C++ References, Const, Classes

CSE 333 Winter 2021

Instructor: John Zahorjan

Teaching Assistants:

Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Lecture Outline

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

Pointers Reminder

Note: Arrow points to *next* instruction.

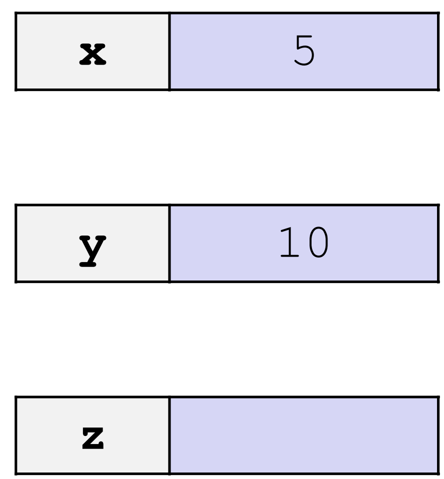
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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.

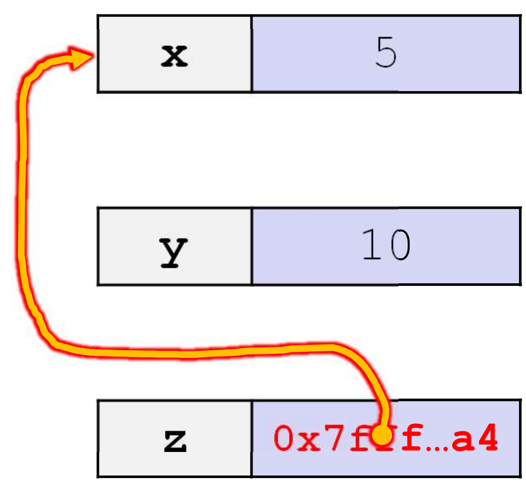
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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.

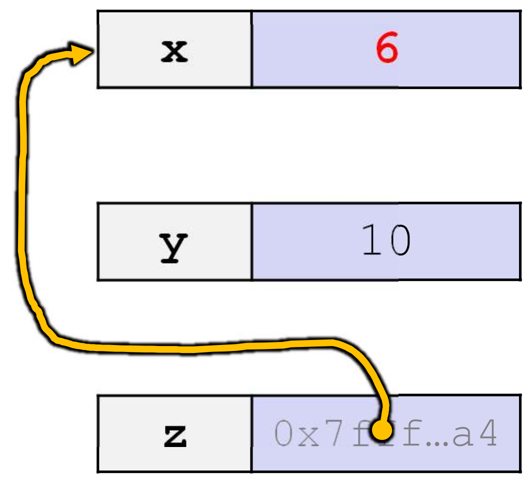
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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.

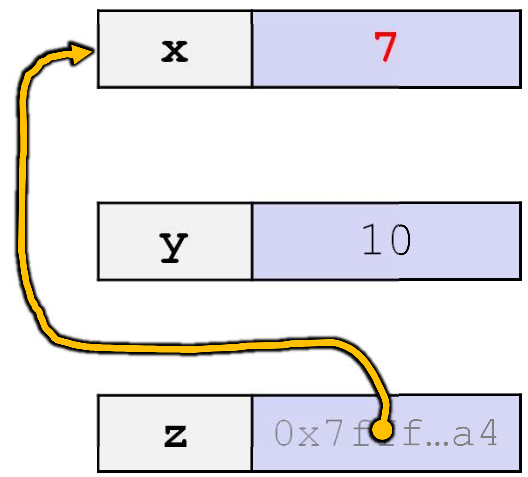
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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.

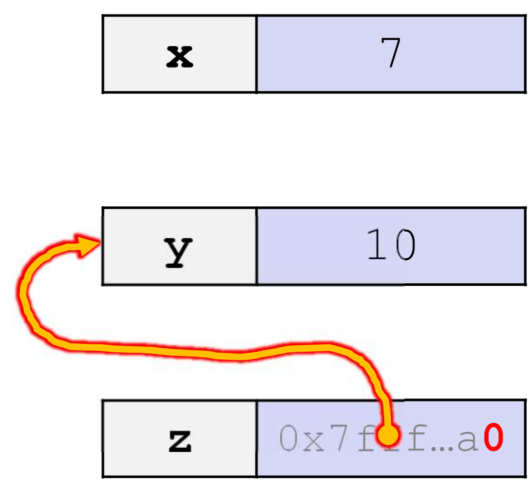
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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.

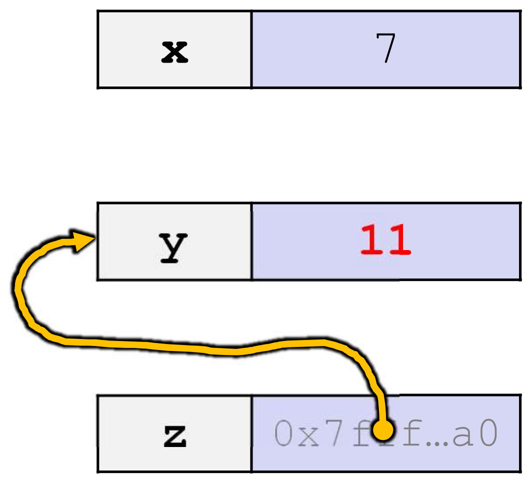
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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

<u>Note</u>: Arrow points to *next* instruction.

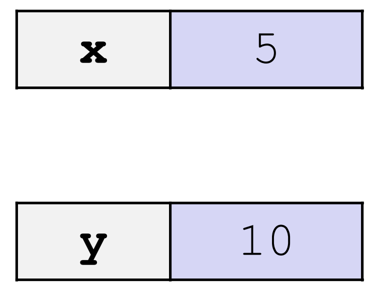
- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
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;
}
```



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;
  z += 1;

  return EXIT_SUCCESS;
}
```

x, z	7
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;

  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

References

- There is no way to change what a reference is an alias for

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

  z = y;    // sets x to 10
  z = &y;   // sets x to the address of y!
  &z = y;   // compile time error

  return EXIT_SUCCESS;
}
```

- That means a reference must always be initialized when declared
  - int& x; // is an error

Using References: Pass-By-Reference

- 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**	5
(main) **b**	10

Note: Arrow points to *next* instruction.

Note: Arrow points to *next* instruction.

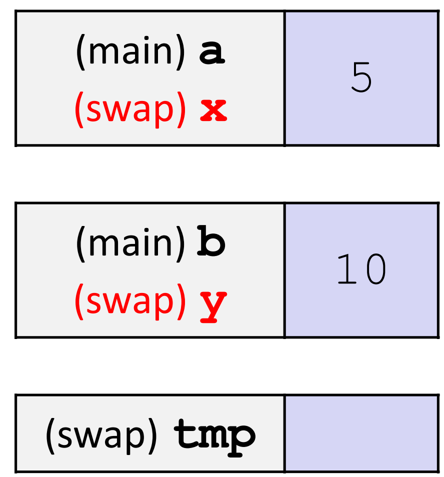
Pass-By-Reference

- 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;    // <-- arrow
  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

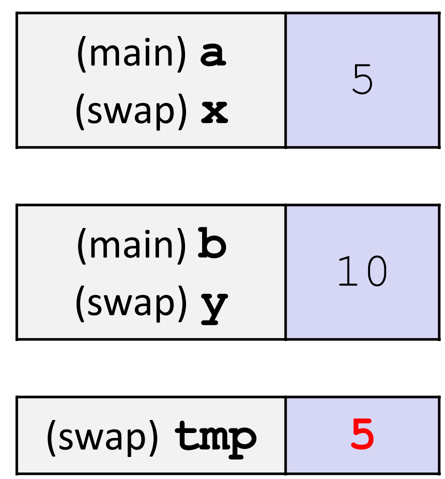
Note: Arrow points to *next* instruction.

- 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;        // <- arrow
  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** (swap) **x**	10
(main) **b** (swap) **y**	10
(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** (swap) **x**	10
(main) **b** (swap) **y**	**5**
(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

Return-by-Reference

- C and C++ normally "return by value"
  - The thing the caller gets back is a copy of the thing the callee returned
- Returning a reference gives caller access to the thing returned
- Example:
  - std::vector<int> my_vec{0,1,2,3,4};
    my_vec.at(3) = -3;
    for (auto & i : my_vec) std::cout << i << " ";
  - Output: 0 1 2 -3 4

Pass-By-Reference / Return-By-Reference

- Why would you use them?
  - Performance
    - It's too expensive to copy the thing being passed or returned
      - In C, you have to copy potentially a lot of bytes
      - In C++, additionally, if you're communicating an object, you have to create the object, which means you **have to** run a constructor
  - Functionality
    - You want to give the callee / caller access to the thing passed / returned
      - Including output parameters

Lecture Outline

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

`const` Keyword

- `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;
}
```

Pointers and `const`

- There are two natural assignments involving pointers:
  1) You can change the value of the pointer (what it points to)
  2) You can change the thing the pointer points to (via dereference)

- `const` can be applied to either/both of these!

- Just like the '*' used to declare a pointer can go in a few places, so can `const`

- Tip: read variable declaration from *right-to-left*
  - *Tip: write "const" so that reading right to left makes sense*

`const` and Pointers

- The syntax with pointers is confusing:

```
  const int y = 6;        // can't assign to y after this
//int const y = 6;        // exactly the same as const int y = 6
  y++;                    // compiler error

  const int *z = &y;      // pointer to a (const int)
//int const *z = &y;      // exactly the same as "const int *"
  *z += 1;                // compiler error
  z++;                    // doesn't cause a compile-time error

  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)
//int const *const v = &x; // exactly the same
  *v += 1;                // compiler error
  v++;                    // compiler error
```

constmadness.cc

const and Pointers

- int const * * const p = y;
- Which of the following aren't errors?
  - p = 0;
  - *p = 0;
  - **p = 0;
  - ***p = 0;
  - &p = 0;
  - p = &0;

const and Pointers

- int const * * const p = &y;
- Which of the following aren't errors?
  - p = 0;
  - *p = 0;
  - **p = 0;
  - ***p = 0;
  - &p = 0;
  - p = &0;

Bonus Examples

- Which of the following lines can compile without error?

```
const int & p = y;
int const & q = y;
int & const r = y;
p = 0;
q = 0;
r = 0;
```

Bonus Examples

- Which of the following lines can compile without error?

```
const int & p = y;
int const & q = y;
int & const r = y;
p = 0;
q = 0;
r = 0;
```

`const` Parameters

- If a method defintely does not modify a parameter, it should specify it as `const`
  - That may allow the compiler to perform some optimizations in the callers that wouldn't be legal otherwise
  - Also, sometimes it's required...

```
int my_strlen(char *p_string)
{
   char *q;
   if ( p_string == nullptr )
    return 0;
   for ( q=p_string; *q; q++)
    ;
   return q-p_string;
}
```

Should be const (points to `char *p_string`)

`const` Parameters

```
int my_strlen(char *p_string)
{
   char *q;
   if ( p_string == nullptr )
     return 0;
   for ( q=p; *q; q++)
     ;
   return q-p;
}
```

```
int main(int argc, char *argv[])
{
  for ( int i=0; i<argc; i++ )
    printf("'%s' -> %d\n", argv[i], my_strlen(argv[i]));
  return EXIT_SUCCESS;
}
```

```
[attu2] > ./a.out one two three
'./a.out' -> 7
'one' -> 3
'two' -> 3
'three' -> 5
```

const Parameters

```
int my_strlen(char *p_string)
{
   char *q;
   if ( p_string == nullptr )
     return 0;
   for ( q=p; *q; q++)
     ;
   return q-p;
}
```

```
int main(int argc, char *argv[])
{
  int len = my_strlen("cse333");
  return EXIT_SUCCESS;
}
```

```
$ g++ -std=c++17 -Wall -g test.cc
test.cc: In function 'int main(int, char**)':
test.cc:15:23: warning: ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
   15 |   int len = my_strlen("cse333");
      |                       ^~~~~~~~
[attu2] ~/tmp> ./a.out
6
```

`const` Parameter Troubles

- The issue occurs much more frequently than you likely expect
- Once some routine says something is const, the compiler wants to keep it const
- If you don’t say const, the caller will have issues
  - That caller can be you...

Lecture Outline

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

C++ class declarations and definitions

- Code for C++ classes (typically) goes in two files
- The .h file declares the class
  - lists instance variables and method names, but not method implementations
  - including the "private" portions
- The .cc file defines the methods
  - Gives code for them
- Usually...

- If the class name is ABCD, the files are usually named ABCD.h and ABCD.cc
  - but it's only convention

Classes – the .h file

- The class declaration goes in a .h file

```
class MyClass {
 public:
  // public member declarations go here
  int ExampleMethod(int x, int y);

 private:
  // private member declarations go here
};
```

  - Members can be functions (methods) or data (variables)
  - The file is usually called MyClass.h
  - Don't forget the trailing semi-colon!

Classes – the .cc file

- Class member function definitions go in the .cc file

```
int MyClass::ExampleMethod(int x, int y, int z) {
  // body statements
}
```

- There is no compiler enforced relationship among the names of the class, the .h file, and the .cc file
  - You must give the method's fully qualified name when defining it
    `MyClass::ExampleMethod`

Class .h and .cc files

- Client code must #include the .h file to use the class
- Private members must be included in the .h file
  - They're private in that the compiler won't compile non-class code that attempts to manipulate them

- So why expose private information to clients?
  - Clients can perform one operation involving private instance variables: object creation
    - The variable declaration: `vector<string> word_list;`
  - The compiler needs to know the size of the object so it can allocate space for it (on the stack, say)

Inlining

- Normally, a function call in the source code results in a procedure call at runtime
  - all the overheads associated with it
- *Inlining* is the idea of injecting the procedure’s code into the caller’s code at compile time
  - Avoids procedure call/return overhead at runtime
  - Enables possible optimizations of code across the (logical) procedure call/return boundaries
- To inline, a procedure the compiler must have access to the procedure’s implementation when compiling a call to it

Inlining

- C++ is very concerned about performance
- It has a few ways the programmer can use to encourage the compiler to inline a method
  - But the compiler knows best – it may, or may not, inline

- The simplest of them is to simply provide the method's definition in the .h file (and not in the .cc file)
  - This is often done for particularly trivial methods, like getters

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_; }            // potential inline
  int get_y() const { return y_; }            // potential inline
  double Distance(const Point& p) const;
  void SetLocation(const int x, const int y);

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

#endif  // _POINT_H_
```

Providing method bodies enables inlining

Cannot be inlined

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_; }      // potential inline
  int get_y() const { return y_; }      // potential inline
  double Distance(const Point& p) const;
  void SetLocation(const int x, const int y);

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

#endif  // _POINT_H_
```

Promises that the method doesn't modify the object. Useful when compiling caller for optimization reasons.

The .cc file - Class Member Definitions

Point.cc

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

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

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;
}
```

Class Usage (.cc file)

usepoint.cc

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

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

  std::cout << "p1 is: (" << p1.get_x() << ", "
            << p1.get_y() << ")\n"
            << "p2 is: (" << p2.get_x() << ", ";
            << p2.get_y() << ")\n"
            << "dist : " << p1.Distance(p2) << std::endl;
  return 0;
}
```