MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

04 C++ Classes (continued)

Department of Computing and Software

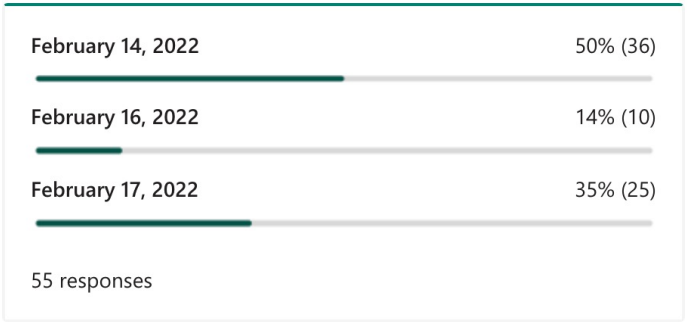
Instructor:

Omid Isfahanialamdari

January 19, 2022

Administration

- Results are suggesting February 14 as the date for the first midterm.
  - Please participate! I will wait until tonight (Wednesday).



- Tutorials will be held on Tuesday and Friday (meeting schedules are created)
  - Tutorial questions are uploaded
- TA Office hours will be held from this week (meeting schedules are created)
- Please ask questions in “Question and Answer” channel
  - We will answer as soon as possible
- Please read the textbook
  - Other online resources for C++

Recap (and more details)

Namespace

- A mechanism that allows a group of related names to be defined in one place
- Access an object **x** in namespace **group** using the notation:

  group::x

  - This is called **x**'s fully qualified name
  - Examples we have seen: **std::cout**
- **using** statement makes some or all of the names from the namespace accessible, without explicitly providing the specifier
  - Examples:
    - **using** std::string;
    - **using namespace** std;
- Motivation

Casting

- Casting is an operation that allows us to change the type of a variable.

```
int cat = 14;
double  dog = (double) cat; // traditional C-style cast
double  pig = double(cat);  // C++ functional cast
```

```
int i1 = 18;
int i2 = 16;
double  dv1 = i1 / i2;                  // dv1 = 1.0
double  dv2 = double(i1) / double(i2);  // dv2 = 1.125
double  dv3 = double( i1 / i2);         // dv3 = 1.0
```

Function Overloading

- **Overloading** means defining two or more functions or operators that have the same name, but whose behavior depends on the types of their actual arguments.

```
int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -123;
    cout << " = " << abs(-123) << endl;
    cout << "absolute value of " << -1.23;
    cout << " = " << abs(-1.23) << endl;
}
```

In C, you can't use the same name for multiple function definitions.

C++ allows multiple functions with the same name as long as argument types are different: the right function is determined at runtime based on argument types.

Function Overloading

- **Overloading** means defining two or more functions or operators that have the same name, but whose behavior depends on the types of their actual arguments.

```
int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -123;
    cout << " = " << abs(-123) << endl;
    cout << "absolute value of " << -1.23;
    cout << " = " << abs(-1.23) << endl;
}
```

In C, you can't use the same name for multiple function definitions.

C++ allows multiple functions with the same name as long as argument types are different: the right function is determined at runtime based on argument types.

Output:

absolute value of -123 = 123

Function Overloading

- **Overloading** means defining two or more functions or operators that have the same name, but whose behavior depends on the types of their actual arguments.

```
int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -123;
    cout << " = " << abs(-123) << endl;
    cout << "absolute value of " << -1.23;
    cout << " = " << abs(-1.23) << endl;
}
```

In C, you can't use the same name for multiple function definitions.

C++ allows multiple functions with the same name as long as argument types are different: the right function is determined at runtime based on argument types.

Output:

absolute value of -1.23 = 1.23

Operator Overloading

```
enum Day { MON, TUE, WED, THU, FRI, SAT, SUN};

Day& operator++(Day& d)  // prefix increment operator
{
    return d = (SUN == d) ? MON : Day(d+1);  // a ? b : c
}
```

```
void print(Day d){
  switch(d){
   case MON : cout << "mon\n"; break;
   case TUE : cout << "tue\n"; break;
   case WED : cout << "wed\n"; break;
   case THU : cout << "thu\n"; break;
   case FRI : cout << "fri\n"; break;
   case SAT : cout << "sat\n"; break;
   case SUN : cout << "sun\n"; break;
  }
}
```

```
int main( ) {
    Day d = TUE;
    cout << "current : " ;
    print(d);
    ++d;
    cout << "current : " ;
    print(d);

    return EXIT_SUCCESS;
}
```

Output:

current : tue
current : wed

Operator Overloading

```
enum Day { MON, TUE, WED, THU, FRI, SAT, SUN};

Day& operator++(Day& d)  // prefix increment operator
{
    return d = (SUN == d) ? MON : Day(d+1);  // a ? b : c
}
```

```
void print(Day d){
  switch(d){
   case MON : cout << "mon\n"; break;
   case TUE : cout << "tue\n"; break;
   case WED : cout << "wed\n"; break;
   case THU : cout << "thu\n"; break;
   case FRI : cout << "fri\n"; break;
   case SAT : cout << "sat\n"; break;
   case SUN : cout << "sun\n"; break;
  }
}
```

```
Day operator++(Day &d, int){  //postfix increment operator
    Day before = d;
    ++d;
    return before;
}
```

```
int main( ) {
    Day d = TUE;
    cout << "current : " ;
    print(d);
    ++d;  // operator++(d)
    cout << "current : " ;
    print(d);
    d++;  // operator++(d, 0)
    cout << "current : " ;
    print(d);
    //cout << d;
    cout << d++ << d ;
```

Output:

current : tue
current : wed
current : thu
I am in the overloaded operator: thu
I am in the overloaded operator: fri

Operator Overloading

```
enum Day { MON, TUE, WED, THU, FRI, SAT, SUN};

Day& operator++(Day& d)  // prefix increment operator
{
    return d = (SUN == d) ? MON : Day(d+1);  // a ? b : c
}
```

```
ostream& operator<<(ostream &out, const Day &d){
    out << "I am in the overloaded operator: " ;
    switch(d){
        case MON : out << "mon\n"; break;
        case TUE : out << "tue\n"; break;
        case WED : out << "wed\n"; break;
        case THU : out << "thu\n"; break;
        case FRI : out << "fri\n"; break;
        case SAT : out << "sat\n"; break;
        case SUN : out << "sun\n"; break;
    }
    return out;
}
```

```
Day operator++(Day &d, int){  //postfix increment operator
    Day before = d;
    ++d;
    return before;
}
```

```
int main( ) {
    Day d = TUE;
    cout << "current : " ;
    print(d);
    ++d;  // operator++(d)
    cout << "current : " ;
    print(d);
    d++;  // operator++(d, 0)
    cout << "current : " ;
    print(d);
    //cout << d;
    cout << d++ << d ;
```

Output:

current : tue
current : wed
current : thu
I am in the overloaded operator: thu
I am in the overloaded operator: fri

C++ Class

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

Access Specifiers

C++ Class

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

Access Specifiers

Member Variables

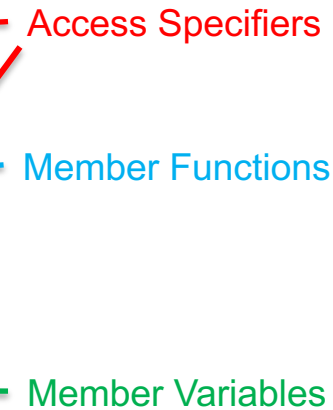
C++ Class

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```



C++ Class

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

Function Declarations

- Recall in-line definitions

Function Definitions

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

C++ Class

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
int main(){
    Passenger p1;
    p1.print();
    p1.name = "X";
    return EXIT_SUCCESS;
}
```

Instantiation

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

C++ Class

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
int main(){
    Passenger p1;
    p1.print();
    p1.name = "X";
    return EXIT_SUCCESS;
}
```

Accessing members

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

C++ Class

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
int main(){
    Passenger p1;
    p1.print();
    p1.name = "X";
    return EXIT_SUCCESS;
}
```

Error! Not allowed!

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

C++ Class

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
int main(){
    Passenger p1;
    p1.print();
    p1.name = "X";
    return EXIT_SUCCESS;
}
```

outputs :
--NO NAME--, NONE

Allowed

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

Class Constructors

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {          same name as class
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

- A user-defined member function
  - initializes member variables
- Obviously declared in “public”
- No return type
- Function overloading

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

Default Constructor

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

- Default constructor
  - called when no arguments supplied

Default Constructor

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
int main(){
    Passenger p1;                    //default constructor
    p1.print();
    Passenger* pp1 = new Passenger;  //default constructor
    pp1->print();
    Passenger pa[20];                //default constructor
    return EXIT_SUCCESS;
}
```

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

Default Constructor

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
int main(){
    Passenger p1;                    //default constructor
    p1.print();      outputs: --NO NAME--, NONE
    Passenger* pp1 = new Passenger;  //default constructor
    pp1->print();    outputs: --NO NAME--, NONE
    Passenger pa[20];                //default constructor
    return EXIT_SUCCESS;
}
```

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

Constructor with Arguments

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

- Note the default argument

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

Constructor with Arguments

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
int main(){
    Passenger p2("John Smith", VEGETARIAN, "293145"); // 2nd cons.
    p2.print();
    Passenger p3("Pocahontas", REGULAR);  // not a frequent flyer
    p3.print();
    Passenger* pp2 = new Passenger("JoeBlow", NO_PREF); // 2nd cons.
    pp2->print();
    return EXIT_SUCCESS;
}
```

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

Constructor with Arguments

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
int main(){
    Passenger p2("John Smith", VEGETARIAN, "293145"); // 2nd cons.
    p2.print();     outputs: John Smith, 293145
    Passenger p3("Pocahontas", REGULAR);  // not a frequent flyer
    p3.print();     outputs: Pocahontas, NONE
    Passenger* pp2 = new Passenger("JoeBlow", NO_PREF); // 2nd cons.
    pp2->print();   outputs: JoeBlow, NONE
    return EXIT_SUCCESS;
}
```

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

Quick Note: Initialization List

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

- Alternate method of initialization (casting)
- Between function signature and its body
- : followed by comma-separated list of:
  - memberName(initValue)

```
Passenger::Passenger(const string& nm, MealType mp, const string& ffn)
    : name(nm), mealPref(mp), isFreqFlyer(ffn != "NONE"), freqFlyerNo(ffn)
    { }
```

Copy Constructor

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

- Called when a reference to another object is given as argument from which to copy information

Copy Constructor

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
int main(){
    Passenger p2("John Smith", VEGETARIAN, "293145"); // 2nd cons.
    Passenger p3("Pocahontas", REGULAR);  // not a frequent flyer
    Passenger p4(p3);                              //copied from p3
    p4.print();
    Passenger p5 = p2;                             //copied from p2
    p5.print();
    return EXIT_SUCCESS;
}
```

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

Copy Constructor

- Passenger Class

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
int main(){
    Passenger p2("John Smith", VEGETARIAN, "293145"); // 2nd cons.
    Passenger p3("Pocahontas", REGULAR);  // not a frequent flyer
    Passenger p4(p3);                              //copied from p3
    p4.print();     outputs: Pocahontas, NONE
    Passenger p5 = p2;                             //copied from p2
    p5.print();     outputs: John Smith, 293145
    return EXIT_SUCCESS;
}
```

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

Quick Note: Organizing C++ Files

- Separating Class definition, implementation of member functions and tests

Passenger.h

```
#ifndef PASSENGER_H
#define PASSENGER_H

#include <iostream>
#include <string>
using namespace std;

enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);
        void print();

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};

#endif
```

TestPassenger.cpp

```
int main(){
    Passenger p2("John Smith", VEGETARIAN, "293145"); // 2nd cons.
    Passenger p3("Pocahontas", REGULAR);  // not a frequent flyer
    Passenger p4(p3);                             //copied from p3
    p4.print();
    Passenger p5 = p2;                            //copied from p2
    p5.print();
    return EXIT_SUCCESS;
}
```

Passenger.cpp

```
#include "Passenger.h"

using namespace std;

Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}

bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}

void Passenger::print() {
    cout << name << ", " << freqFlyerNo << endl;
}
```

Destructors

- Destructor:
  - needed when you allocate memory dynamically!
  - declared using ~classname
  - called when a class object **dies**
- Avoid shallow copy
  - implement your own copy constructor

```
class Vect {
    public:
        Vect(int n = 10);
        Vect(const Vect& a);
        Vect& operator=(const Vect& a);
        ~Vect();
    private:
        int* data;
        int size;
};
```

```
Vect::Vect(int n) {
    size = n;
    data = new int[n];
}

Vect::~Vect() {
    delete [] data;
}
```

```
Vect::Vect(const Vect& a) {           // copy constructor from a
    size = a.size;                    // copy sizes
    data = new int[size];             // allocate new array
    for (int i = 0; i < size; i++) {  // copy the vector contents
        data[i] = a.data[i];
    }
}

Vect& Vect::operator=(const Vect& a) {  // assignment operator from a
    if (this != &a) {                   // avoid self-assignment
        delete [] data;                 // delete old array
        size = a.size;                  // set new size
        data = new int[size];           // allocate new array
        for (int i=0; i < size; i++) {  // copy the vector contents
            data[i] = a.data[i];
        }
    }
    return *this;
}
```

```
int main() {
    Vect a(100);  // a is a vector of size 100
    Vect b(a);    // initialize b from a (DANGER!)
    Vect c;       // c is a vector (default size 10)
    c = a;        // assign a to c (DANGER!)

    return EXIT_SUCCESS;
}
```

Quick Note: Dynamic Memory Allocation

- allocate with **new**
- deallocate with **delete**
  - Otherwise, memory leak will happen!

```
int *p = new int;
// ...
delete p;

char* buffer = new char[500];
buffer[3] = 'a';
// ...
delete [] buffer;
```

Destructors

- Destructor:
  - needed when you allocate memory dynamically!
  - declared using ~classname
  - called when a class object **dies**
- Avoid shallow copy
  - implement your own copy constructor

```
class Vect {
    public:
        Vect(int n = 10);
        Vect(const Vect& a);
        Vect& operator=(const Vect& a);
        ~Vect();
    private:
        int* data;
        int size;
};
```

```
Vect::Vect(int n) {
    size = n;
    data = new int[n];
}

Vect::~Vect() {
    delete [] data;
}
```

```
Vect::Vect(const Vect& a) {           // copy constructor from a
    size = a.size;                    // copy sizes
    data = new int[size];             // allocate new array
    for (int i = 0; i < size; i++) {  // copy the vector contents
        data[i] = a.data[i];
    }
}

Vect& Vect::operator=(const Vect& a) {  // assignment operator from a
    if (this != &a) {                   // avoid self-assignment
        delete [] data;                 // delete old array
        size = a.size;                  // set new size
        data = new int[size];           // allocate new array
        for (int i=0; i < size; i++) {  // copy the vector contents
            data[i] = a.data[i];
        }
    }
    return *this;
}
```

```
int main() {
    Vect a(100);  // a is a vector of size 100
    Vect b(a);    // initialize b from a (DANGER!)
    Vect c;       // c is a vector (default size 10)
    c = a;        // assign a to c (DANGER!)

    return EXIT_SUCCESS;
}
```

Destructors

- Destructor:
  - needed when you allocate memory dynamically!
  - declared using ~classname
  - called when a class object **dies**
- Avoid shallow copy
  - implement your own copy constructor

```
class Vect {
    public:
        Vect(int n = 10);
        Vect(const Vect& a);
        Vect& operator=(const Vect& a);
        ~Vect();
    private:
        int* data;
        int size;
};
```

```
Vect::Vect(int n) {
    size = n;
    data = new int[n];
}

Vect::~Vect() {
    delete [] data;
}
```

```
int main() {
    Vect a(100);  // a is a vector of size 100
    Vect b(a);    // initialize b from a (DANGER!)
    Vect c;       // c is a vector (default size 10)
    c = a;        // assign a to c (DANGER!)

    return EXIT_SUCCESS;
}
```

```
Vect::Vect(const Vect& a) {           // copy constructor from a
    size = a.size;                    // copy sizes
    data = new int[size];             // allocate new array
    for (int i = 0; i < size; i++) {  // copy the vector contents
        data[i] = a.data[i];
    }
}

Vect& Vect::operator=(const Vect& a) {  // assignment operator from a
    if (this != &a) {                   // avoid self-assignment
        delete [] data;                 // delete old array
        size = a.size;                  // set new size
        data = new int[size];           // allocate new array
        for (int i=0; i < size; i++) {  // copy the vector contents
            data[i] = a.data[i];
        }
    }
    return *this;
}
```

Destructors

- Destructor:
  - needed when you allocate memory dynamically!
  - declared using ~classname
  - called when a class object **dies**
- **Avoid shallow copy**
  - implement your own copy constructor

```
class Vect {
    public:
        Vect(int n = 10);
        Vect(const Vect& a);
        Vect& operator=(const Vect& a);
        ~Vect();
    private:
        int* data;
        int size;
};
```

```
Vect::Vect(int n) {
    size = n;
    data = new int[n];
}

Vect::~Vect() {
    delete [] data;
}
```

```
Vect::Vect(const Vect& a) {           // copy constructor from a
    size = a.size;                    // copy sizes
    data = new int[size];             // allocate new array
    for (int i = 0; i < size; i++) {  // copy the vector contents
        data[i] = a.data[i];
    }
}

Vect& Vect::operator=(const Vect& a) {  // assignment operator from a
    if (this != &a) {                   // avoid self-assignment
        delete [] data;                 // delete old array
        size = a.size;                  // set new size
        data = new int[size];           // allocate new array
        for (int i=0; i < size; i++) {  // copy the vector contents
            data[i] = a.data[i];
        }
    }
    return *this;
}
```

```
int main() {
    Vect a(100);  // a is a vector of size 100
    Vect b(a);    // initialize b from a (DANGER!)
    Vect c;       // c is a vector (default size 10)
    c = a;        // assign a to c (DANGER!)

    return EXIT_SUCCESS;
}
```

Destructors

- Destructor:
  - needed when you allocate memory dynamically!
  - declared using ~classname
  - called when a class object **dies**
- **Avoid shallow copy**
  - implement your own copy constructor

```
class Vect {
    public:
        Vect(int n = 10);
        Vect(const Vect& a);
        Vect& operator=(const Vect& a);
        ~Vect();
    private:
        int* data;
        int size;
};
```

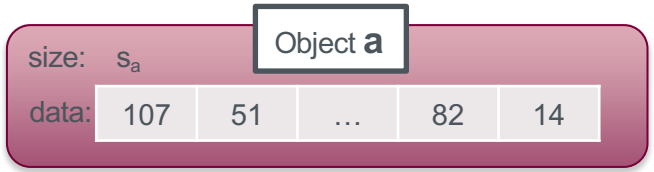
```
Vect::Vect(int n) {
    size = n;
    data = new int[n];
}

Vect::~Vect() {
    delete [] data;
}
```

```
Vect::Vect(const Vect& a) {           // copy constructor from a
    size = a.size;                    // copy sizes
    data = new int[size];             // allocate new array
    for (int i = 0; i < size; i++) {  // copy the vector contents
        data[i] = a.data[i];
    }
}

Vect& Vect::operator=(const Vect& a) {  // assignment operator from a
    if (this != &a) {                   // avoid self-assignment
        delete [] data;                 // delete old array
        size = a.size;                  // set new size
        data = new int[size];           // allocate new array
        for (int i=0; i < size; i++) {  // copy the vector contents
            data[i] = a.data[i];
        }
    }
    return *this;
}
```

```
Vect b(a);    // initialize b from a (DANGER!)
```



Destructors

- Destructor:
  - needed when you allocate memory dynamically!
  - declared using ~classname
  - called when a class object **dies**
- **Avoid shallow copy**
  - implement your own copy constructor

```
class Vect {
    public:
        Vect(int n = 10);
        Vect(const Vect& a);
        Vect& operator=(const Vect& a);
        ~Vect();
    private:
        int* data;
        int size;
};
```

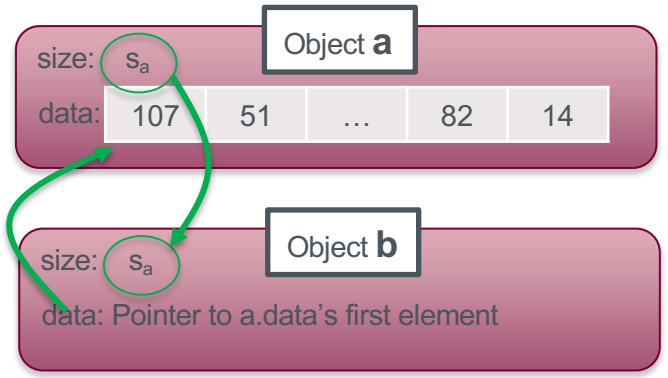
```
Vect::Vect(int n) {
    size = n;
    data = new int[n];
}

Vect::~Vect() {
    delete [] data;
}
```

```
Vect::Vect(const Vect& a) {           // copy constructor from a
    size = a.size;                    // copy sizes
    data = new int[size];             // allocate new array
    for (int i = 0; i < size; i++) {  // copy the vector contents
        data[i] = a.data[i];
    }
}

Vect& Vect::operator=(const Vect& a) {  // assignment operator from a
    if (this != &a) {                   // avoid self-assignment
        delete [] data;                 // delete old array
        size = a.size;                  // set new size
        data = new int[size];           // allocate new array
        for (int i=0; i < size; i++) {  // copy the vector contents
            data[i] = a.data[i];
        }
    }
    return *this;
}
```

```
Vect b(a);    // initialize b from a (DANGER!)
```



Classes with Dynamically-Allocated Members

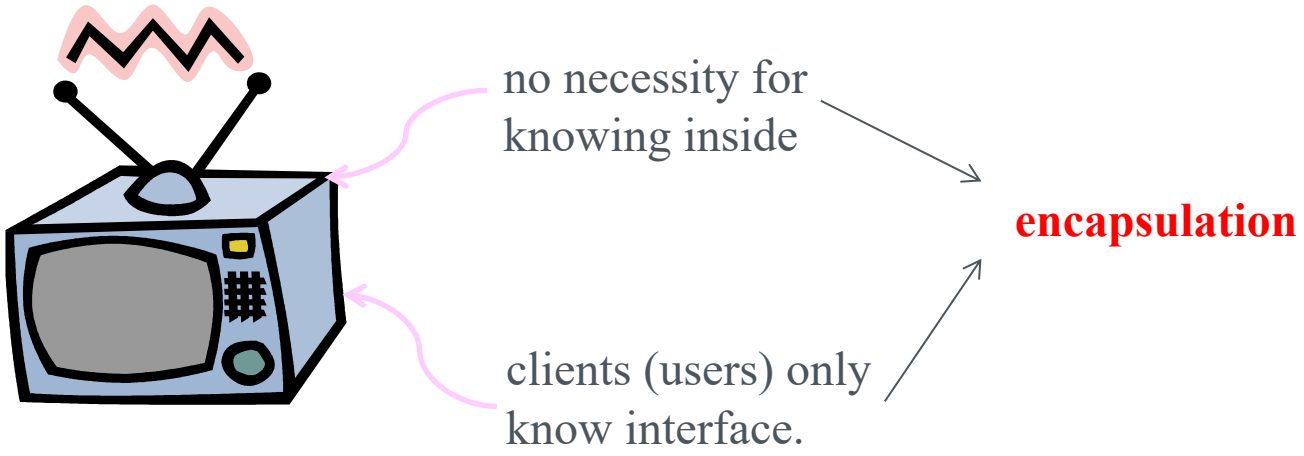
- Every class that allocates its own objects using **new** should:
  - Define a **destructor** to free any allocated objects.
  - Define a **copy constructor**, which allocates its own new member storage and copies the contents of member variables.
  - Define an **assignment operator**, which deallocates old storage, allocates new storage, and copies all member variables.

```
int main() {
    Vect a(100);  // a is a vector of size 100
    Vect b(a);    // initialize b from a (DANGER!)
    Vect c;       // c is a vector (default size 10)
    c = a;        // assign a to c (DANGER!)

    return EXIT_SUCCESS;
}
```

Encapsulation

- **Encapsulation** conceals the functional details defined in a class from external world (clients).
  - Information hiding
    - By limiting access to member variables/functions from outside
  - Operation through **interface**
    - Allows access to member variables through interface
  - Separation of **interface from implementation**
    - declaration vs definition



Friends of a Class

- In some cases, information-hiding is too prohibitive.
  - Only public members of a class are accessible by non-members of the class
- "friend" keyword
  - To give nonmembers of a class access to the nonpublic members of the class
- Friend
  - Functions
  - Classes – we skip this!
    - Poor class structure design

```
class Point{
    public:
        Point(double x1, double y1);
        friend ostream& operator<<(ostream &out, const Point &p1);
    private:
        double x, y;
};

Point::Point(double x1, double y1){
    x = x1;
    y = y1;
}
```

```
ostream& operator<<(ostream &out, const Point &p1){
    out << "P(" << p1.x << ", " << p1.y << ")";
    return out;
}
```

```
Point p(2.0, 4.0);
cout << p << endl;
```

Output:
P(2, 4)

Friends of a Class

- In some cases, information-hiding is too prohibitive.
  - Only public members of a class are accessible by non-members of the class
- “friend” keyword
  - To give nonmembers of a class access to the nonpublic members of the class
- Friend
  - Functions
  - Classes – we skip this!
    - Poor class structure design

```
void rectangle::setLT(point pt) {
  leftTop.set(pt.x, pt.y);
}
```

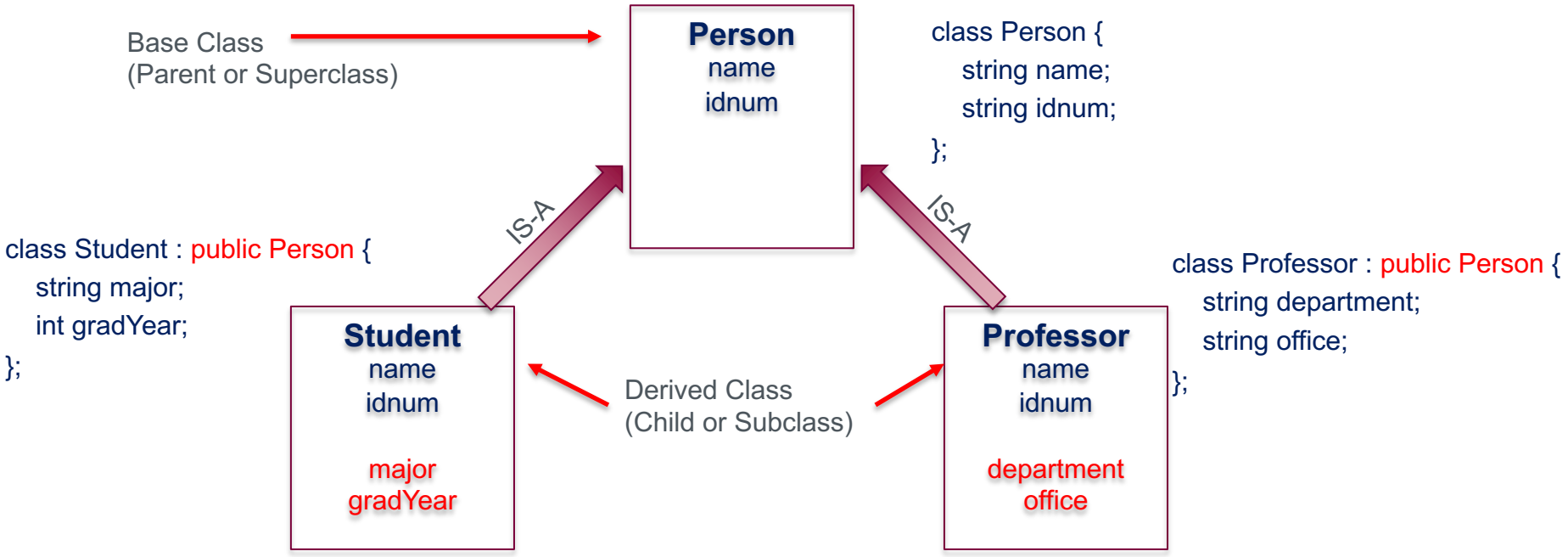
```
class Point{
    public:
        Point(double x1, double y1);
        void set(int a, int b);
    private:
        friend class rectangle;
        double x, y;
};

class rectangle {

    public:
        void setLT(point pt);
    private:
        Point leftTop, rightBottom;
};
```

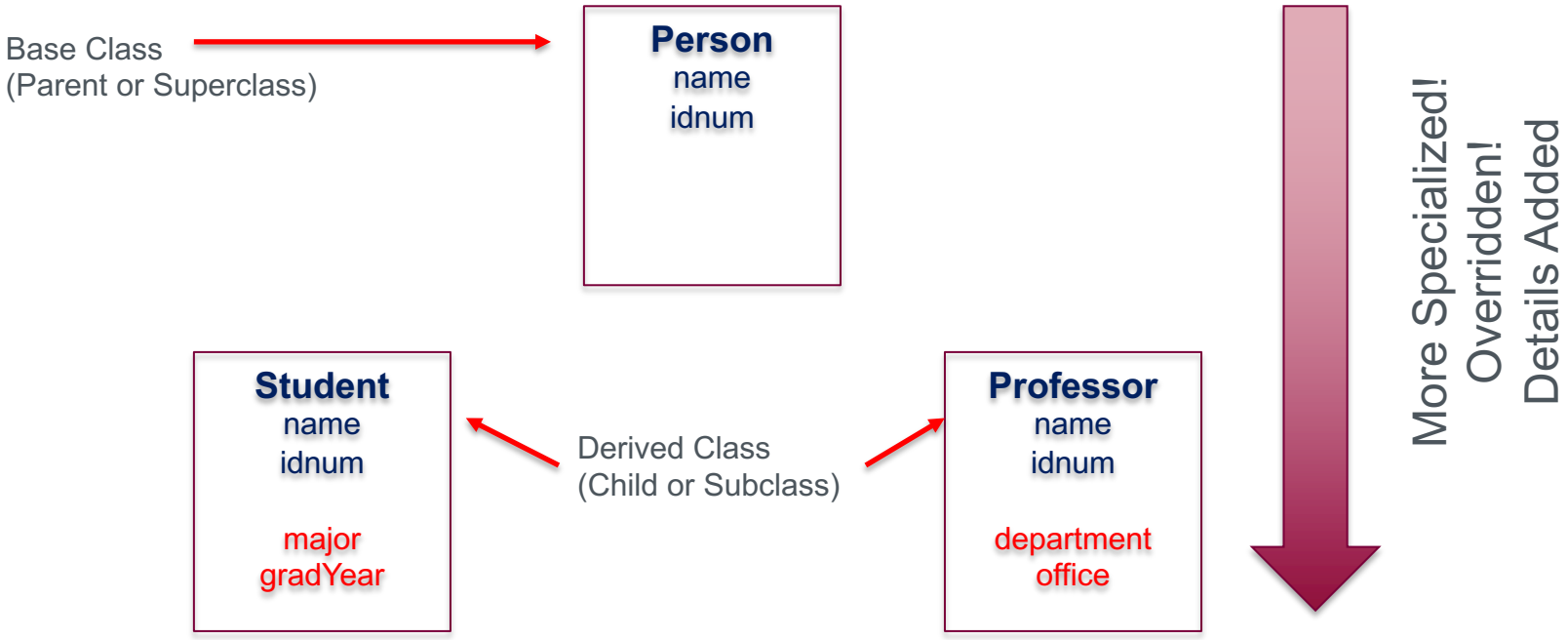
Inheritance

- Subclassing: define a class based on another class
  - Another class is parent class (or superclass)
  - New class is child class (subclass)
  - Hierarchical classification in a tree form
  - A way of "polymorphism" – we will discuss later!



Inheritance

- Subclassing: define a class based on another class
  - Another class is parent class (or superclass)
  - New class is child class (subclass)
  - Hierarchical classification in a tree form
  - A way of "polymorphism" – we will discuss later!



Inheritance

```
class Person {  // Person (base class)
    private:
        string name;   // name
        string idNum;  // university ID number

    public:
        Person(const string& nm, const string& id);
        void print();      // print information
        string getName();  // retrieve name
};
```

```
class Student : public Person {  // Student (derived from Person)
    private:
        string major;  // major subject
        int gradYear;  // graduation year

    public:
        Student(const string& nm, const string& id, const string& maj, int year);
        void print();                              // print information
        void changeMajor(const string& newMajor);  // change major

};
```

Inheritance

```
class Person {  // Person (base class)
    private:
        string name;   // name
        string idNum;  // university ID number

    public:
        Person(const string& nm, const string& id);
        void print();      // print information
        string getName();  // retrieve name
};
```

```
Person::Person(const string& nm, const string& id)
    : name(nm), idNum(id)  // initialize name and id
    { }

void Person::print() { // definition of Person print
    cout << "Name " << name << ", " << "IDnum " << idNum << endl;
}

string Person::getName() { // definition of Person getName
    return name;
}
```

Inheritance

```
class Person {  // Person (base class)
    private:
        string name;   // name
        string idNum;  // university ID number

    public:
        Person(const string& nm, const string& id);
        void print();      // print information
        string getName();  // retrieve name
};
```

```
class Student : public Person {  // Student (derived from Person)
    private:
        string major;  // major subject
        int gradYear;  // graduation year

    public:
        Student(const string& nm, const string& id, const string& maj, int year);
        void print();                              // print information
        void changeMajor(const string& newMajor);  // change major

};
```

Inheritance

```
class Student : public Person {  // Student (derived from Person)
    private:
        string major;  // major subject
        int gradYear;  // graduation year

    public:
        Student(const string& nm, const string& id, const string& maj, int year);
        void print();                                // print information
        void changeMajor(const string& newMajor);  // change major

};
```

```
Student::Student(const string& nm, const string& id, const string& maj, int year)
    : Person(nm, id), major(maj), gradYear(year)
    { }

void Student::print() {   // definition of Student print
    Person::print();      // first print Person information
    cout << "Major " << major << ", Year " << gradYear << endl;  // then student-specific info
}

void Student::changeMajor(const string& newMajor) {  // definition of Student print
    major = newMajor;
}
```

Inheritance

```
int main() {
    Person person("Mary", "12-345");  // declare a Person
    Student student("Bob", "98-764", "Math", 2012);  // declare a Student
    cout << student.getName() << endl;  // invokes Person::getName()
    person.print();   // invokes Person::print()
    student.print();  // invokes Student::print()
    //person.changeMajor("Physics");   // ERROR!
    student.changeMajor("English");  // Okay

    return EXIT_SUCCESS;
}
```

Bob
Name Mary, IDnum 12-345
Name Bob, IDnum 98-764
Major Math, Year 2012

Questions?