COMP 3522

Object Oriented Programming in C++

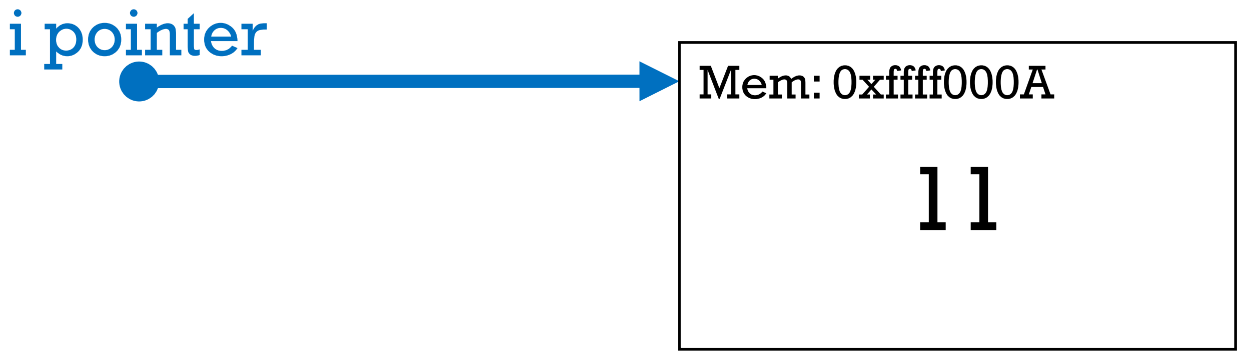
Week 3, Day 1

Continuing from last class: Memory leak

```
int *i = new int{11};
int *a = new int{99};
i = a; //creates a memory leak

delete i; //free allocated memory
```

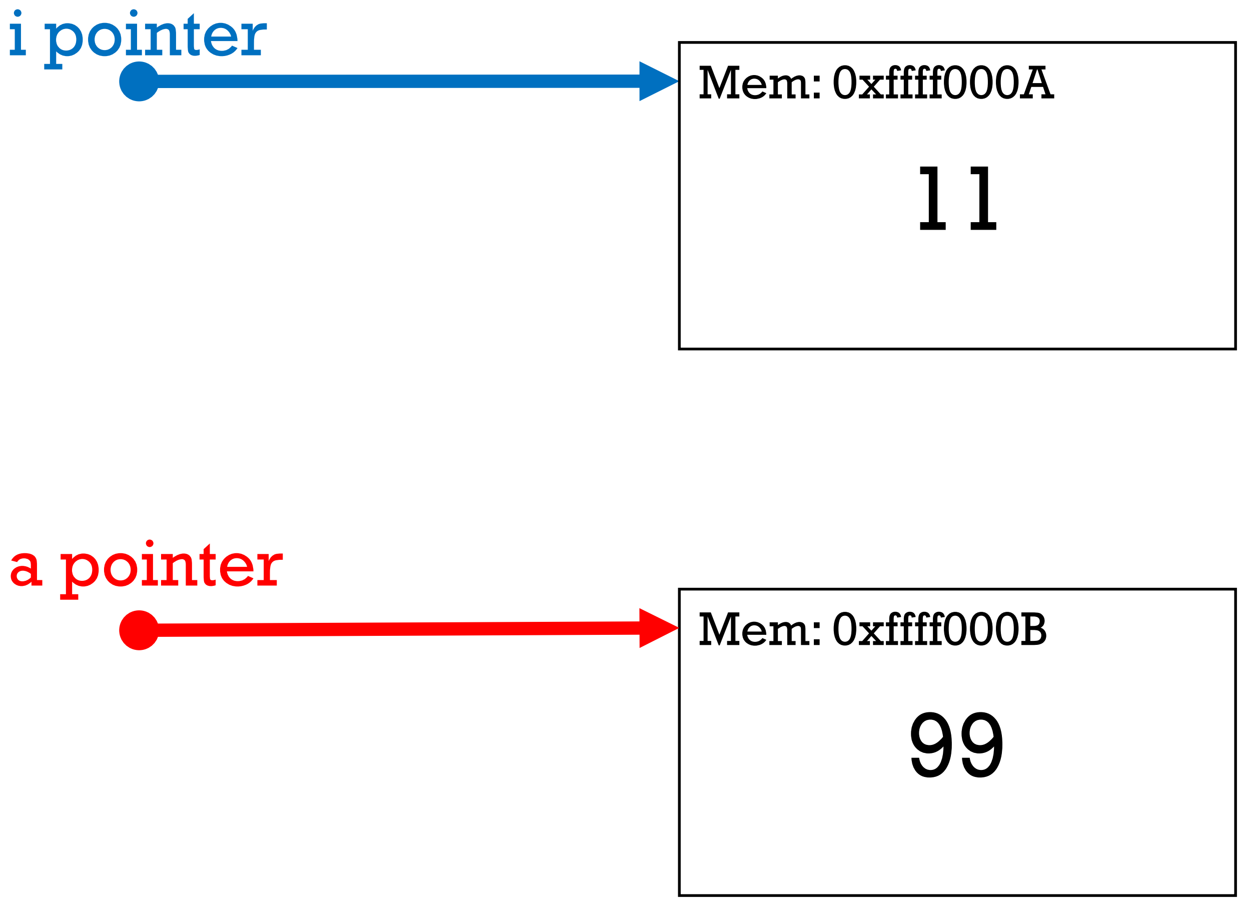
Memory leak

```
int *i = new int{11};
```



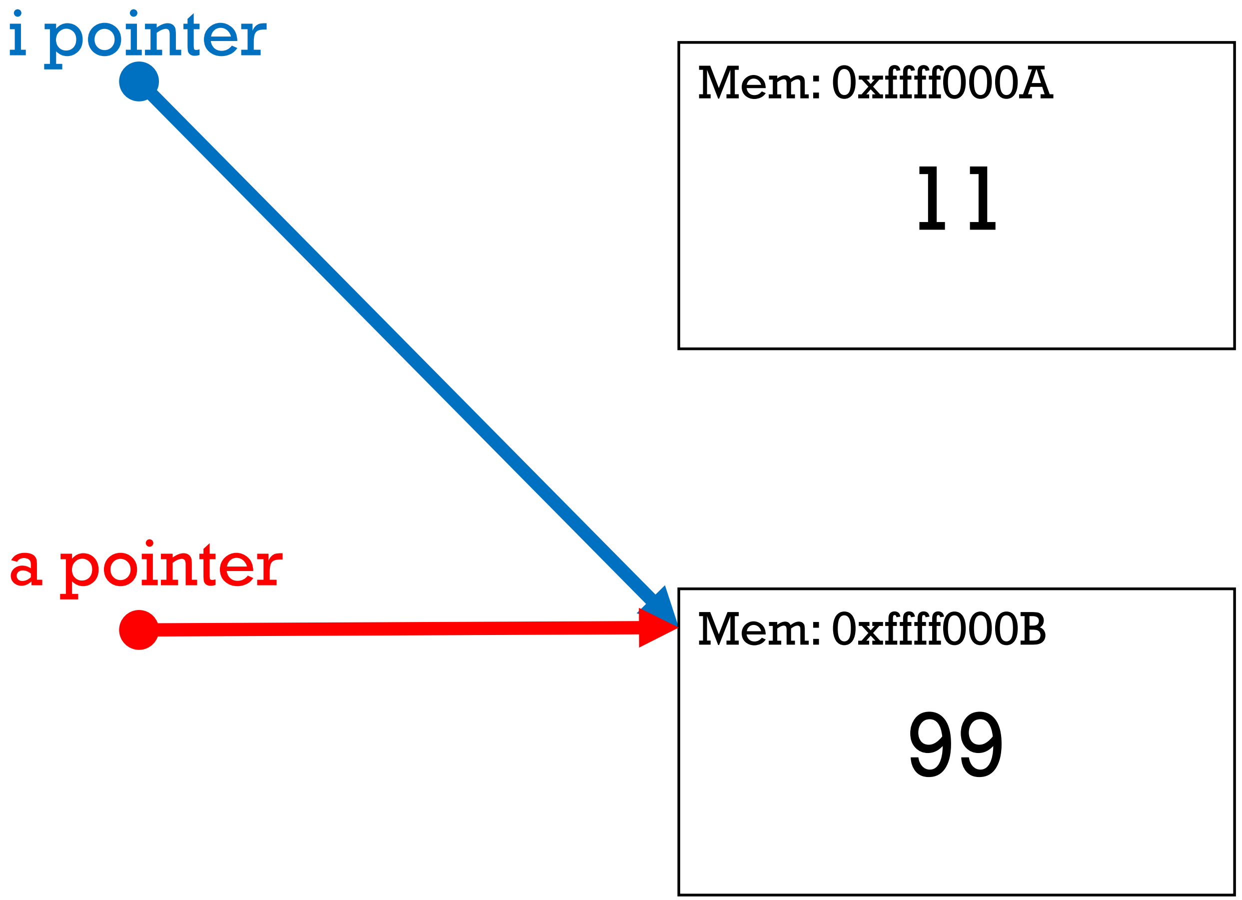
Memory leak

```
int *i = new int{11};
int *a = new int{99};
```



Memory leak

```
int *i = new int{11};
int *a = new int{99};
i = a; //creates a memory leak
```



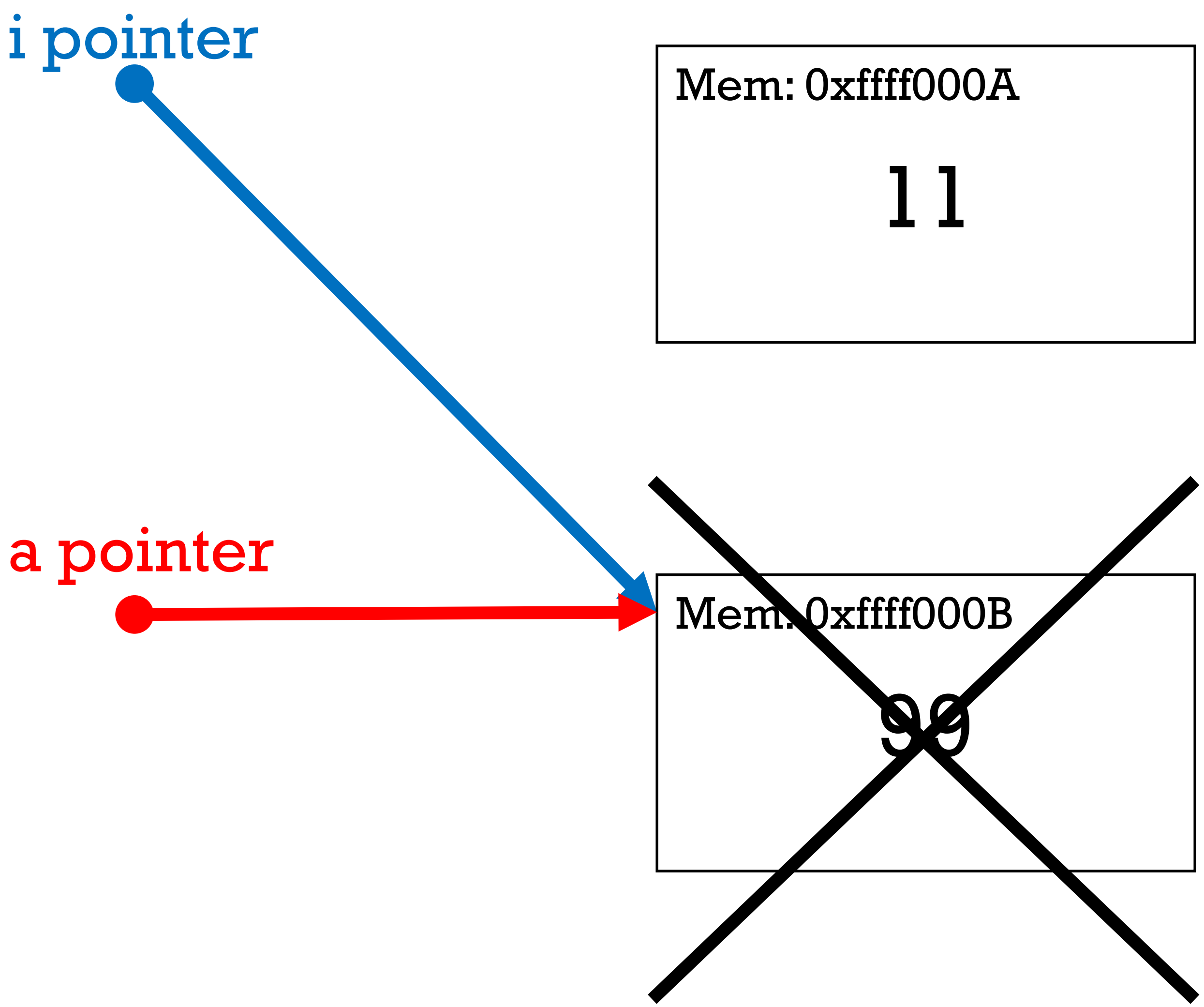
Memory leak

int *i = new int{11};

int *a = new int{99};

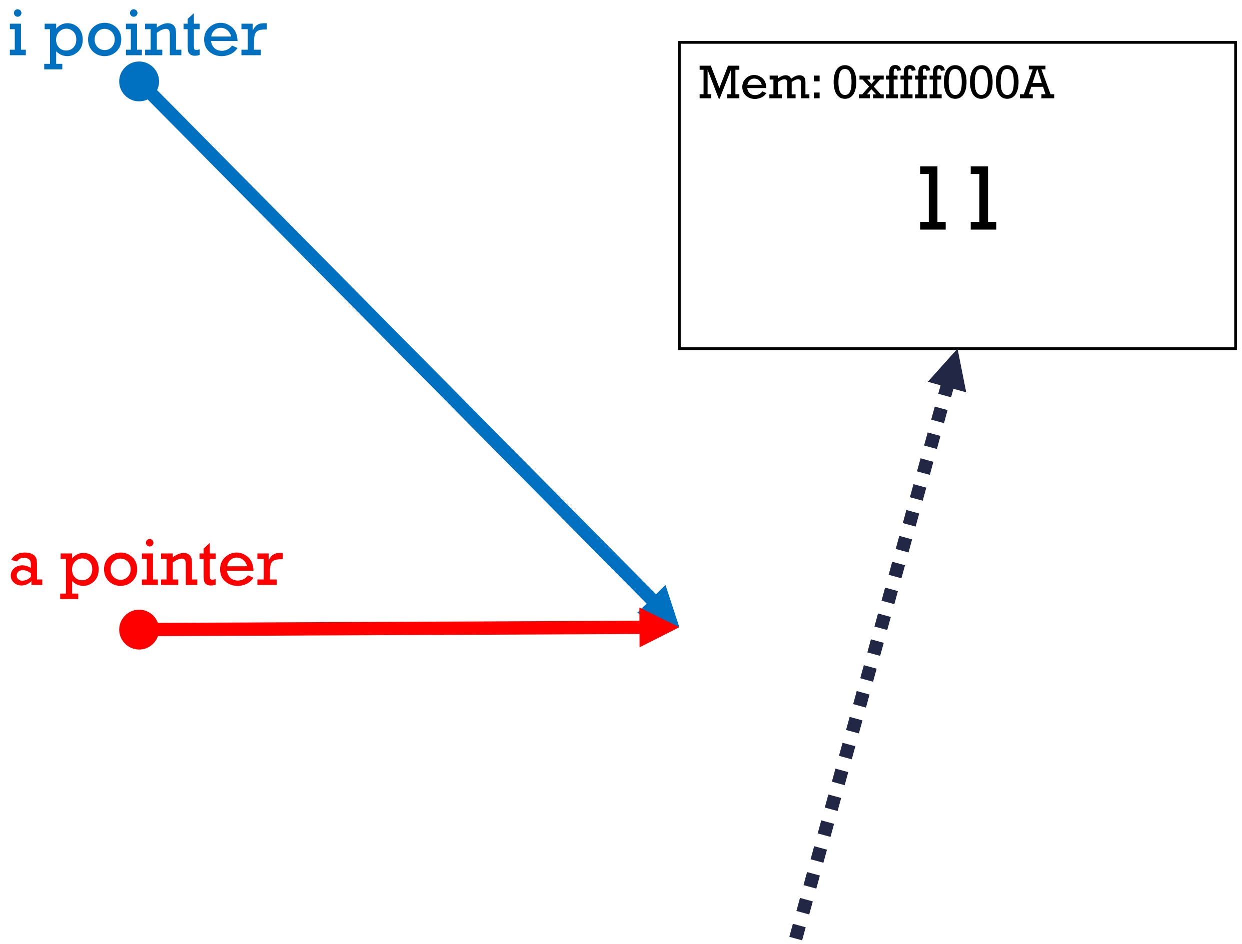
i = a; //creates a memory leak

delete i; //free allocated memory



Memory leak

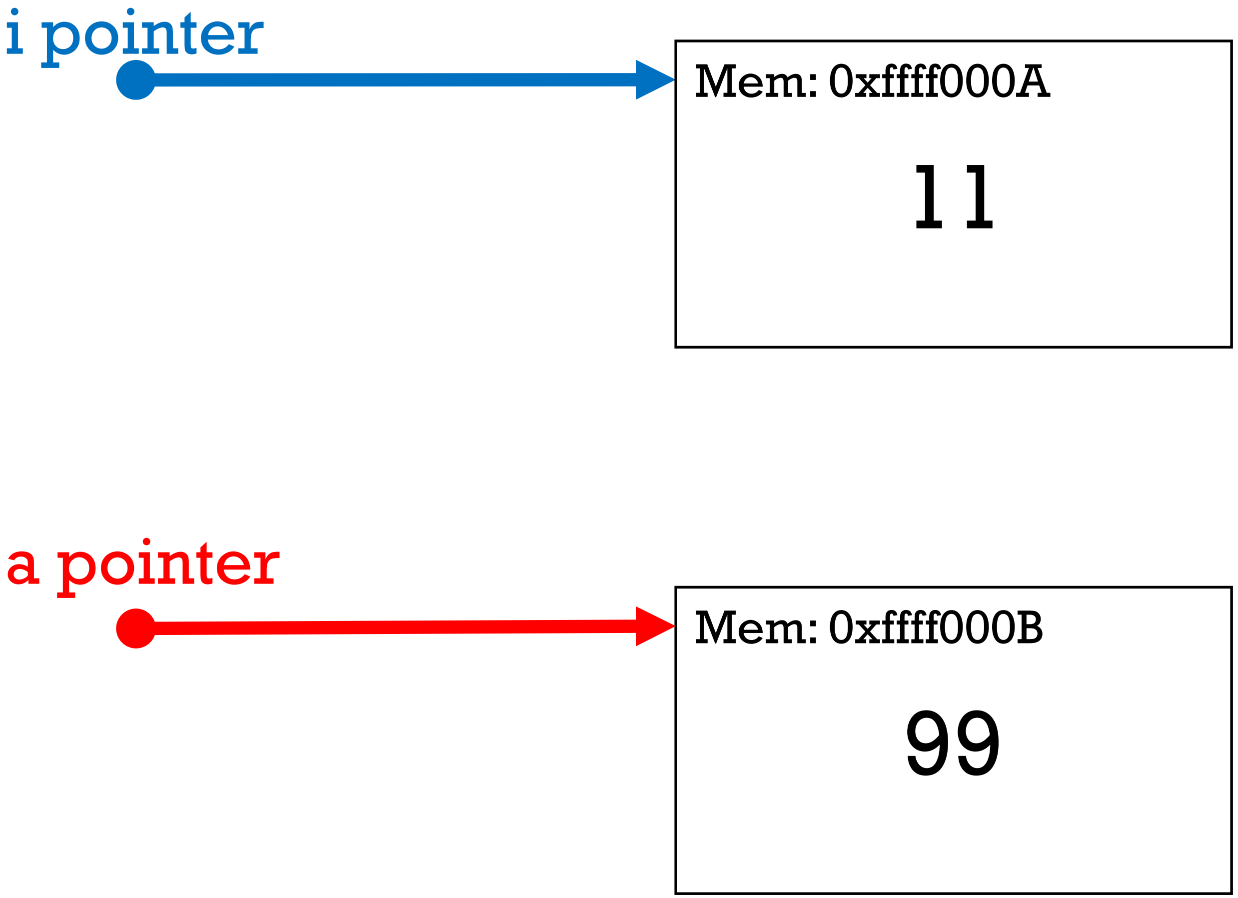
```
int *i = new int{11};
int *a = new int{99};
i = a; //creates a memory leak

delete i; //free allocated memory
```



Nothing pointing at data object, so no way for us to delete it. MEMORY LEAK

Memory leak solution

```
int *i = new int{11};
int *a = new int{99};
```

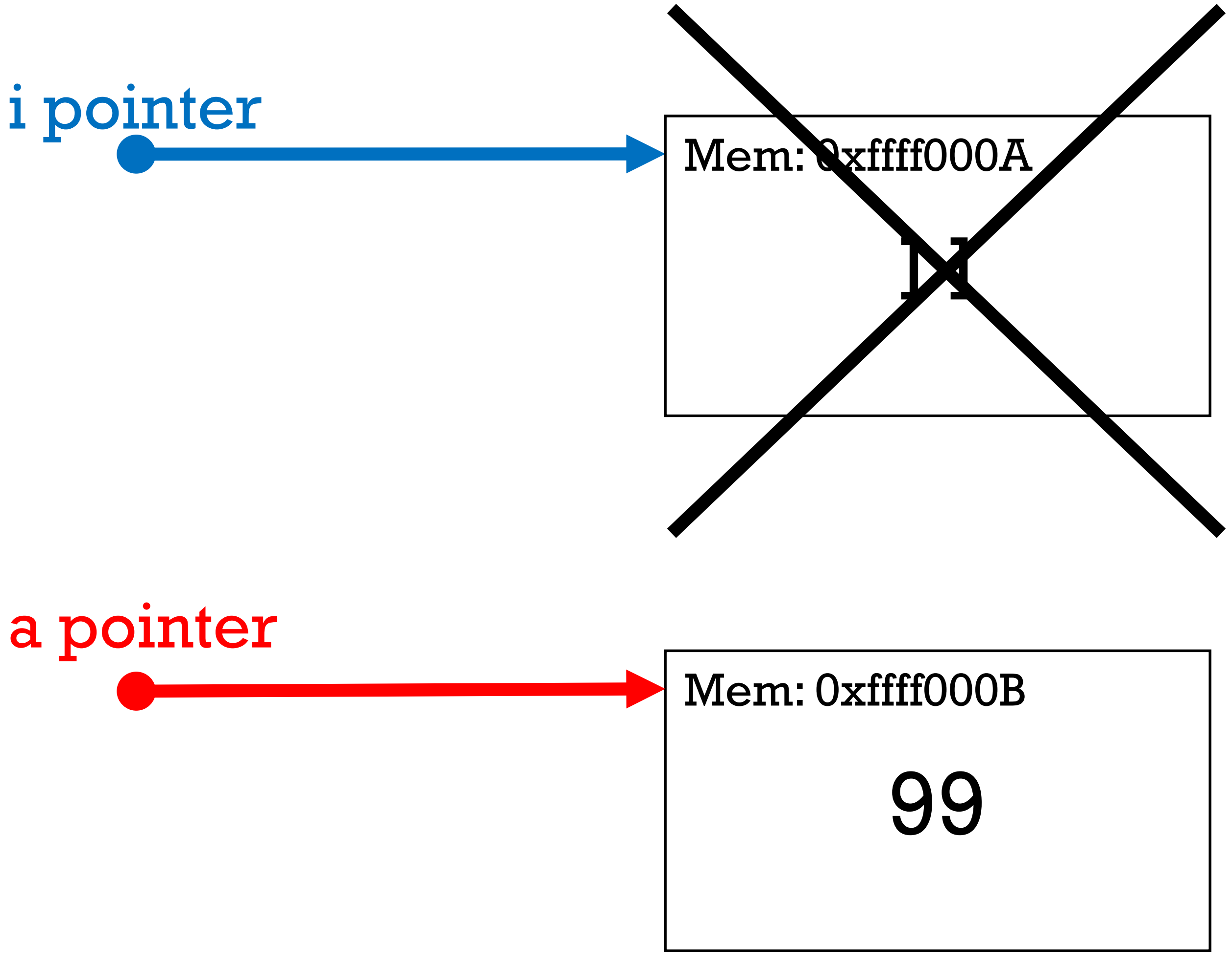


Memory leak solution

int *i = new int{11};

int *a = new int{99};

delete i; //deletes memory i pointing at



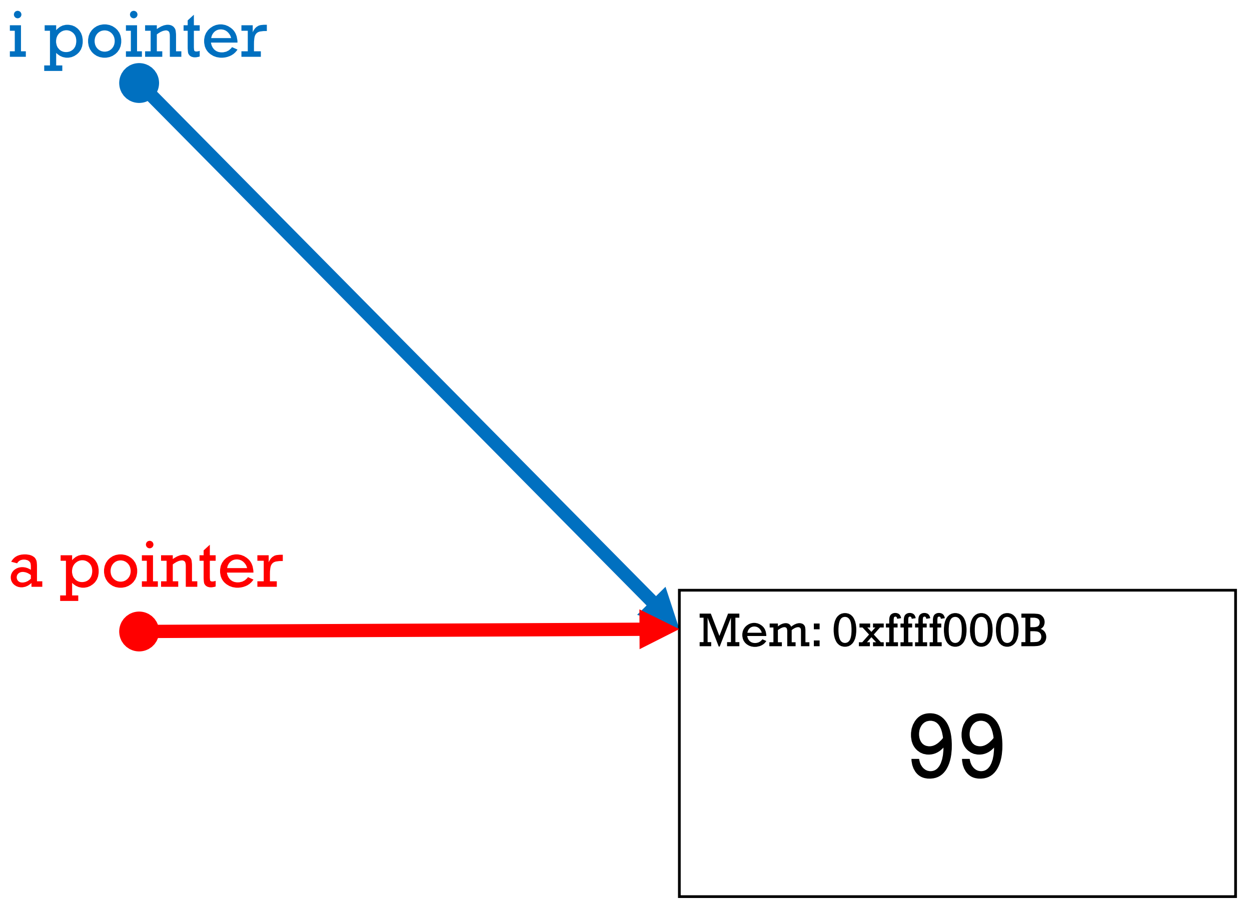
Memory leak solution

int *i = new int{11};

int *a = new int{99};

delete i; //deletes memory i pointing at

i = a; //i can now safely point to something else



THE C++ VECTOR

The C++ vector (think ArrayList)

- In **<vector>**
- A sequence container that **can change size** (like Java's ArrayList)
- Part of the STL (which we will cover in a few weeks)
- But for now it's very useful, even without knowing how to use its iterators
- http://www.cplusplus.com/reference/vector/vector/
- http://en.cppreference.com/w/cpp/container/vector

The C++ vector (think ArrayList)

- There are some very useful member functions:
  - **push_back(const T& value)** appends the given value to the end
  - **size()** //returns number of elements in vector
  - **operator[size_type pos]** returns a reference to the element at pos
  - **at(size_type pos)** returns a reference to the element at pos. Differs from operator[] by doing bounds check and throws exception
- We can use the for-each loop with the vector (it's called the **ranged-for** in C++)

The C++ vector (think ArrayList)

```
Vector <int> intVector;
intVector.push_back(5);
intVector.push_back(10);
intVector.push_back(15);
```

```
for(int i=0; i<intVector.size(); i++)
{
    cout << intVector[i]
}
```

The C++ vector (think ArrayList)

```
Vector <int> intVector;
intVector.push_back(5);
intVector.push_back(10);
intVector.push_back(15);
```

```
for(int i=0; i<intVector.size(); i++)
{
    cout << intVector[i]
}
```

```
for(int value: intVector)
{
    cout << value
}
```

vector.cpp

Agenda

1. Classes and objects
2. Default constructor
3. Default arguments
4. Forward declaration
5. Inline functions
6. Most vexing parse

COMP 3522

CLASSES AND OBJECTS

OOP in C++ (finally!)

- Let’s review some fundamental OOP concepts:
  - **Encapsulation**
  - **Abstraction**
  - **Inheritance**
  - **Polymorphism**

Encapsulation

- Process of combining data members & functions into a single unit called class
    - **make data members private**
    - **create public getter/setter functions**

```cpp
class Encapsulation
{
    private:
        int x;

    public:
        void set(int a)
        {
            x = a;
        }

        int get()
        {
            return x;
        }
};
```

```cpp
// main function
int main()
{
    Encapsulation obj;

    obj.set(5);

    cout<<obj.get();
    return 0;
}
```

Abstraction

- Only show relevant details to user and hide irrelevant details
- Abstraction in class using access specifiers
  - public, private
- Abstraction in header files
  - ie: pow() function in math.h
  - Don’t know how pow implemented in math, we just use it
  - cout << pow(7,3); //seven to the power of three = 343

Abstraction

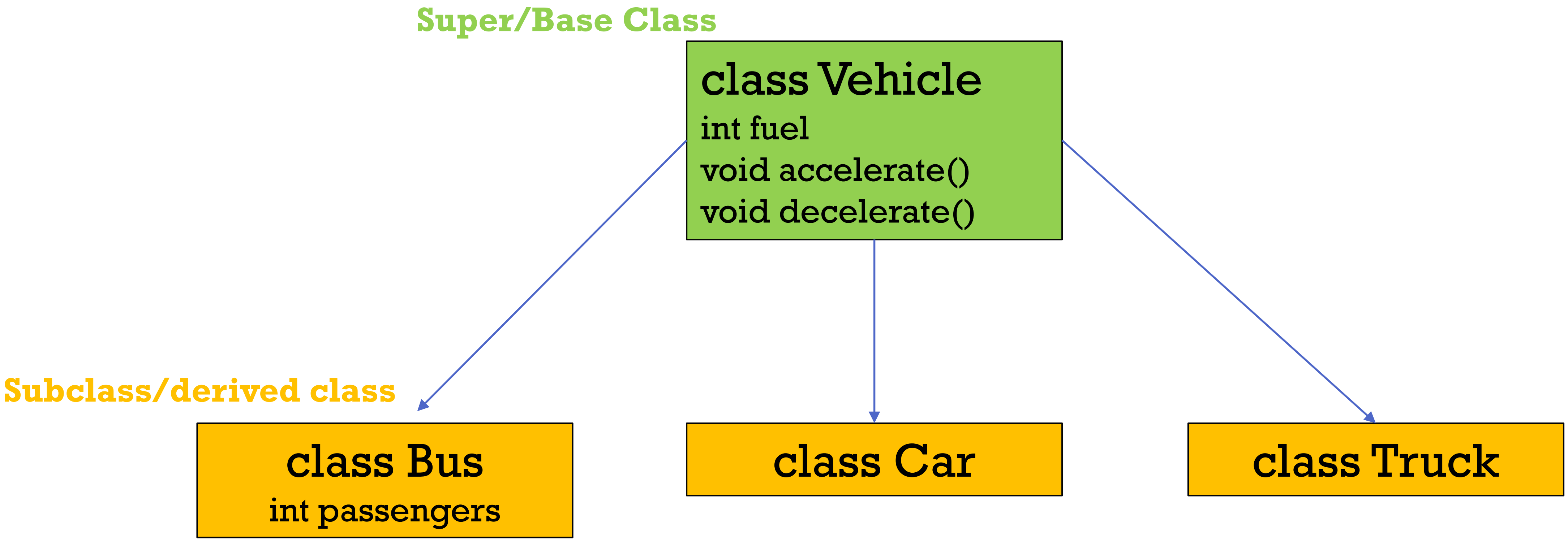
```cpp
class ImplementAbstraction
{
    private:
        int a, b;

    public:
        void set(int x, int y)
        {
            a = x;
            b = y;
        }

        void display()
        {
            cout<<"a = " <<a << endl;
            cout<<"b = " << b << endl;
        }
};
```

```cpp
int main()
{
    ImplementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

Inheritance

- Ability of a class to derive properties and characteristics from another class
- **Subclass/derived class** – the class that inherits properties from another class
- **Super/Base Class** – class whose properties inherited by subclass

Inheritance



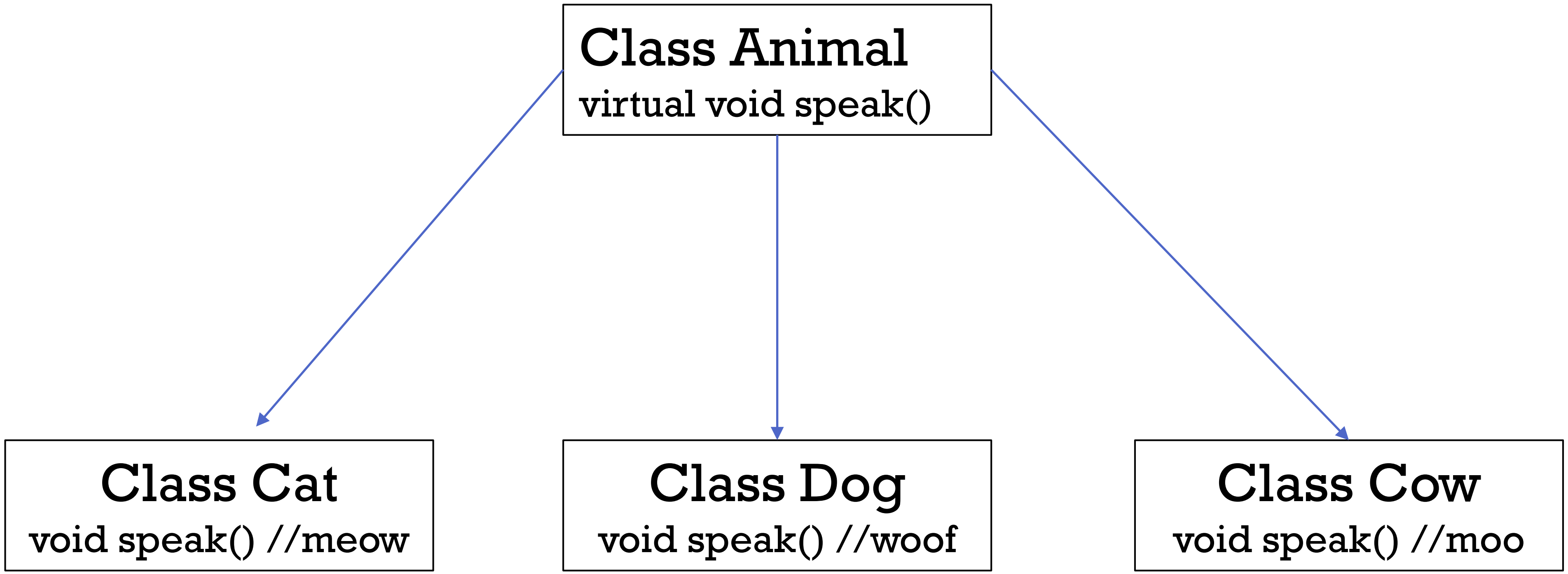
Inheritance

```
//Base class
class Vehicle
{
    public:
        int fuel;
        void accelerate();
        void decelerate();
};

// Sub class inheriting from Base Class(Parent)
class Bus : public Vehicle
{
    public:
        int passengers;
};
```

Polymorphism

- Having many forms
- Call to a member function will cause different function to be executed depending on the type of object invoked

Polymorphism



The C++ class

```
class Animal {
public:
    virtual void speak() {
        cout << "???" << endl;
    }
};

class Cat : public Animal {
public:
    void speak() {
        cout << "meow" << endl;
    }
};
```

…similar code for Cow and Dog

```
Cat cat;
Dog dog;
Cow cow;
Animal *a;

a = &cat;
a->speak(); //meow
a = &dog;
a->speak(); //woof
a = &cow;
a->speak(); //moo
```

The C++ class

- Defined using keyword **class** or **struct**
- A class defines a new data type that can contain:
  1. **Data** referred to as member variables or data members
  2. **Functions** referred to as member functions or (rarely) methods
  3. **Type definitions**
  4. **Contained classes**

C++ Accessibility

- Class members (data and functions) have visibility (just like Java!)
  - **public** members are accessible anywhere
  - **private** members are only accessible from within the class
  - **protected** members are accessible in the class and its subclasses (in C++ we call these derived classes)
- **By default, all class members have private access**
- Note: a struct and a class are the same thing in C++, except when we use the keyword "**struct" members get public access by default**!

Class example: Circle (part 1). Circle.hpp

```
class Circle
{
    private:
        double radius;
    public:
        void set_radius(int);
        double area(void);
};
```

Class example: Circle (part 2). Circle.cpp

```
void Circle::set_radius (int new_radius)
{
    radius = new_radius;
}

double Circle::area()
{
    return 3.14 * radius * radius;
}
```

Class example: Circle (part 3)

```
Circle my_first_circle;
my_first_circle.set_radius(2);
cout << my_first_circle.area() << endl;
```

We can also do this:

```
class Circle
{
        double radius;
    public:
        void set_radius(int);
        double area(void);
}my_circle;
```

We can also do this:

```
class Circle
{
        double radius;
    public:
        void set_radius(int);
        double area(void)
            {return 3.14 * radius * radius};
};
```

DEFAULT CONSTRUCTOR AND ARGUMENTS

Where's the constructor?

- We should probably add a constructor to our Circle class:

```
class Circle
{
        double radius;
    public:
        Circle(int); // No return type
        void set_radius(int);
        double area(void);
};
```

Where’s the constructor?

- Don’t forget to implement the constructor function

```
Circle::Circle(int r)
{
    radius = r;
}
```

But now that we have a constructor…

We **can't** do this anymore:

```
Circle constructed_with_default_ctor;
```

The compiler will complain that we don't have a default constructor (memories of Java…)

Let's overload our constructor!

```
class Circle
{
        double radius;
    public:
        Circle(); // No return type
        Circle(int); // No return type
        void set_radius(int);
        double area(void);
};
```

And add this...

```
Circle::Circle ()
{
    radius = 10; // Magic numbers are bad
                 // But this is a lecture
}
```

We should use “member initialization”

```
Circle::Circle(int r) : radius(r)
{
    // Possibly empty if there’s nothing else
    // to do
}
```

Be careful!

```
Circle my_circle; // Calls the default ctr

Circle my_circle(); // This is a function
                    // prototype. MOST VEXING PARSE

Circle my_circle{}; // Calls default ctr
```

Circle.cpp and Visibility.cpp

Did someone say complex numbers?

- Suppose we have a class representing a complex number
- A complex number has two parts:
    1. Real (r)
    2. Imaginary (i)

```
class complex
{
  private:
    double r, i;
  …
```

The complex number constructor

Here's a good first pass at the constructor:

```
public:
  complex(double rnew, double inew)
  {
    r = rnew;
    i = inew;
  }
  …
```

There's a problem, though

- The compiler wants to ensure that all member variables are initialized
- It generates a call to the default constructor for the members we don't initialize ourselves:

```
public:
  complex(double rnew, double inew) : r(), i()
  {
    r = rnew;
    i = inew;
  }
  …
```

This doesn’t always work

- For simple arithmetic types like int and double, it doesn’t really matter if we set their value in an initialization list or in the constructor body
  - Data members of fundamental types that do not appear in the initialization list remain uninitialized
- **There’s a problem with classes though:**
  - A member data item of a class type is implicitly default-constructed if it is not contained in the initialization list
  - In other words, the default constructor is called on class types if they’re not initialized in the initialization list

The complex number constructor part 1

We should always use the special C++ syntax called the member initialization list:

```
public:
  complex(double rnew, double inew) :
    r(rnew), i(inew) { }
  ...
```

The complex number constructor part 2

In C++, we can use the same identifiers for the constructor parameters and the class members:

- Names in the initialization list outside the parentheses refer to the members
- Inside the parentheses the names follow the scoping rules for a member function (names local to the member function including argument identifiers hide names from the class)

```
  …
private:
  double r, i;
public:
  complex(double r, double i) : r(r), i(i) { }
  …
```

The complex number constructor part 3

Let's create a second constructor where we set the imaginary part of the complex number to `0`

```
public:
  complex(double r, double i) : r(r), i(i) { }
  complex(double r) : r(r), i(0) { }
```

The complex number constructor part 4

We probably want a **default** constructor too:

```
public:
  complex(double r, double i) : r(r), i(i) { }
  complex(double r) : r(r), i(0) { }
  complex() : r(0), i(0) { }
```

Too much! How can we simplify this?

Default arguments!

We can reduce code duplication and complexity by including **default arguments**

```
public:
  complex(double r = 0, double i = 0)
    : r(r), i(i) { }
  …
```

Default arguments

- Can be provided **for trailing arguments only**:

```
int f(int, int = 0, char * = nullptr); // OK
int g(int = 0, int = 0, char *);        // ERROR
int h(int = 0, int, char * = nullptr); // ERROR

// Space between * and = is needed!
int creates_error(char *= nullptr);     // ERROR
```

C++ constructor style note

- **Data members MUST be initialized in the order in which they are declared in the class**
- The compiler may emit a warning if we don't respect this recommendation
- In C++ the **order of class/struct member initialization is determined by the order of member declaration** and not by the order of their appearance in member initialization list.
- Avoid generating this warning

https://stackoverflow.com/questions/24285112/why-must-initializer-list-order-match-member-declaration-order

complex.cpp

Default constructor (a close analysis)

- A constructor
  - no arguments, or has default values for every argument
- Not mandatory, but we should define one whenever possible
  - It is cumbersome (as we will see) to implement containers (lists, trees, matrices) of types that don’t have default constructors
  - Eliminates the possibility of uninitialized variables of a type
  - Variables initialized in an inner scope that exist for algorithmic reasons in an outer scope must already be constructed with a meaningful value

Ask yourself: does this type have a ‘special’ value or state we can ‘naturally’ use as a default?

And just to make things more exciting

- We can also assign default values to member variables
- When we do this, we only need to set values in the constructor that are different from the defaults
- The benefit is more pronounced in large classes

```
class complex
{
  private:
    double r = 0.0, i = 0.0;
    …
```

Member functions can be const

- We can add the const specifier to a member function prototype
- Specifies that the member function does not modify the object for which it is called
- Compiler will catch accidental attempts to violate this promise
- We should always use this with getters, for example:

```
double Cat::get_weight_grams() const
{
    return weight_grams;
}
```

Organizing our code

- Each unit of source code is typically split into:
  - Header file with declarations (.h or .hpp)
  - Source file (.cpp)
- The header file contains declarations of functions and classes
- Declarations tell the compiler that the code for the functions with the given signatures exists somewhere and that they can be called in the current compilation unit
- The source file contains the definitions (implementations) of the functions and classes declared in the header file

Q: Where do default argument values go?

In the function prototype in the header file

```
// Header file
void f(int x = 1, int y = 2);

// Source file
void f(int x, int y) { … }
```

FORWARD DECLARATION

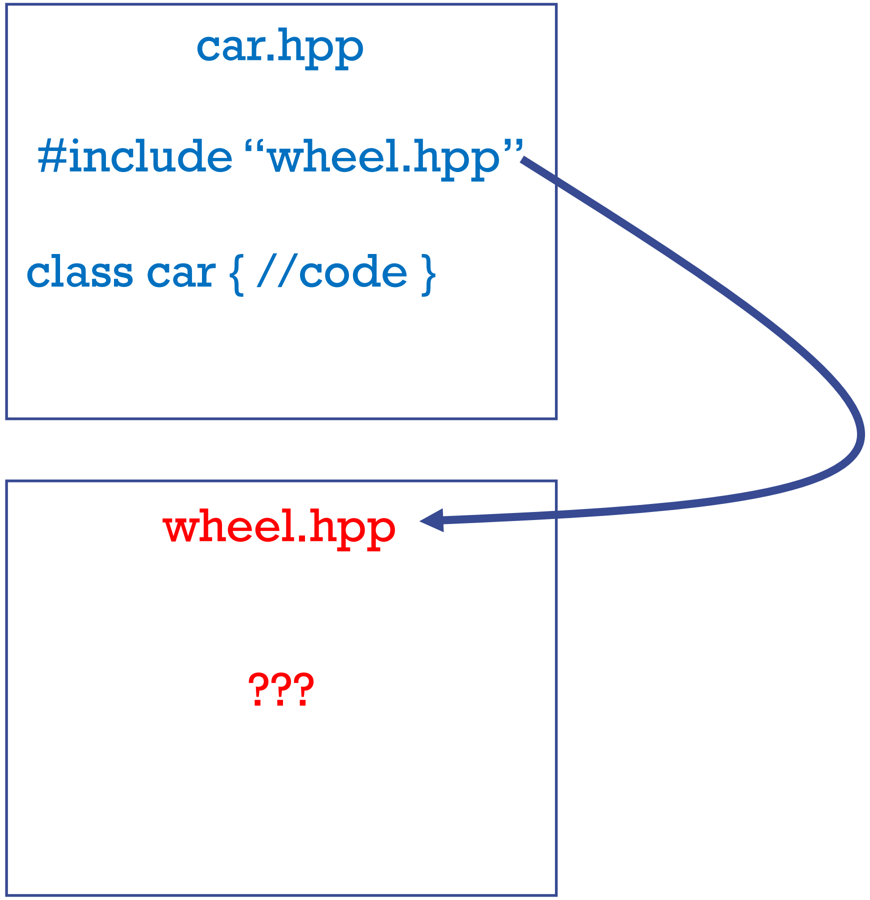
Forward declaration (motivation) (1 of 3)

- Our first C++ OOP conundrum

- Suppose we have a car class and a wheel class
    1. A car has wheels
    2. A wheel has a pointer to the car that possesses it

Forward declaration (motivation) (2a of 3)

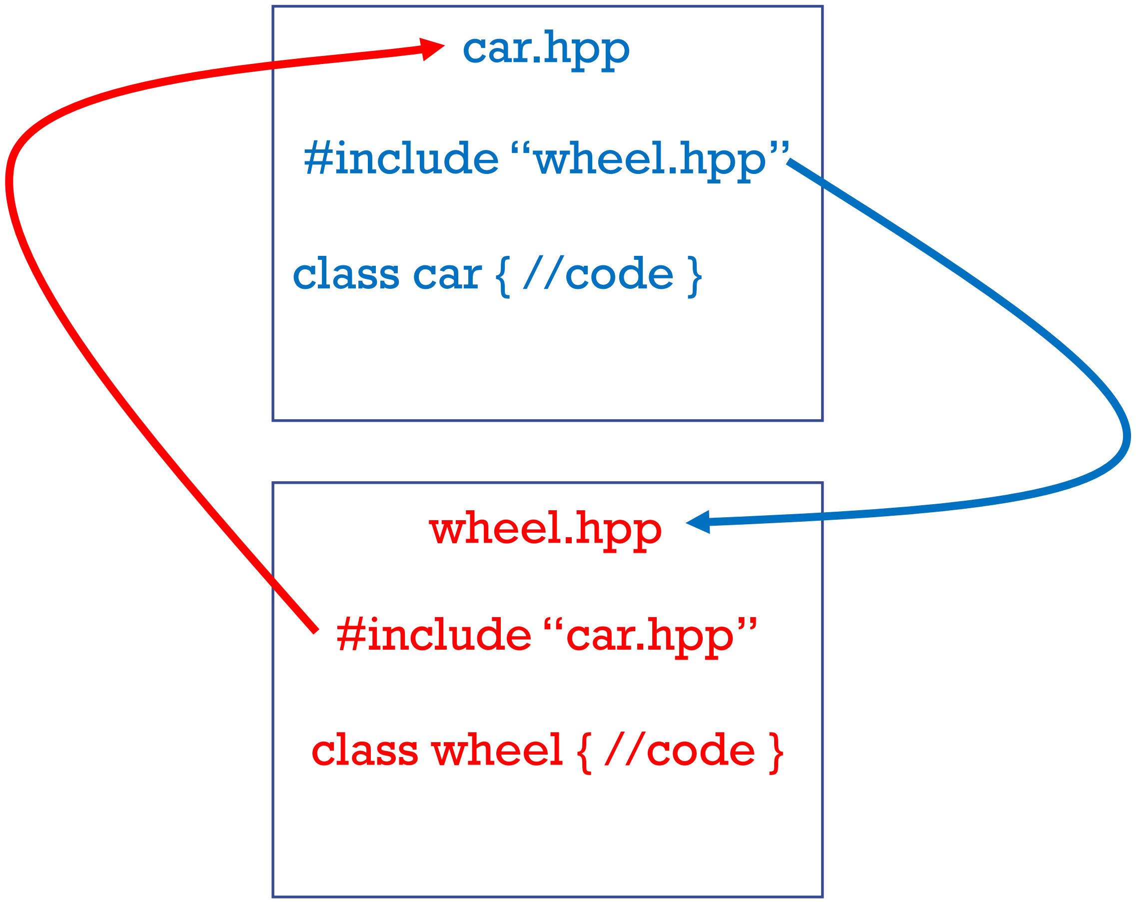
```
File car.hpp

#include "wheel.hpp"

class car
{
    wheel [] wheels;
}
```



Forward declaration (motivation) (2b of 3)

File wheel.hpp

```
#include “car.hpp” // UH OH!  THIS IS TROUBLE!

class wheel
{
    car * owner;
}
```



Forward declaration (motivation) (3 of 3)

- How do we include the car inside the wheel header file?

- If we #include "car.hpp", then we would have to insert the car.hpp file which includes the wheel.hpp file which includes the car.hpp file which includes the wheel.hpp file…
- The compiler error message is **not helpful**
- The solution is forward declaration

Solution: forward declaration!

```
File wheel.hpp

class car; // Forward declaration (so simple!)

class wheel
{
    car * owner; // Must be pointer or reference
}
```

Caution

- **If you use forward declaration, you can only declare a reference or a pointer to that type**
    - Compiler does not know how to allocate object
    - If you forget this, your code will not compile and you will see a **field 'class' has incomplete type error**.

INLINE FUNCTIONS

Motivation for inline functions (I of II)

Invoking a function can be **expensive**. Here is a partial list of things we have to do:

1. **Store** values used in registers by calling function
2. **Create** new activation record (activation frame) on the call stack
3. **Push** arguments to the frame for the new function call
4. **Execute** the new function, possibly calling yet another function
5. **Store** results (possibly)
6. **Restore** registers and pop the activation record…

Motivation for inline functions (II of II)

- That's a lot to do!
  - **Slow**
    - Doesn't use the CPU's **cache** very well
    - Can be very **wasteful** for small functions that are called frequently
  - *But* we want our functions to be *atomic* and *easy to maintain*.

- There is an easy solution:

Inline Functions!

Inline function (non-member function)

An inline function includes the inline specifier in the function prototype

1. Inline non-member function (not part of a class)

```
inline <return_type> <function_name> (<arguments>)
{
      …
};
```

Inline function (member function)

2. Inline member function (part of a class)

```
inline <return_type>
<class_name>::<function_name> (<arguments>)
{
      ...
};
```

Member functions are implicitly inline

```
class C
{
     public:
          void f() { … }  // Implicitly inline
          void g();
          void h();
}

inline void C::g() { … }   // Explicitly inline

// C.cpp
void C::h() { … }          // NOT inline
```

How does it work?

- **Copy and paste!**

- Just like a MACRO, except easier to debug
  - Breakpoints
  - Appears on call stack

- Macro vs inline: Use inline when we can!

Example 1

```
inline int divide(int a, int b)
{
	return a/b;
}
int main() {
	int num{2};
	int denom{2};
	int result = divide(num, denom);

	return 0;
}
```

```
int main()
{
	int num{2};
	int denom {2};
	int result =
		num / denom;
	return 0;
}
```

Example 2

```
class Cat
{
 int age_in_years;
   public:
     inline int get_age() const;
}
int Cat::get_age() const
{
  return age_in_years;
}
int main()
{
  Cat * my_cat = new Cat();
  int age = my_cat->get_age();
}
```

```
int main()
{
  Cat * my_cat =
           new Cat();
  int age =
           my_cat->
           age_in_years;
}
```

Caveats

1. The compiler is not obliged to inline your function
   - in fact the compiler may even inline some functions when you don't use the inline keyword
2. **<u>Implementation must be in the header file</u>**
   - when defining inline function in classes
   - the compiler needs to "see" what it is cutting and pasting
   - each cpp file is compiled separately, so when we compile foo.cpp which includes bar.hpp, the compiler doesn't know what's in bar.cpp
3. GREAT for small frequently-used functions (getters, setters, etc.)
4. BUT the executable increases in size due to code duplication!

Prefer inline functions vs macros

- A macro may evaluate its arguments more than once
- Inline functions perform type checking, macros do not

```
#define SQUARE(x) ((x)*(x))
inline int square(int x) { return x * x; }

int value = SQUARE(n++); // UNEXPECTED OUTPUT!
int value = square(n++); // This works
```

Inline.cpp

MOST VEXING PARSE

The Most Vexing Parse:

```
struct A
{
    void doSomething(){}
};

//what does this look like to you?
A functionA();
```

The Most Vexing Parse:

```
struct A
{
    void doSomething(){}
};

//what does this look like to you?
A functionA();

//How about this?
A objectInstance();
```

The Most Vexing Parse:

```
struct A
{
   void doSomething(){}
};

//what does this look like to you?
A functionA();

//How about this?
A objectInstance();

//How about this?
A a();
```

The Most Vexing Parse:

```
struct A
{
   void doSomething(){}
};
int main()
{
   A a();

   a.doSomething();
}
```

The Most Vexing Parse:

```
struct B
{
    B(int x){}
};
```

```
struct A
{
    A (B const& b){}
    void doSomething(){}
};
```

```
int main()
{
    A a(B(x));
    a.doSomething();
}
```

The Most Vexing Parse:

```
struct B
{
   B(int x){}
};

struct A
{
   A (B const& b){}
   void doSomething(){}
};

int main()
{
   A a(B(x)); //compiler sees something similar to:
              //A function_a (B arg_x);

   a.doSomething();
}
```

How to fix The Most Vexing Parse:

```
struct B
{
    B(int x){}
};
```

```
struct A
{
    A (B const& b){}
    void doSomething(){}
};
```

```
int main()
{
    A a(B{x});
    a.doSomething();
}
```

https://www.fluentcpp.com/2018/01/30/most-vexing-parse/

This is The Most Vexing Parse In C++:

```
Circle my_circle; // Calls the default ctr

Circle my_circle(); // This is a function
                    // prototype

Circle my_circle{}; // Calls default ctr
```