README

- IF YOU KNOW SOMETHING, PUT IT HERE PLS
    - especially for the TODO things lol
- The midterm will cover lectures 1 through 9
    - TODO: include many people's notes from lectures 1 through 9
- To jump through the doc, do *View > Show document outline* if it's not already enabled
- Contacts:
    - Radish: @radix#9084
    - Aditya: @Yedi#4835
    - Marvin: @Mqrv#1626
    - *I nominate ~~Yedi and~~ Solidish Snake to put their info here as well*
- TODO remove comic sans bc ew :(
    - >:(-radish

Professor Gygi's study guide

1. What is the difference between a *class* and an *instance* of a class? Give an example.

A class is a user-defined type that contains (a) *data members* and (b) *member functions*. One example of a class is **Angle**:

```
class Angle {
  private:
    int theta;
  public:
    Angle(void);
    void set(int dt);
};
```

An instance of a class is one specific object.
Angle myAngle;   // myAngle is an object of type Angle

For instance, you could have a class **Dog** and an object of that class named **fido**. Doing an operation on Fido (like feeding him with **fido.feed()**) does not do that operation on all **Dog**s, only on **fido**.

2. What is information hiding, and how is it implemented in C++?

Information hiding refers to members that are placed under the key words **private** and **protected** in order to prevent unauthorized access.

Data encapsulation refers to designing an interface for editing objects so that you don't have to directly modify them. You group data members and member functions together and place them under the keywords public, private, and protected.

3. What is operator overloading, and how is it implemented in C++ (ch 12)?

Operator overloading is redefining certain operators so that they can be applied to objects of your own class types. Some operators including **. :: ? :** cannot be overloaded.

Syntax: **returnType operator<OPERATOR_SYMBOL>(argType arg1[, argType arg2])**

std::ostream& operator<<(std::ostream& out, const Fraction& frac) {...}

- This overloads the << operator so that you're allowed to do things like **std::cout << myFraction;** Otherwise, the compiler will error with a really long list of attempted overloads.

friend Fraction operator+(const Fraction& first, const Fraction& second) {...}

- This overloads the + operator so you can add two fractions together, without having to manually add them up by using the numerators and denominators.

4. What's the difference between a friend function and a member function?

Member functions are automatically passed the **this** object that they are called "on". If you define a member function to have 2 args, for instance, it'll actually be able to operate on 3 args, because the *object* you're calling this function *on* is implicitly passed to it. Keep in mind that member functions are part of a class, so if defining them outside of a class, you must namespace them appropriately.

Friend functions are normal free functions; they are passed what you pass to it. This is good for operations like adding/subtracting, because a member function that adds Fractions can only take a Fraction as the first arg (like **myFraction + 4)**. If the left value of the operator is something else like an int like **3 + myFraction;**, then the nonfriend operator overload will not be called and the compiler will complain. Unlike member functions, friend functions do not belong to a class. When defining them, do not namespace them with the class you declared them as a friend in.
TLDR: They are normal functions (NOT MEMBER functions) that have access to a specific class' private variables.

5. Differences in function calls:

```
void Shape::member(Shape s1, Shape s2);
void Shape::member(Shape *s1, Shape *s2);
void Shape::member(Shape& s1, Shape& s2) const;
void Shape::member(const Shape& s1, const Shape& s2);
void Shape::member(const Shape& s1, const Shape& s2) const;
```

Member function named member takes two arguments (objects s1 and s2, which are type Shape). 0

Same as above, except it takes two **Shape** pointers instead of two **Shape**s.

Member function takes two arguments (both references to objects s1 and s2 of type Shape), and the function is declared as a const function (doesn't allow to functions to modify the implicit this object, which they are called with)

Member function that takes two arguments (both references to const objects s1 and s2, which are of type Shape).

Member function takes two arguments (both references to const objects s1 and s2, which are of type Shape), and the member function is declared const (can be used with shape instances that are also declared const).

6. TODO Explain the & in lines 1 and 3 separately.

```
1 Value& operator=(const Value &rhs)
2 {
3 if (&rhs == this)
4 return *this;
5 v = rhs.v;
6 return *this;
7 }
```

In line 1, the & signifies that the `Value` instances returned and passed in will be references.

In line 3, the & is an operator that returns the address of the variable, `rhs`. This allows `this` and `rhs` to be compared, as they are now the same type (`Value*`).

From @36 on Piazza: The Value& means "reference to a value object" which is the return type. The "this" pointer contains the address of the current object (in this case the left hand side). Therefore, *this is the object itself, and it is appropriate to return *this as a reference to the current object. - This is a quote by Professor Gygi (Thank you Professor Gygi!)

7. Declare a member function f of class A, which takes a constant reference to an object of type A, returns an integer, and promises **not** to modify the instance of A calling the function f. Use the full definition of the function name, as it would appear in an implementation file, separate from the declaration of the class A .

int A::f(const A& thing) const {...}

from @36 on Piazza: The additional const at the end means it will not modify the state of an object it is called upon.

8. Find all errors in the following declaration:

```
1 class Thing
2 {
3 public:
4 char c;
5 int *p;
6 float& f;
7 Thing(char ch, float x) { c = ch; p = &x; f = x; }
9 }
```

- (Why would you declare private pointers and addresses? idk if it's wrong but it certainly seems weird to me idk)
- Should be **private** instead of **public** on line 3 (because data hiding)
- On line 7, the float reference `f` cannot be initialized as written. It must be in the member initialization list of the constructor:'
    - `Thing(char ch, float x) : f(x) {  ... }`
    - Note that this is undefined behavior. The variable x is a local variable, and its value will be undefined once the constructor returns.z
- In addition, the integer pointer p cannot be initialized with the address of a floating point without a cast.
- There is an error on line 6, you can’t initialize a reference. So the reference member should be initialized and declared at the same step.
- Missing semicolon on line 9 after the }

9. Explain why the explicit keyword may be needed when declaring a constructor.

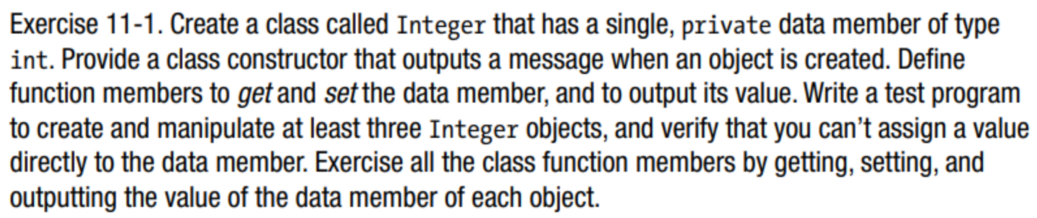
The **explicit** keyword may be needed when declaring a constructor to prevent the compiler from implicitly converting one data type to the class type of the constructor. This may be useful to reduce unexpected behavior, as some constructors may have side effects (e.g. changing a global variable, printing to stdout, or writing to a file).

10. Assuming that a class Object has been defined, write C++ statements that invoke the following kinds of operators:

Copy constructor:	**Object thing(oldThing);**
Assignment operator:	**Object newThing = existingThing; // idk**
Default constructor:	**Object thing;**

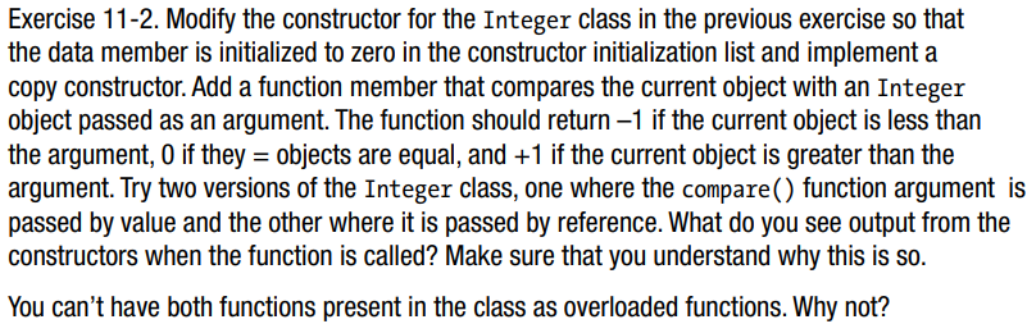
11. Textbook exercises

11.1


Exercise 11-1. Create a class called Integer that has a single, private data member of type int. Provide a class constructor that outputs a message when an object is created. Define function members to *get* and *set* the data member, and to output its value. Write a test program to create and manipulate at least three Integer objects, and verify that you can't assign a value directly to the data member. Exercise all the class function members by getting, setting, and outputting the value of the data member of each object.

```
class Integer {
   private:
      int num;
   public:
      Integer(void) { num = 0; std::cout << “message”; }
      int get(void) { return num; }
      void set(int i) { num = i; }
};



int main () {
   Integer obj1, obj2, obj3;     // beware messing with pointers/refs here;
                                 // but declaring all Integers in the same
                                 // line should be ok
   obj1.set(5);                  // ok
   std::cout << obj1.get();      // prints 5
   std::cout << obj2.get();      // prints 0 because that's default
   obj1.num = 5;                 // this won’t compile bc num is private

}
```

11.2

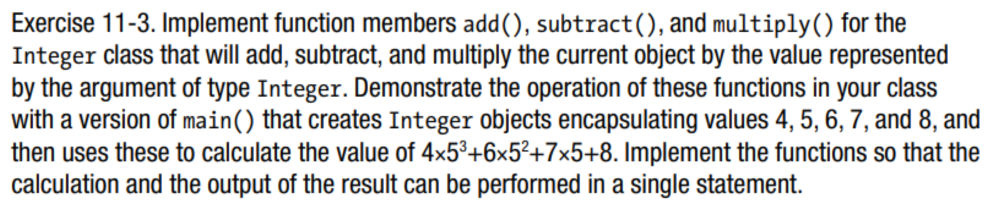


Exercise 11-2. Modify the constructor for the `Integer` class in the previous exercise so that the data member is initialized to zero in the constructor initialization list and implement a copy constructor. Add a function member that compares the current object with an `Integer` object passed as an argument. The function should return –1 if the current object is less than the argument, 0 if they = objects are equal, and +1 if the current object is greater than the argument. Try two versions of the `Integer` class, one where the `compare()` function argument is passed by value and the other where it is passed by reference. What do you see output from the constructors when the function is called? Make sure that you understand why this is so.

You can't have both functions present in the class as overloaded functions. Why not?

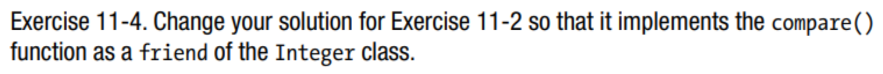
```
Integer(void) : num(0) { std::cout << “message”; }
Integer(int i) { num = i; } <- can someone double check this
Integer(const Integer& num2);
int compare(Integer& obj);

Integer::integer(const Integer& num2) {
     Integer(num); ←-- idk if this is right
}

// pls check this whole thing
int Integer::compare(Integer& num2) {
     if (this->num < num2.num) {
          return -1;
     }
     else if (this->num == num2.num) {
          return 0;
     }
     else if (this->num > num2.num) {
          return +1;
     }

}
```

11.3

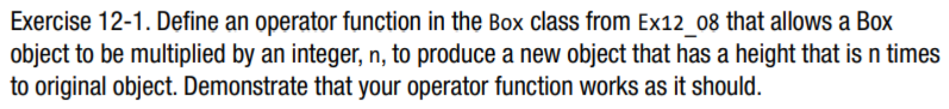

Exercise 11-3. Implement function members `add()`, `subtract()`, and `multiply()` for the `Integer` class that will add, subtract, and multiply the current object by the value represented by the argument of type `Integer`. Demonstrate the operation of these functions in your class with a version of `main()` that creates `Integer` objects encapsulating values 4, 5, 6, 7, and 8, and then uses these to calculate the value of $4\times5^3+6\times5^2+7\times5+8$. Implement the functions so that the calculation and the output of the result can be performed in a single statement.

```
Int Integer::add(const Integer& x) { return (this->num += x.num);}
Int Integer::subtract(const Integer& y) {return this->num - y.num;}
Int Integer::multiply(const Integer& z) {return this->num * z.num};

Int main () {
        Integer a;
        a.set(10);
        a.add(4);

}
```

11.4


Exercise 11-4. Change your solution for Exercise 11-2 so that it implements the `compare()` function as a `friend` of the `Integer` class.

12.1


Exercise 12-1. Define an operator function in the `Box` class from `Ex12_08` that allows a Box object to be multiplied by an integer, `n`, to produce a new object that has a height that is n times to original object. Demonstrate that your operator function works as it should.

12. Review the meaning of the options -g -o -c -Wall used with the g++ compiler

-g : request debug symbols for the output file
-o : change the name of the output file
-c: only compile the file(s), don't link them. outputs a .o file that can be used to link to other .o files later.
-Wall: "warn all": enable more warnings than usual. warnings can lead to unexpected or undefined behavior, so it is best to use this flag so that the

compiler can warn you before you spend too much time trying to figure out the issue in a debugger.

Lectures

Lectures 1 and 2

1. **make** command: builds a program by compiling all source files into object files and then linking all the object files to produce an executable.
    a. One .o file produced for each .cpp file
- Dependencies are described in the makefile. make takes into account all dependencies between files

Distinctions between source file, object file, & executable file.
- Source code: code that we write (.cpp files, .h files, etc)
- Object file (.o) : generated after compiling source code
- Exec file (.exe): file generated after linking set of .o files together with a linker

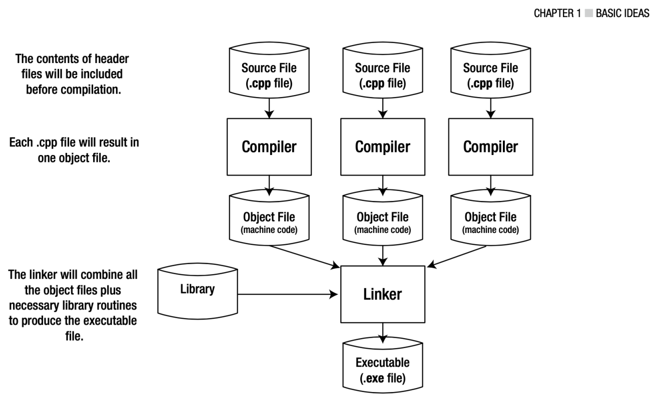

CHAPTER 1 BASIC IDEAS
The contents of header files will be included before compilation.
Source File (.cpp file)
Source File (.cpp file)
Source File (.cpp file)
Each .cpp file will result in one object file.
Compiler
Compiler
Compiler
Object File (machine code)
Object File (machine code)
Object File (machine code)
The linker will combine all the object files plus necessary library routines to produce the executable file.
Library
Linker
Executable (.exe file)

Figure 1-2. The compile and link process

2. Tar file: file containing an archive of several files and/or directories
    a. Archive:
    b. Create tar file: tar -cvf name_of_archive.tar file1 file2 dir1
        i. -c -> Create an Archive
        ii. -v -> Verbose Mode
        iii. -f -> Ability to specify file name
    c. Extract files from archive: tar -xvf name_of_archive.tar
        i. -x -> Extract
    d. Read tar manual page (list of commands)
        i. $ man tar
3. **Preprocessing directive**: source code gets modified before it is compiled to executable form

a. Ex: #include <iostream> , #ifndef, #endif
Header file content inserted in place of the #include directive before compilation

b. Using directive : imports all the names from a namespace, so that you don't have to qualify any name defined in that namespace each time that you have to use it

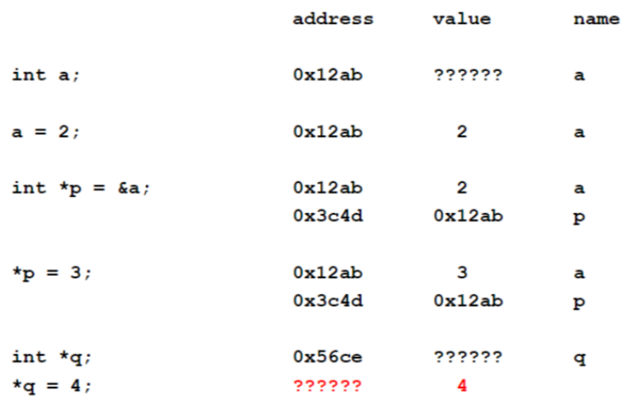
   i. Ex: using namespace std;
   ii. Using an entire namespace can lead to subtle errors (using.cpp's)

4. Namespace:
5. Stream:
6. Endl vs \n : both create a newline, however endl also flushes the stream, and therefore is slower.

- Stream insertion operator (<<) ex: cout <<
- Stream extraction operator (>>) ex: cin >>
- Not necessary to return 0 in main

Lecture 3

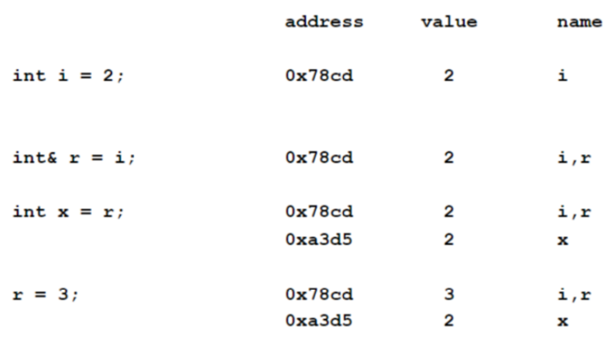
Pointers (review):



	address	value	name
`int a;`	0x12ab	??????	a
`a = 2;`	0x12ab	2	a
`int *p = &a;`	0x12ab	2	a
	0x3c4d	0x12ab	p
`*p = 3;`	0x12ab	3	a
	0x3c4d	0x12ab	p
`int *q;`	0x56ce	??????	q
`*q = 4;`	??????	4	

- Pointer: var that stores memory address of another variable
    - *p : pointer dereference
    - &a : address of var a
    - Dangerous to declare pointer var without initialization
- Stack memory: temporary storage space. Place where local variable and arguments are stored. Deleted automatically when function completes
    - Good way to remember: "t" in stack stands for temporary
- Heap memory (dynamically allocated space). Stuff on the heap stays until you free the space

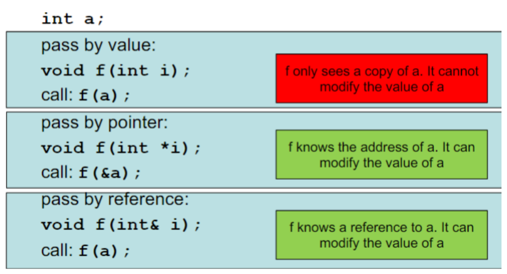
References (not in C)



	address	value	name
`int i = 2;`	0x78cd	2	i
`int& r = i;`	0x78cd	2	i,r
`int x = r;`	0x78cd	2	i,r
	0xa3d5	2	x
`r = 3;`	0x78cd	3	i,r
	0xa3d5	2	x

- Alternative name for an object
- Syntax sugar for pointers, with stricter semantics
- Safer version of pointers. Compiler will error on uninitialized references, whereas programs with pointers uninitialized will not be checked, and compile correctly (potential hard-to-find bugs)
- Var r and i both refer to the same value, and hold the same memory address. Therefore any change to the value of r will auto change the value of i, and vice versa
- Can use references to pass parameters to function
    - Void fr (int& r)

3 ways to pass parameters to a function



`int a;`

pass by value:
`void f(int i);`
call: `f(a);`
f only sees a copy of a. It cannot modify the value of a

pass by pointer:
`void f(int *i);`
call: `f(&a);`
f knows the address of a. It can modify the value of a

pass by reference:
`void f(int& i);`
call: `f(a);`
f knows a reference to a. It can modify the value of a

Const and Pointers

Int a = 4;
Int b = 6;

1. Pointer to const int | const int *p = &a;
    a. allowed to change what the pointer (p) points to. But you can’t dereference pointer p and change the value of whatever the pointer points to (int a) (in other words: not allowed to modify *p, but can modify p)
        i. Ex: compiler will reject: *p = 8 and *p = b;
        ii. Ex: compiler will accept: p = &b;
2. Constant pointer to int | int * const p
    a. Allowed to modify *p , but not p.
        i. Ex: compiler will accept: *p = 8
        ii. Ex: compiler will reject: p = &b;
3. Const pointer to const int | const int * const p;
    a. Cannot modify p or *p.

Const and References

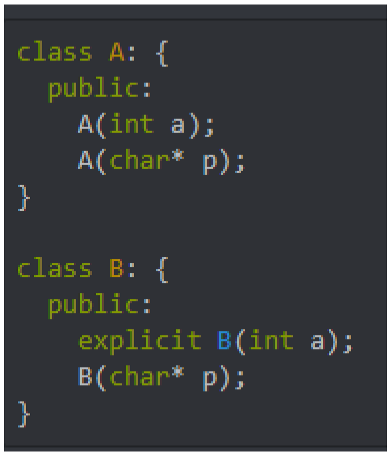
1. Const ref to int | void f(const int& x)
    a. Function f cannot modify x

Classes

- User-defined type
- Classes are defined to encapsulate data and functions (basically the implementation details of the class are kept hidden from user)
- Class include:
    - Member functions: functions that may act upon data members
    - Data members: any collection of data
    - Public and private allows you to control who/what can have access to the data members or member functions
- private members: only accessible from member functions of the class themselves.
- Public members: member variables accessible by any code with an available instance.

Random Tips

1. Friend functions get access to private values of the class.
2. Explicit vs Implicit
    a. Explicit means we tell the compiler NOT to implicitly convert functions. For example;



```
class A: {
  public:
    A(int a);
    A(char* p);
}

class B: {
  public:
    explicit B(int a);
    B(char* p);
}
```

        i. In this example above, if we try to do “A a = ‘x’”, it will convert the ‘x’ to an int, then create the object. We don’t want that. Therefore we use the explicit keyword, as shown in B. This prevents implicit conversion between types.