z = &y;        // pointer to a (const int)  
    ✗ *z += 1;                 // compiler error  
    ✓ z++;                     // ok  
  
    ✓ int* const w = &x;       // (const pointer) to a (variable int)  
    *w += 1;                   // ok  
    ✗ w++;                     // compiler error  
  
    const int* const v = &x;  // (const pointer) to a (const int)  
    ✗ *v += 1;                 // compiler error  
    ✗ v++;                     // compiler error  
  
    return EXIT_SUCCESS;  
}
```



const Parameters

Make parameters const when you can!

- ❖ A const parameter *cannot* be mutated inside the function

- Therefore it does not matter if the argument can be mutated or not

- ❖ A non-const parameter *may* be mutated inside the function

- Compiler won't let you pass in const parameters

```
void Foo(const int* y) {  
    std::cout << *y << std::endl;  
}  
  
void Bar(int* y) {  
    std::cout << *y << std::endl;  
}  
  
int main(int argc, char** argv) {  
    const int a = 10;  
    int b = 20;  
  
    { Foo(&a);      // OK  
      Foo(&b);      // OK  
      Bar(&a);      // not OK - error  
      Bar(&b);      // OK  
    }  
  
    return EXIT_SUCCESS;  
}
```

doesn't actually modify value of y!

pollev.com/cse333

What will happen when we try to compile and run this code?

A. Output "(2,4,0)"

B. Output "(2,4,3)"

C. Compiler error
about arguments
to Foo (in main)

D. Compiler error
about body of Foo

E. We're lost...

can't change x, but can change the thing it points to poll2.cc

```
void Foo(int* const x,  
int ref ↗ int& y, int z) {  
    *x += 1; // allowed  
    y *= 2; // allowed  
    z -= 3; // allowed, but has no lasting effect  
}  
  
int main(int argc, char** argv) {  
    const int a = 1;  
    int b = 2, c = 3;  
  
    Foo(X &a, ✓ b, ✓ c);  
    std::cout << "(" << a << "," << b  
        << "," << c << ")" << std::endl;  
  
    return EXIT_SUCCESS;  
}
```

local copy of int



When to Use References?

- ❖ A stylistic choice, not mandated by the C++ language
- ❖ Google C++ style guide suggests:

- Input parameters:

- Either use values (for primitive types like `int` or small structs/objects)
- Or use `const` references (for complex struct/object instances)

avoid making copy of argument

don't change in function body; allows both const & non-const arguments

- Output parameters:

- Use `const` pointers
 - Unchangeable pointers referencing changeable data

- Ordering:

- List input parameters first, then output parameters last

output parameter unnecessary for this example, but used for illustration

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

ordinary int (not int&) probably better here, but shows how const ref can be used

Lecture Outline

- ❖ C++ References
- ❖ `const` in C++
- ❖ **C++ Classes Intro**

Classes

❖ Class definition syntax (in a `.h` file):

```
class Name {  
    public:  
        // public member definitions & declarations go here  
  
    private:  
        // private member definitions & declarations go here  
}; // class Name
```

don't forget!

- Members can be functions (methods) or data (variables)

❖ Class member function definition syntax (in a `.cc` file):

```
retType Name::MethodName(type1 param1, ..., typeN paramN) {  
    // body statements  
}
```

- (1) *define* within the class definition or (2) *declare* within the class definition and then *define* elsewhere

Class Organization

- ❖ It's a little more complex than in C when modularizing with `struct` definition:
 - Class definition is part of interface and should go in `.h` file
 - Private members still must be included in definition (!)
 - Usually put member function definitions into companion `.cc` file with implementation details
 - Common exception: setter and getter methods
 - These files can also include **non-member functions** that use the class
- ❖ Unlike Java, you can name files anything you want
 - Typically `Name.cc` and `Name.h` for **class** `Name`

Const & Classes

- ❖ Like other data types, **objects** can be declared as `const`:
 - Once a `const` object has been constructed, its member variables can't be changed
 - Can only invoke member functions that are labeled `const`
- ❖ You can declare a member **function** of a class as `const`
 - This means that it cannot modify the object it was called on
 - The compiler will treat member variables as `const` inside the function at compile time
 - If a member function doesn't modify the object, mark it `const`!

Class Definition (.h file)



Point.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 function

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

#endif // POINT_H_
```

declarations (points to the first four public member functions)

this const means that this function is not allowed to change the object on which it is called (the implicit "this" pointer) (points to the const in the constructor and the const in the inline functions)

function definitions (points to the inline functions)

compiler may choose to expand inline (like a macro) instead of an actual function call (points to the inline keyword)

naming convention for class data members (Google C++ style guide) (points to the x_ and y_ data members)

Class Member Definitions (.cc file)

Point.cc

```
#include <cmath>
#include "Point.h"
```

BAD STYLE
used here on purpose

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

equivalent to y_ = y;

"this" is a (Point const)*

```
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);
}
```

equivalent to p.x_

```
void Point::SetLocation(const int x, const int y) {
    x_ = x;
    y_ = y;
}
```

can't be const because we are mutating "this"

Class Usage (.cc file)

usepoint.cc

```
#include <iostream>
#include <cstdlib>
#include "Point.h"

using namespace std;

int main(int argc, char** argv) {
    Point p1(1, 2); // allocate a new Point on the Stack
    Point p2(4, 6); // allocate a new Point on the Stack } calls defined constructor

    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 EXIT_SUCCESS;
}
```

"dot notation" used for member functions

Reading Assignment

- ❖ Before next time, *read* the sections in *C++ Primer* covering class constructors, copy constructors, assignment (`operator=`), and destructors
 - Ignore “move semantics” for now
 - The table of contents and index are your friends...

Extra Exercise #1

- ❖ Write a C++ program that:
 - Has a class representing a 3-dimensional point
 - Has the following methods:
 - Return the inner product of two 3D points
 - Return the distance between two 3D points
 - Accessors and mutators for the x , y , and z coordinates

Extra Exercise #2

- ❖ Write a C++ program that:
 - Has a class representing a 3-dimensional box
 - Use your Extra Exercise #1 class to store the coordinates of the vertices that define the box
 - Assume the box has right-angles only and its faces are parallel to the axes, so you only need 2 vertices to define it
 - Has the following methods:
 - Test if one box is inside another box
 - Return the volume of a box
 - Handles `<<`, `=`, and a copy constructor
 - Uses `const` in all the right places