# IMMUTABILITY, REFERENCE, AND INHERITANCE

## LECTURE 10-1

JIM FIX, REED COLLEGE CS2-S20

# TODAY'S PLAN

- OPERATOR METHODS AND FUNCTIONS FOR BINARY OPERATIONS
- PASSING BY REFERENCE
- IMMUTABILITY WITH `const`
- INHERITANCE
  - ACCOUNT EXAMPLES
  - DYNAMIC DISPATCH WITH `virtual`

## MORE ON OPERATIONS

I checked C++'s rules for operations and have two things to report:

- binary operators can be defined as functions or as methods
- operators **++** and **--** can be overloaded as both prefix and suffix

# OVERLOADING THE TIMES OPERATOR AS A FUNCTION

- Here is how we might overload **+** for class **`Rational`**:

```
class Rational {
private:
  int num;
  int den;
public:
  ...
  friend Rational operator*(Rational q1, Rational q2);
  ...
};
```

- Here is its implementation:

```
Rational operator*(Rational q1, Rational q2) {
  return Rational {q1.num*q2.num, q1.den*q2.den};
}
```

- Here is its use in a client:

```
Rational product = q1 * q2;
```

# OVERLOADING THE PREFIX INCREMENT OPERATOR

- This overloads the prefix form of **`++`** for a class **`Counter`**:

```
class Counter {
private:
  int value;
public:
  ...
  void operator++();
  ...
};
```

- Here is its implementation:

```
void Counter::operator++() {
  value = value + 1;
}
```

- Here is its use in a client:

```
Counter c;
...
++c;
```

# OVERLOADING THE POSTFIX INCREMENT OPERATOR

- This overloads the postfix form of **`++`** for a class **`Counter`**:

```
class Counter {
private:
  int value;
public:
  ...
  void operator++(int unused);
  ...
};
```

- Here is its implementation:

```
void Counter::operator++(int unused) {
  value = value + 1;
}
```

- Here is its use in a client:

```
Counter c;
...
c++;
```

# OVERLOADING THE POSTFIX INCREMENT OPERATOR

- This overloads the postfix form of **++** for a class **`Counter`**:

```
class Counter {
private:
  int value;
public:
  ...
  void operator++(int unused);
  ...
};
```

*This apparently tells C++ that we want the postfix form.*

- Here is its implementation:

```
void Counter::operator++(int unused) {
  value = value + 1;
}
```

- Here is its use in a client:

```
Counter c;
...
c++;
```

# OVERLOADING THE POSTFIX INCREMENT OPERATOR

- This overloads the postfix form of **++** for a class **`Counter`**:

```
class Counter {
private:
  int value;
public:
  ...
  void operator++(int unused);
  ...
};
```

*This apparently tells C++ that we want the postfix form.*

- Here is its implementation:

```
void Counter::operator++(int unused) {
  value = value + 1;
}
```

- Here is its use in a client:

```
Counter c;
...
c++;
```

*Note: postfix form use...*

# OVERLOADING THE POSTFIX INCREMENT OPERATOR

- This overloads the postfix form of **++** for a class **Counter**:

```
class Counter {
private:
  int value;
public:
  ...
  void operator++(int unused);
  ...
};
```

*This apparently tells C++ that we want the postfix form.*

- Here is its implementation:

```
void Counter::operator++(int unused) {
  value = value + 1;
}
```

- Here is its use in a client:

```
Counter c;
...
c++;
```

*Note: postfix form use… …sends the method an* **unused** *of 0.*

# RECALL: IN C++ ARGUMENTS ARE PASSED BY VALUE

- Consider these function definitions

```
void increment(int i) {
  i = i+1;
}
void swap(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

- They don't do much. The code below does this:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(count);
swap(a,b);
std::cout << count << " " << a << " " << b << "\n";
```

# RECALL: IN C++ ARGUMENTS ARE PASSED BY VALUE

- Consider these function definitions

```
void increment(int i) {
  i = i+1;
}
void swap(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

- They don't do much. The code below does this:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(count);
swap(a,b);
std::cout << count << " " << a << " " << b << "\n";
```

CONSOLE

```
1 10 17 42
2 10 17 42
```

# PASSING POINTERS

- If we use pointers instead

```
void increment(int* ip) {
  (*ip) = (*ip)+1;
}
void swap(int* xp, int* yp) {
  int tmp = (*xp);
  (*xp) = (*yp);
  (*yp) = tmp;
}
```

- ...then we achieve what we want:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(&count);
swap(&a,&b);
std::cout << count << " " << a << " " << b << "\n";
```

# PASSING POINTERS

- If we use pointers instead

```
void increment(int* ip) {
  (*ip) = (*ip)+1;
}
void swap(int* xp, int* yp) {
  int tmp = (*xp);
  (*xp) = (*yp);
  (*yp) = tmp;
}
```

- ...then we achieve what we want:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(&count);
swap(&a,&b);
std::cout << count << " " << a << " " << b << "\n";
```

CONSOLE

```
1 10 17 42
2 11 42 17
```

# PASSING POINTERS

- If we use pointers instead

```
void increment(int* ip) {
  (*ip) = (*ip)+1;
}
void swap(int* xp, int* yp) {
  int tmp = (*xp);
  (*xp) = (*yp);
  (*yp) = tmp;
}
```

*We pass pointers that refer to the storage of the variables.*

- ...then we achieve what we want:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(&count);
swap(&a,&b);
std::cout << count << " " << a << " " << b << "\n";
```

CONSOLE

```
1 10 17 42
2 11 42 17
```

# PASSING POINTERS

- If we use pointers instead

```
void increment(int* ip) {
  (*ip) = (*ip)+1;
}
void swap(int* xp, int* yp) {
  int tmp = (*xp);
  (*xp) = (*yp);
  (*yp) = tmp;
}
```

*This makes *ip, *xp, *yp "aliases" of count, a, b.*

- ...then we achieve what we want:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(&count);
swap(&a,&b);
std::cout << count << " " << a << " " << b << "\n";
```

*We pass pointers that refer to the storage of the variables.*

CONSOLE

```
1 10 17 42
2 11 42 17
```

# PASSING AND RETURNING STRUCTS

- When a structure is passed as an argument with a function call, each of its components is copied into the local storage of the callee.

```
struct point100d {
  double x1;
  double x2;
  ...
  double x100;
};

void print(point100d p) {
  std::cout << "(" << p.x1 << ",";
  std::cout << p.x2 << ",";
  ...
}
...
  point100d big_point = ...;
  print(big_point);
...
```

*Copies 100 doubles, 640 bytes.*

## PASSING AND RETURNING STRUCTS

- When a structure is passed as an argument with a function call, each of its components is copied into the local storage of the callee.

```
struct point100d {
  double x1;
  double x2;
  ...
  double x100;
};

void print(point100d* p) {
  std::cout << "(" << p->x1 << ",";
  std::cout << p->x2 << ",";
  ...
}
...
  point100d big_point = ...;
  print(&big_point);
...
```

*In C, people passed pointers to prevent all this copying... a pointer is only 8 bytes.*

~~*Copies 100 doubles, 640 bytes.*~~

# PASSING AND RETURNING STRUCTS

- Copying of components happens when a function returns a struct.

```
struct point100d {
  double x1;
  double x2;
  ...
  double x100;
};

point100d input(void) {
  point100d p;
  std::cin >> p.x1;
  std::cin >> p.x2;
  ...
  return p;
}
...
  point100d big_point = input();
...
```

*Copies 100 doubles, 640 bytes.*

# PASSING AND RETURNING STRUCTS

- Copying of components happens when a function returns a struct.

```
struct point100d {
  double x1;
  double x2;
  ...
  double x100;
};

point100d input(void) {
  point100d p;
  std::cin >> p.x1;
  std::cin >> p.x2;
  ...
  return p;
}
...
  point100d big_point = input();
...
```

*No easy way around this unless you heap allocate the struct's storage...*

*Copies 100 doubles, 640 bytes.*

# PASSING ~~AND RETURNING~~ STRUCTS

- Copying of components happens when a function returns a struct.

```
struct point100d {
  double x1;
  double x2;
  ...
  double x100;
};

void get(point100d *p) {
  std::cin >> p->x1;
  std::cin >> p->x2;
  ...
  std::cin >> p->x100;
}
...
  point100d big_point;
  get(&big_point);
...
```

*No easy way around this unless you heap allocate the struct's storage...*

*...or write the code differently.*

# PASSING "BY REFERENCE"

- C++ allows you to pass parameters ***by reference***.

```
void increment(int& i) {
  i = i+1;
}
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

*The use of & makes the parameters i, x, and y aliases of count, a, b.*

CONSOLE

```
10 17 42
11 42 17
```

- The client code looks none the wiser:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(count);
swap(a,b);
std::cout << count << " " << a << " " << b << "\n";
```

- But, under the covers, C++ does all the logistical work of passing pointers instead of copying values.

# PASSING STRUCTS "BY REFERENCE"

- We can do the same to avoid copying when we pass structs:

```
void print(point100d& p) {
  std::cout << "(" << p.x1 << ",";
  std::cout << p.x2 << ",";
  ...
  std::cout << p.x100 << ")" << std::endl;
}
```

- And we can modify structs' components this way, of course, too:

```
void get(point100d& p) {
  std::cin >> p.x1;
  std::cin >> p.x2;
  ...
  std::cin >> p.x100;
}
```

# PASSING OBJECTS BY REFERENCE

- We can do the same to avoid copying when we pass objects as parameters:

```
class Point100d {
  double x1;
  double x2;
  ...
  double x100;
  void operator+=(Point100d& that) {
    this->x1 += that.x1;
    this->x2 += that.x2;
    ...
    this->x100 += that.x100;
  }
};
```

# PASSING OBJECTS BY REFERENCE

- We can do the same to avoid copying when we pass objects as parameters:

```
class Point100d {
  double x1;
  double x2;
  ...
  double x100;
  void operator+=(Point100d& that) {
    this->x1 += that.x1;
    this->x2 += that.x2;
    ...
    this->x100 += that.x100;
  }
};
```

- But, this kind of reference passing *might be concerning* to the client.
- It *might not want the method to change* the contents of what it passes.

# CONST PARAMETERS

- The keyword const advertises and enforces this restriction:

```
class Point100d {
  double x1;
  double x2;
  ...
  double x100;
  void operator+=(const Point100d& that) {
    this->x1 += that.x1;
    this->x2 += that.x2;
    ...
    this->x100 += that.x100;
  }
};
```

- The `const` keyword indicates that the contents of `that` aren't modified.
- The compiler enforces this. Raises an error if the method's body violates it.

# CONST METHODS

- Consider the print method below:

```
class Point100d {
  double x1;
  double x2;
  ...
  double x100;
  void print(void) const {
    std::cout << "(" << this->x1 << ",";
    std::cout << this->x2 << ",";
    ...
    std::cout << this->x100 << ")";
  }
};
```

- The `const` keyword indicates that the contents of `this` aren't modified.
- The compiler enforces this, too, makes sure the method body behaves.

# EXAMPLE CLASS INTERFACES WITH CONST AND REFERENCE

```
class Rational {
private:
  int num;
  int den;

public:
  // constructors
  Rational(void);
  Rational(std::string s);
  Rational(int n);
  Rational(int n, int d);

  // methods
  Rational plus(const Rational& that) const;
  Rational times(const Rational& that) const;
  std::string to_string(void) const;
};

Rational operator+(const Rational& q1, const Rational& q2);
Rational operator*(const Rational& q1, const Rational& q2);
```

# EXAMPLE CLASS INTERFACES WITH CONST AND REFERENCE

```
class Stck {

private:
  int *elements;
  int num_elements;
  int capacity;

public:
  Stck(int capacity);
  bool is_empty() const;
  void push(int value);
  int pop();
  int top() const;
  std::string to_string() const;
  ~Stck();
  friend ostream& operator<<(ostream& os, const Stck& s);
  friend istream& operator<<(istream& is, Stck& s);
};
```

# HMMM...

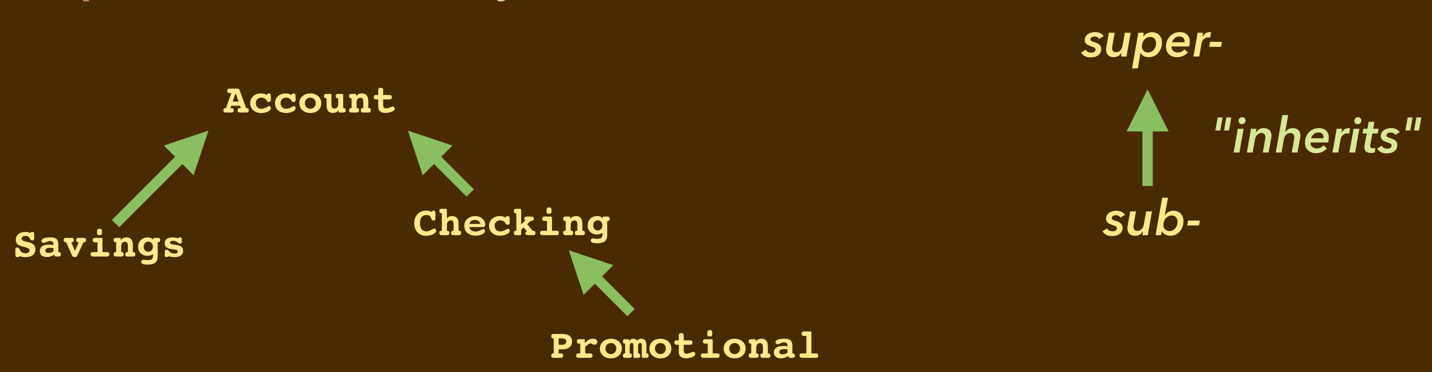
```
class Stck {

private:
  int *elements;
  int num_elements;
  int capacity;

public:
  Stck(int capacity);
  bool is_empty() const;
  void push(int value);
  int pop();
  int top() const;
  std::string to_string() const;
  ~Stck();
  friend ostream& operator<<(ostream& os, const Stck& s);
  friend istream& operator<<(istream& is, Stck& s);
};
```

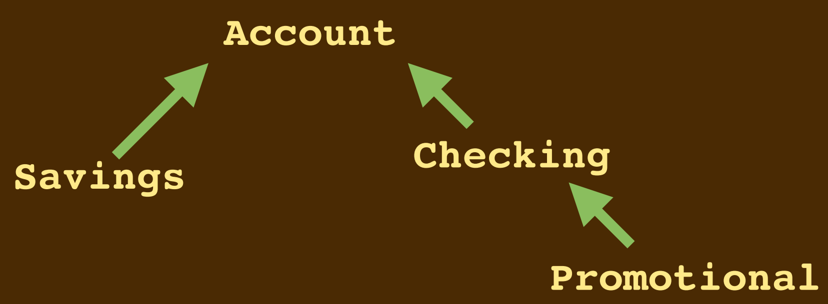
# INHERITANCE

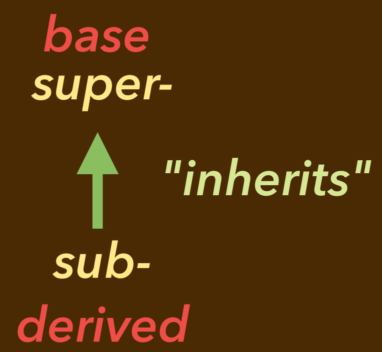
- **RECALL:** OO languages allow us to extend object classes:
  - ➡adding instance variables enhances what they can represent.
  - ➡adding methods enhances their behavior.
- The standard mechanism for this is ***subclassing***.
  - ➡ A subclass ***inherits*** the fields and behavior of its *superclass*.
  - ➡ The extensions make it more ***specialized***.
  - ➡ We can develop a *class hierarchy*.
- Example:



# INHERITANCE

- **RECALL:** OO languages allow us to extend object classes:
  - ➡adding instance variables enhances what they can represent.
  - ➡adding methods enhances their behavior.
- The standard mechanism for this is ***subclassing***.
  - ➡ A subclass ***inherits*** the fields and behavior of its *superclass*.
  - ➡ The extensions make it more ***specialized***.
  - ➡ We can develop a *class hierarchy*.
- Example:



# ACCOUNT CLASS

```
class Account {
private:
  static long gNextNumber; // used to generate account nos.
  // instance variables
  std::string name;        // description of the account
  long number;             // account no.
  double balance;          // money held
  double rate;             // monthly interest
public:
  ...
};
```

# ACCOUNT CLASS

```
class Account {
private:
  static long gNextNumber;
  // instance variables
  ...
public:
  Account(std::string name, double amount, double interest);
  // getters
  double getBalance() const;
  std::string getName() const;
  long getNumber() const;
  double getRate() const;
  // methods
  void deposit(double amount);    // add money
  void gainInterest();            // each month
  double withdraw(double amount); // remove money
};
```

# ACCOUNT CLASS IMPLEMENTATION (MISSING GETTERS)

```
Account::Account(std::string name, double amount, double
interest) : name {name},
            balance {amount},
            rate {interest},
            number {Account::gNextNumber++}
{ }

void Account::deposit(double amount) {
  balance += amount;
}
void Account::gainInterest() {
  deposit(rate * balance);
}
double Account::withdraw(double amount) {
  if (amount > balance) {
    amount = balance;
    balance = 0.0;
  } else {
    balance -= amount;
  }
  return amount;
}
```

# SUBCLASSES OF ACCOUNT

- Savings accounts accrue 2% interest. They charge a penalty for withdrawal.

```
class Savings : public Account { ... }
```

- Checking accounts accrue 1% interest, but only if balance is above $1000.

```
class Checking : public Account { ... }
```

- Promotional checking accounts accrue 0.7% interest, but give you $100 to open the account. You must stay above $100 to earn that interest.

```
class Promotional : public Checking { ... }
```

- The keyword **`public`** means that
  - all public members are accessible as public members in the derived class,
  - all protected members are accessible as public members in the derived class,
  - private members are only accessible if a friend.

# ACCOUNT CLASS, READIED FOR DERIVING

```
class Account {
private:
  static long gNextNumber;
protected:
  // instance variables
  std::string name;
  long number;
  double balance;
  double rate;
public:
  // methods
  ...
};
```

Not publicly accessible, but accessible to any derived class.

# ACCOUNT CLASS, READIED FOR DERIVING

```
class Account {
private:
  static long gNextNumber;
  // instance variables
  ...
public:
  // methods
  Account(std::string name, double amount, double interest);
  virtual double getBalance() const;
  virtual std::string getName() const;
  virtual long getNumber() const;
  virtual double getRate() const;
  virtual void deposit(double amount);
  virtual void gainInterest();
  virtual double withdraw(double amount);
};
```

Virtual keyword indicates that the code of overriding methods in subclass will get called.

# EXTENSIONS AND OVERRIDES

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

class Promotional : public Checking {
public:
  Promotional(std::string name, double amount);
};
```

# SAVINGS ACCOUNT

Savings accounts accrue 2% interest. They charge a penalty for withdrawal.

- We add a `penalty` instance variable.
- We *override* the `withdraw` method to charge that penalty.

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

Savings::Savings(std::string name, double amount) :
  Account {name,amount,0.02}, penalty {50.0}
{ }

double Savings::withdraw(double amount) {
  double howmuch = Account::withdraw(amount);
  Account::withdraw(penalty);
  return howmuch;
}
```

# SAVINGS ACCOUNT

Savings accounts accrue 2% interest. They charge a penalty for withdrawal.

- We add a `penalty` instance variable.
- We *override* the `withdraw` method to charge that penalty.

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

Savings::Savings(std::string name, double amount) :
  Account {name,amount,0.02}, penalty {50.0}
{ }

double Savings::withdraw(double amount) {
  double howmuch = Account::withdraw(amount);
  Account::withdraw(penalty);
  return howmuch;
}
```

# CHECKING ACCOUNT

Checking accounts accrue 1% interest, but only if balance is above $1000.

- We add a `level` instance variable.
- We *override* the `gainInterest` method to check that level.

```
class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

Checking::Checking(std::string name, double amount) :
  Account {name, amount, 0.01}, level {1000.0}
{ }

void Checking::gainInterest() {
  if (balance >= level) {
    Account::gainInterest();
  }
}
```

# CHECKING ACCOUNT

Checking accounts accrue 1% interest, but only if balance is above $1000.

- We add a **`level`** instance variable.
- We *override* the **`gainInterest`** method to check that level.

```
class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

Checking::Checking(std::string name, double amount) :
  Account {name, amount, 0.01}, level {1000.0}
{ }

void Checking::gainInterest() {
  if (balance >= level) {
    Account::gainInterest();
  }
}
```

# PROMOTIONAL (CHECKING) ACCOUNT

Promotional accrues less interest, has an opening gift, has lower threshold.

- They derive from **`Checking`**. No extensions or overrides.

```
class Promotional : public Checking {
public:
  Promotional(std::string name, double amount);
};


Promotional::Promotional(std::string name, double amount) :
  Checking {name, amount + 100.0}
{
  rate = 0.07;
  level = 100.0;
}
```

# NON-VIRTUAL METHODS: DISPATCH ACCORDING TO TYPE

- Consider these two class definitions

```
class A {
    ...
    void m(...); // NOTE: not virtual!!!
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *a = new B();
a->m(x);
```

- Since **m** is not marked virtual, the code for **A::m** runs instead.
  - This is sometimes called "*static dispatch*" of the "message" **m**.

# VIRTUAL METHODS: DISPATCH ACCORDING TO CONTENTS

- Consider these two class definitions

```
class A {
    ...
    virtual void m(...); // yes virtual
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *a = new B();
a->m(x);
```

- Since **`m`** is marked virtual, the code for **`B::m`** runs like we'd normally expect.
    - This is sometimes called "*dynamic dispatch*" of the "message" **`m`**.

# CONTINUING PLAN

## THIS WEEK

- (MAYBE) ANOTHER SUBCLASSING EXAMPLE
- TEMPLATES
- COPY CONSTRUCTOR AND COPY ASSIGNMENT
- MOVE CONSTRUCTOR AND MOVE ASSIGNMENT
  - R-VALUE REFERENCES

## NEXT WEEK

- SECOND MIDTERM ON CIRCUITS AND ASSEMBLY