```
// __class declaration/definition__
// (think Java interface)
class Rectangle {
  // access specifier
  // can be set in any order
  // if not specified
  // default is `private`
  private:
    double width;
    double length;
  public:
    // set Functions (accessors)
    void setWidth(double width);
    void setLength(double length);
    // get Functions (mutators)
    // const specifies fn will not
    // change data in calling obj
    double getWidth() const;
    double getLength() const;
    double getArea() const;
}
```

```
  // __implementation__
  // `::` = "scope resolution operator"
  int Rectangle::setWidth(double w) {
    width = w;
  }
```

```
  // __instantiation/instance definition__
  Rectangle box;
  double rectWidth = 10.0;
  double rectLength = 20.0;
  box.setWidth(rectWidth);
  box.setLength(rectLength);
```

```
  // Define Rectangle pointer
  Rectangle *rectPtr = nullptr;
  // dynamically allocate Rect obj
  rectPtr = new Rectangle;
  rectPtr->setWidth(10.0);
  rectPtr->setLength(20.0);
  // delete obj from mmry
  delete rectPtr;
  rectPtr = nullptr;
```

```
    #ifndef RECTANGLE_H
    #define RECTANGLE_H
    // inline member functions
    // def. in class declatration
    // good for short fn bodies
    class Rectangle {
      private:
        double width;
      public:
        void setWidth(double);
        double getWidth() const {
          return width;
        }
    }
    #endif
    // - reg fns: compiler stores return address of
call, allocates memory for local vars etc.
    // - inline fns: code is copied into prog. in
place of call - larger exe, faster execution
```

```
// __Constructors__
// - memb fn automatically called when obj created
// - purpose to construct obj
// - fn name = class name
// - no return type
class Rectangle {
  private:
    double width;
  public:
    // default constructor
    Rectangle();
}
// implementing default constructor
Rectangle::Rectangle() {
  width = 0.0;
  height = 0.0
}
void Rectangle::SetWidth(double w) {
  if(w > 0)
    width = w;
  else {
    cout << "Invalid width \n";
    exit(EXIT_FAILURE);
  }
}
// calling default constructor
Rectangle rect;
```

```
// __constructor default args__
// prototype
Rectangle(double, double);
// definition/implementation
Rectangle::Rectangle(double w, double len) {
  width = w;
  len = len;
}
// calling
Rectangle rect(10, 5);
// if all constructor's params have
// default args it is default constructor
Rectangle(double = 0, double = 0);
// will return a rectangle of {0,0}
Rectangle r;
// when all constructors require args
// class has no default constructor
// no args == error!
// don't provide more than one default constructor!
// won't compile!
Square();
Square(int = 0);
```

```
// __destructors__
// - memb fn auto called when obj destroyed
// name = ~classname
// no return type, no args
// cannot be overloaded (1 per class)
// if constr. alloc. dynamic mem. destruct. should
release
class InventoryItem{
  private:
    char *description;
  InventoryItem(char *desc) {
    // allocate just enough mem for description
    description = new char[strlen(desc) + 1];
    // copy desc into mem
    strcopy(description, desc);
    // destructor
    ~InventoryItem() {
      delete [] description;
    }
    // getter
    const char *getDescrition() const {
      return description;
    }
  }
}
```

```
// __dynamically alloc. objs__
// - when obj is dyn. alloc. with the `new` operator,
its constructor executes
Rectangle *r = new Rectangle(10, 20);
// - when destroyed, destructor executes
delete r;
```

```
// __private member fns__
// can only be called by another memb.fn
// used 4 internal processing by class, not outside
class
```

```
// __instance and static members__
// instance var: memb var in a class. Each has own
copy.
// static var: one var shared among all objs of class
// static memb fn: can be used to access static memb
var
// can be called before any objs dfn
class Tree {
  private:
    // declaration in class
    static int objectCount;
  public:
    Tree() {
      objectCount++;
    }
    int getObjectCount() const {
      return objectCount;
    }
};
// definition of static memb var (outside class)
int Tree::objectCount = 0;
// `main fn`
int main () {
  Tree oak;
  Tree elm;
  Tree pine;
  cout << pine.getObjectCount; // 3
}
```

```
// __static memb fn__
// declared with static before rtrn type
// can only access static memb data
// can be called independent of objs
```

```
// __friends of classes__
// - a fn or class that is not a memb of a
//   class, but has access to private membs
//   of the class
// - can be stand-alone fn or memb of other class
// - declared with `friend` keyword in proto.
friend void setAVal(intVal&, int); // stand-alone
friend void SomeClass::setNum(int num); // memb fn
class FriendClass {
  // will be friend
};
class OtherClass {
  public: // declare entire FriendClass
    friend class FriendClass;
    // as class of this class
};
```

```
// __memberwise assignment__
// can use `=` to assign one obj to another
// or to init an obj w/an obj's data
int main() {
  Rectangle box1(10.0, 10.0);
  Rectangle box2(20.0, 20.0);
  box2 = box1;
};
```

```
// __copy constructors__
// - special constr. used when a new obj is
//   initialized to data of other obj of same class
// - default copy constr. copies field-to-field
// - default works fine in many cases
// - default = problem when obj contains pointer
//   - membs will point to same dyn. mem.
//   - solve by defining own copy constr.
//             - takes ref param to obj of class
SomeClass::SomeClass(const SomeClass &obj) {
  value = new int;
  *value = obj.value;
}
```

```
// - since copy constr. has ref. to obj copying from
//   it can modify that obj
// prevent this by making param `const`
SomeClass::SomeClass(const SomeClass &obj) {}
// definition
class StudentTestScores {
  private:
    double *testScores; // points to arr. of scores
    int numTestScores;
    void createTestScoreArray(int size) {
      numTestScores = size;
      testScores = new double[size];
      for(int i = 0; i < size; i++) {
        testScores[i] = DEFAULT_SCORE;
      }
    }
  public:
    StudentTestScores(int numScores) {
      createTestScoreArray(numScores);
    }
    // copy constr. -> uses `const` in param so not
same mem.
    StudentTestScores(const studentTestScores &obj) {
      numTestScores = obj.numTestScores;
      testScores = new double[numTestScores]
    }
    // destrctr.
    ~StudentTestScores() {
      delete [] testScores;
    }
};
```

```
// __operator overloading__
// - name of fn. overload = `operator` followed by
the op.
// - proto. goes in class declaration
// - fn. def. goes w/other memb.fn.
    void      operator=(const SomeClass &rhs);
// rtrn type  fn.name     param for obj on right
// - can be invoked as memb.fn.
obj1.operator=(obj2);
// - or in more conventional way
obj1 = obj2;
```

```
// can return value
class Point2d {
  private:
    int x, y;
  public:
    double operator-(const pointwd &right) {
      return sqrt(pow((x-right.x),2) + pow((y-
right.y), 2));
    }
};
```

```
// __operator overloading cont__
// rtrn.type same as left operand supports:
object1 = object2 = object3;
// fn.decl:
const SomeClass operator=(const someClass &rval);
// include as last statement:
return *this;
// - can change meaning of operator
// - cannot change num. of operands
// - cannot overload
?:  . .*  ::  sizeof
// - `++, --` overloaded differently for
prefix/postfix notation
// - overloaded relationals should return `bool`
// - overloaded stream operators `>>, <<` must return
ref. to
//   `istream, ostream` objs and take `istream,
ostream` as params
```

```
// __overloaded `[]` operator
// - can create classes that behave like arrays,
provide bounds-checking
//   on subscripts
// - must consider constructor/destructor
// - returns ref. to obj, not obj itself
```

```
// __this pointer__
//  - avail. to memb.fns
//  - always points to instnc. of class whose fn.
called
//  - passed as hidden arg. to all non-static
memb.fns
student2.getStudentName(); // `this` points to
student2
```

```
// __overloaded operators as member functions__
void printLine(ostream& out, int n){
  char ch = out.fill();
  out << setfill('-') << setw(n) << "-" <<
setfill(ch) << endl;
  return;
}
class Boolean{
  public:
  Boolean(int = 0);             // default
constructor
  void setValue(int x);         // mutator
  int getValue() const;         // accessor
  Boolean operator!() const;    // overloaded
operator for unary logical not
  // overloaded operator for binary logical and
  Boolean operator&&(const Boolean& rhs) const;
  private:
  int value;
};
Boolean::Boolean(int x){
  setValue(x);
}
void Boolean::setValue(int x){
  value = (x == 0) ? 0 : 1;
}
int Boolean::getValue() const{
  return value;
}
Boolean Boolean::operator!() const{
  Boolean temp;
  temp.setValue(1 - this->getValue());
  return temp;
}
Boolean Boolean::operator&&(const Boolean& rhs)
const{
  Boolean temp;
  temp.setValue(this->getValue() * rhs.getValue());
  return temp;
}
int main(){
  Boolean a, b;
  printLine(cout, 30);
  cout << "  a       b       !a    a && b" << endl;
  printLine(cout, 30);
  for (int i = 0; i <= 1; i++){
    b.setValue(0);
    for (int j = 0; j <= 1; j++){
      cout << setw(5) << ((a.getValue() == 0) ?
"false" : "true") << "   "
           << setw(5) << ((b.getValue() == 0) ?
"false" : "true") << "   "
           << setw(5) << ((!a.getValue() == 0) ?
"false" : "true") << "   "
           << setw(6) << (((a && b).getValue() == 0)
? "false" : "true")
           << endl;
      b = !b;
    }
    a = !a;
  }
  printLine(cout, 30);
  return 0;
}
```

```
// __overloaded operators as friend functions__
void printLine(ostream& out, int n){
  char ch = out.fill();
  out << setfill('-') << setw(n) << "-" <<
setfill(ch) << endl;
  return;
}
class Boolean{
  friend ostream& operator<<(ostream& out, const
Boolean& rhs);
  // overloaded operator for unary logical not as a
friend function
  friend Boolean operator!(const Boolean& rhs);
  // overloaded operator for binary logical and as a
friend function
  friend Boolean operator&&(const Boolean& lhs, const
Boolean& rhs);
  public:
  Boolean(int = 0);             // default
constructor
  void setValue(int x);         // mutator
  int getValue() const;         // accessor
  private:
  int value;
};
Boolean::Boolean(int x){
  setValue(x);
}
void Boolean::setValue(int x){
  value = (x == 0) ? 0 : 1;
}
int Boolean::getValue() const{
  return value;
}
Boolean operator!(const Boolean& rhs){
  Boolean temp;
  temp.setValue(1 - rhs.getValue());
  return temp;
}
Boolean operator&&(const Boolean& lhs, const Boolean&
rhs){
  Boolean temp;
  temp.setValue(lhs.getValue() * rhs.getValue());
  return temp;
}
ostream& operator<<(ostream& out, const Boolean&
rhs){
  char ch = out.fill();
  out << setw(5) << ((rhs.getValue() == 0) ? "false"
: "true");
  out.fill(ch);
  return out;
}
int main(){
  Boolean a, b;
  printLine(cout, 30);
  cout << "  a       b       !a    a && b" << endl;
  printLine(cout, 30);
  for (int i = 0; i <= 1; i++){
    b.setValue(0);
    for (int j = 0; j <= 1; j++){
      cout << a << "   "
           << b << "   "
           << (!a) << "   "
           << ' ' << (a && b)
           << endl;
      b = !b;
    }
    a = !a;
  }
  printLine(cout, 30);
  return 0;
}
```