Week 13 - Monday

COMP 2400

Last time

- What did we talk about last time?
- Introduced C++

Questions?

Project 6

Quotes

If you think C++ is not overly complicated, just what is a protected abstract virtual base pure virtual private destructor, and when was the last time you needed one?

Tom Cargill

OOP in C++

Object Oriented Programming

- Let's see how objects work in C++ by looking at classically important elements of OOP
  - Encapsulation
  - Dynamic dispatch
  - Polymorphism
  - Inheritance
  - Self-reference

Encapsulation

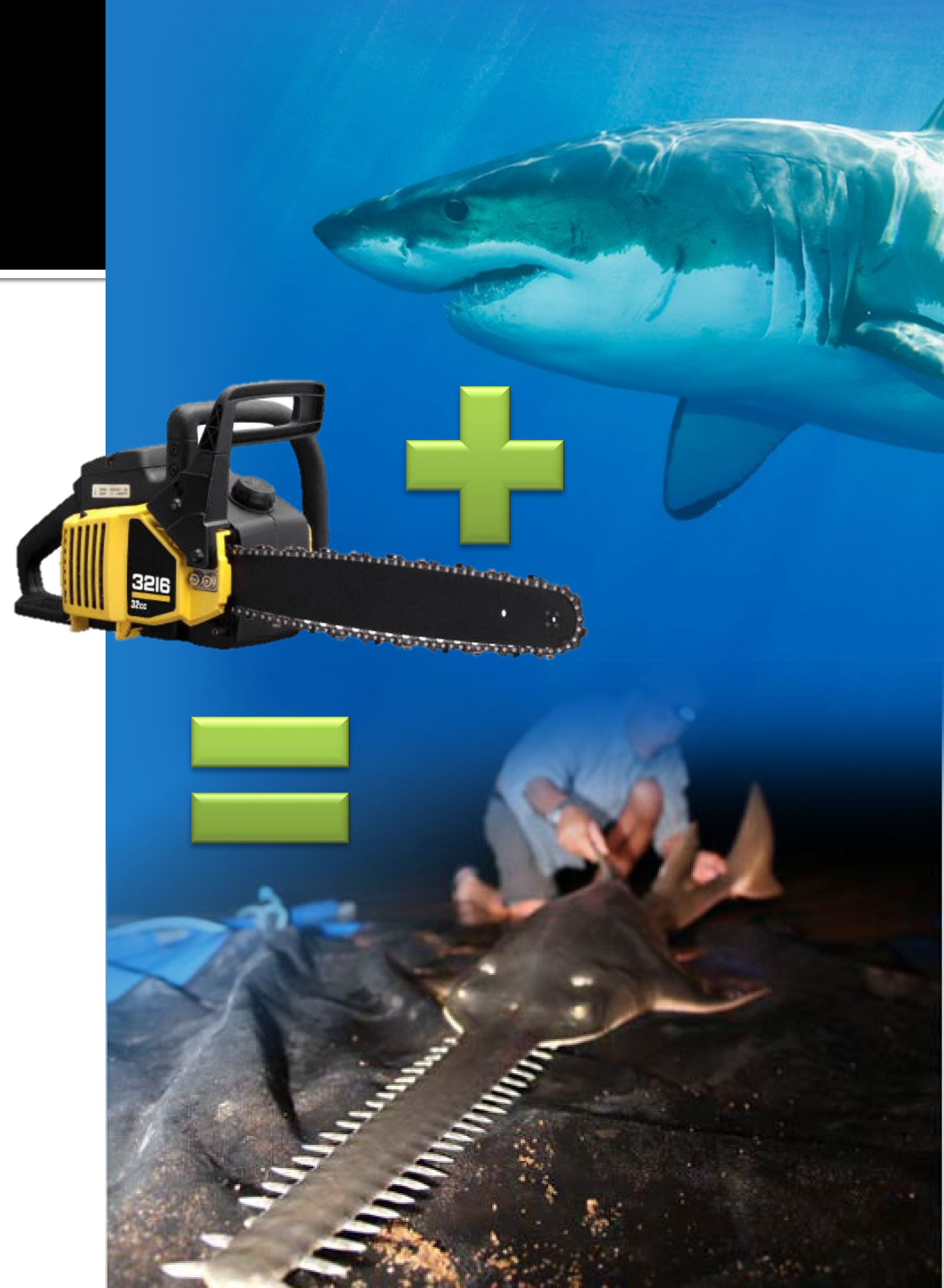
- Information hiding
- We want to bind operations and data tightly together
- Consequently, we don't want you to touch our privates
- Encapsulation in C++ is provided by the **`private`** and **`protected`** keywords
  - Unlike Java, you mark sections as **`public`**, **`private`**, or **`protected`**, not individual members and methods
- Hardcore OOP people think that **all** data should be private and most methods should be public

Encapsulation example

```
class A
{

private:
  int a;

public:
  int getA()
  {
     return a;
  }

  void setA(int value)
  {
     a = value;
  }
};
```

Inheritance

- Allows code reuse
- Is thought of as an "is-a" relationship
- C++ allows multiple inheritance, but you should only use it if you know what you're doing, usually as part of a design pattern
- Deriving a subclass usually means creating a "refined" or "more specific" version of a superclass



Inheritance example

```
class B : public A
{ //has member and methods from A
};

class C : public A
{
  private: //has A stuff and more
    int c;
  public:
    int getC(){ return c; }
    void increment() { c++; }
};
```

Polymorphism

- A confusing word whose underlying concept many programmers misunderstand
- Polymorphism is when code is designed for a superclass but can be used with a subclass
- If `AudiRS5` is a subtype of `Car`, then you can use an `AudiRS5` where you could use a `Car`

Polymorphism example

```
void drive( Car* c );
//defined somewhere
…
class AudiRS5 : public Car
{};
…
Car car;
AudiRS5 audi;
drive( &audi ); //okay
drive( &car );  //okay
```

Dynamic dispatch

- Polymorphism can be used to extend the functionality of an existing method using dynamic dispatch
- In dynamic dispatch, the method that is actually called is not known until run time

Dynamic dispatch example

```
class A {
  public: virtual void print()
  { cout << "A";}
};

class B : public A
{
  public: void print()
  { cout << "B";}
};
```

Dynamic dispatch example

```
A a;
B b;
A* p;

a.print();   // A
b.print();   // B

p = &a;
p->print(); // A
p = &b;
p->print(); // B
```

Self-reference

- Objects are able to refer to themselves
- This can be used to explicitly reference variables in the class
- Or, it can be used to provide the object itself as an argument to other methods
- Self-reference in C++ is provided in part through the **`this`** keyword
  - **`this`** is a pointer to the object you're inside of

Self reference example

```
class Stuff
{

private:
  int things;

public:
  void setThings(int things)
  {
     this->things = things;
  }
};
```

Self reference example

```
class SelfAdder
{
public:
  void addToList(List& list)
  {
    list.add(this);
  }
};
```

C++ Madness

Dividing up code

- In industrial-strength C++ code, the class declaration is usually put in a header file (`.h`) while the class definition is in an implementation file (`.cpp`)
- Benefits:
  - Easy to see members and methods
  - Header files can be sent to clients without divulging class internals
  - Separate compilation (faster)
  - Easier to take care of circular dependencies

Dividing up code header

```
class Complex
{
  double real;
  double imaginary;

public:
  Complex(double realValue = 0, double
  imaginaryValue = 0);
  ~Complex(void);

  double getReal();
  double getImaginary();
};
```

Dividing up code implementation

```
Complex::Complex(double realValue, double imaginaryValue)
{
  real = realValue;
  imaginary = imaginaryValue;
}

Complex::~Complex(void)
{}

double Complex::getReal()
{ return real; }

double Complex::getImaginary()
{ return imaginary; }
```

Overloading operators

- In C++, you can **overload operators**, meaning that you can define what + means when used with classes you design
- Thus, the following ***could*** be legal:

```
Hippopotamus hippo;
Sandwich club;
Vampire dracula = club + hippo;
```

Overloading operators

- But, what does it mean to "add" a **`Hippopotamus`** to a **`Sandwich`** and get a **`Vampire`**?
- Overloading operators is usually a bad idea
- You can get confusing code
- Most languages don't allow it
- It C++ it is useful in two cases:
  - To make your objects easy to input/output using **`iostream`**
  - To perform mathematical operations with numerical classes (like **`Complex`**!)

(Partial) overloading operators header

```
Complex& operator=( const Complex& complex );

Complex operator+( const Complex& complex ) const;

Complex operator-( const Complex& complex ) const;

Complex operator-() const;

Complex operator*( const Complex& complex ) const;
```

(Partial) overloading operators implementation

```
Complex& Complex::operator=
(const Complex& complex)
{
  real = complex.real;
  imaginary = complex.imaginary;

  return *this;
}
```

Programming practice

- Let's finish the **`Complex`** type
- Then, we can ask the user to enter two complex numbers
- We can do the appropriate operation with them

Upcoming

Next time…

- Templates
- STL

Reminders

- Keep working on Project 6
  - Due next Friday