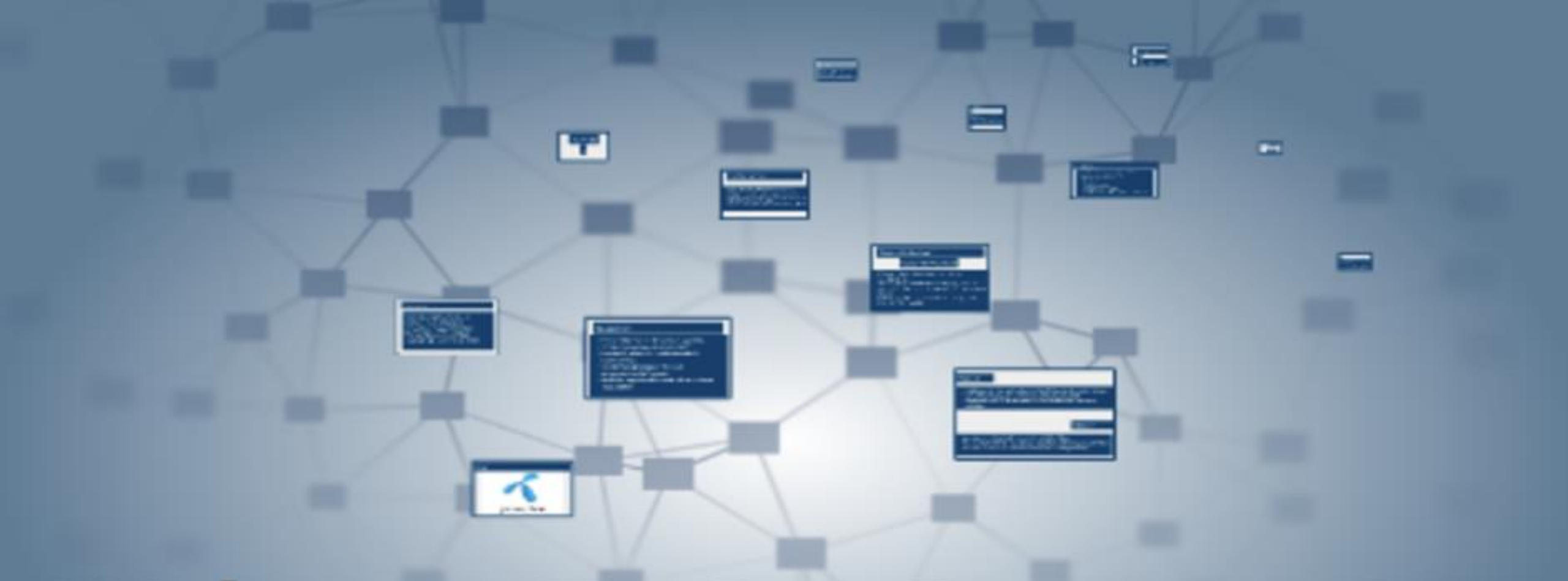


Welcome To Our Presentation

Lösningen ska innehålla

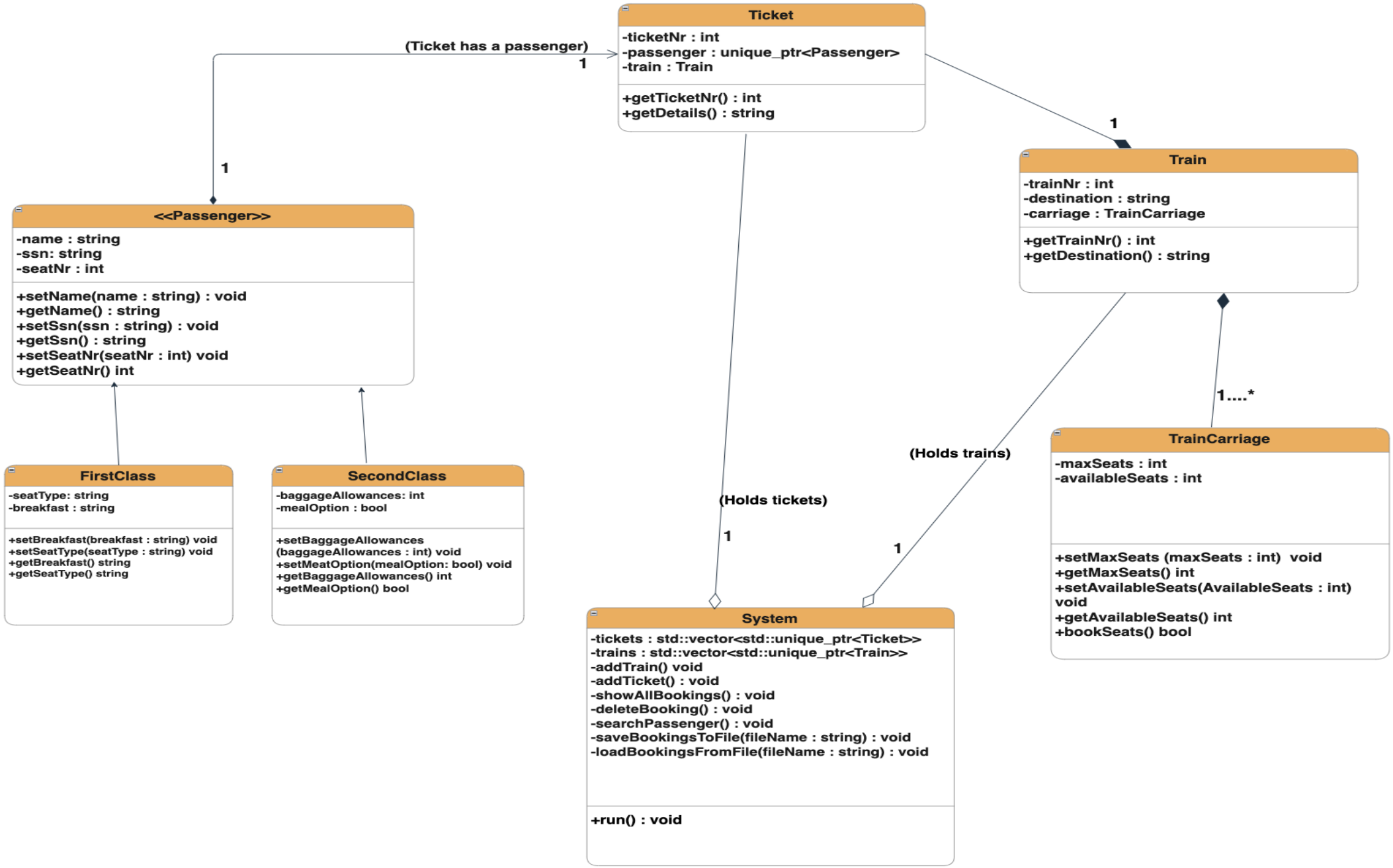
- Ett visst antal, minst 5, egendefinierade **relevanta** klasser, vilka på förhand ska vara **godkända av lärare**
- **Flera objekt** av någon av de egendefinierade klasserna ska finnas. Dessa ska hanteras i en behållare (array, vector, ...).
- Egendefinierat **logiskt och rimligt arv** med **abstrakt klass/er**
- **Överskuggning och dynamisk bindning** på ett lämpligt och relevant sätt
- Rimlig användning **av dynamisk minneshantering** (genom användande av new och delete)
- Någon **datastruktur från standardbiblioteket** (vector, stack, kö, ...)
- Läsning från och/eller skrivning till **textfiler**

Train-Booking-System

- Tågbokningssystem
- Möjlighet till:
- Boka ett tåg och se biljett informationen
- Visa alla bokningar
- Söka upp passagerare
- Ta bort bokningar

- Demokörning

Train booking system



Relationer mellan klasser

- **Composition** (mellan `Train` och `TrainCarriage`).

Exempel**:** Ett tåg är beroende av sina vagnar, så vagnarna är en del av tåget.

- **Aggregation** (mellan `System` och `Train`).

Exempel**:** Systemet hanterar biljetter men "äger" dem inte, och tåg kan potentiellt existera oberoende.

- **Association** (mellan `Ticket` och `Passenger`).

Exempel**:** En biljett är kopplad till en passagerare, men de är separata enheter.

Objekt

- **Hantering av objekt:** Klasser som Ticket och Train hanteras i vektorer. Det gör att vi dynamiskt kan lägga till eller ta bort objekt vid behov.
- Eftersom antalet objekt kan förändras under programmets gång.

Kod exempel: system.cpp

```
void System::addTrain() {
    int trainNr;
    std::string destination;

    std::cout << "Enter train number: ";
    std::cin >> trainNr;
    std::cin.ignore();
    std::cout << "Enter destination: ";
    std::getline(std::cin, destination);

    auto train = std::make_unique<Train>(trainNr, destination);
    trains.push_back(std::move(train));

    std::cout << "Train added successfully!\n";
}
```

- make_uniqie<Train>(trainNr, destination);
- Skapar nytt train objekt på heapen och retunera en unique_ptr, (undviker minnesläckor)
- Push_back: flyttar objektet i trains container som lagrar tågen
- Move: flyttar pekare till containern(undviker kopiering)

Egendefinierat logiskt och rimligt arv med abstrakt klass

Subbklasser "FirstClass" och "SecondClass" ärver den abstrakta basklassens egenskaper och metoder.

Varför används det:

-Kod som delas mellan flera klasser kan placeras i basklassen, då undviker vi duplicering.

-Gör det möjligt att hantera objekt av olika typer (`FirstClass` och `SecondClass`) genom en gemensam bas (`Passenger`), vilket förenklar kod som använder dessa objekt.

```
class FirstClass : public Passenger { /* ... */ };
class SecondClass : public Passenger { /* ... */ };
```

- `Basklassen Passenger` är abstrakt och innehåller gemensamma attribut och metoder som `name`, `ssn`, och `seatNr`, samt den rena virtuella metoden `description() = 0`.

```
virtual string description() const = 0;
```

Överskuggning och dynamisk bindning

- First Class o Second Class: olika version av description()
- Möjliggör polymorfism: Passenger* kan peka på både och

Kod exempel: SecondClass.cpp o FirstClass.cpp

```
string SecondClass::description() const {
    return "Type: Second Class, Name: " + getName() + ", SSN: " + getSsn() +
        ", Seat Number: " + to_string(getSeatNr()) +
        ", Baggage Allowances: " + to_string(baggageAllowances) +
        ", Meal Option: " + (mealOption ? "Yes" : "No");
}
```

```
string FirstClass::description() const {
    return "Type: First Class, Name: " + getName() + ", SSN: " + getSsn() +
        ", Seat Number: " + to_string(getSeatNr()) + ", Seat Type: " + seatType +
        ", Breakfast: " + breakfast;
}
```

```
Passenger* getPassenger() const;
```

Rimlig användning av dynamisk minneshantering (genom smarta pekare)

Vi har smarta pekare i vårt system som förenklar det att allokera minnet genom att automatiskt frigöra minnet när det inte längre behövs.

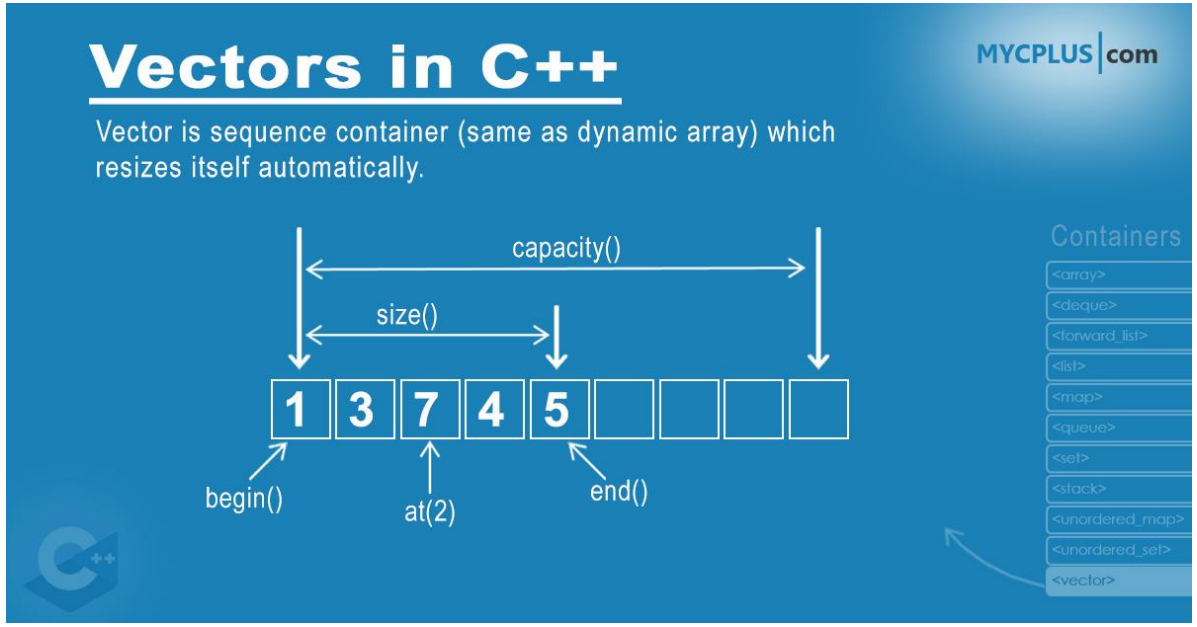
- Smarta pekare säkerställer att allokerat minne frigörs korrekt.
- **Säkerhet:** De förhindrar att minnet används efter att det har frigjorts.
- **Enklare hantering:** Man slipper manuellt kalla på `delete`.

`std::unique_ptr` används i klassen "Ticket" och "System" för att hantera objekt som `Passenger`, `Ticket`, och `Train`. Detta säkerställer att minneshantering sköts automatiskt, vilket förhindrar läckor.

```
std::unique_ptr<Passenger> passenger;
```

```
std::vector<std::unique_ptr<Ticket>> tickets;
std::vector<std::unique_ptr<Train>> trains;
```

Datastruktur från standard bibliotek



Exempel på datastrukturer från standardbiblioteket som används:

- **std::ifstream, std::ofstream**: För filhantering
- **std::numeric_limits**: hantera ogiltig användarinmatning.
- **std::remove_if**: filtrering och borttagning av biljetter.

Läsning från och/eller skrivning till textfiler

Läsning och skrivning används i vårt projekt för att spara och ladda bokningar.

Varför används det:
- Gör att programmet kan återuppta sin tidigare status genom att läsa inlagrade data.
- Användarna slipper lägga in information manuellt vid varje start.
- Data kan sparas för framtida referens eller felsökning.

`saveBookingsToFile` skriver alla biljetter och tåg till `bookings.txt`.
`loadBookingsFromFile` läser in data från samma fil `bookings.txt`.

```
void System::saveBookingsToFile(const std::string& fileName) const {
    std::ofstream outFile(fileName);
    if (!outFile) {
        std::cerr << "Error: Could not open file " << fileName << " for writing.\n";
        return;
    }
```

```
void System::loadBookingsFromFile(const std::string& fileName) {
    std::ifstream inFile(fileName);
    if (!inFile.is_open()) {
        std::cerr << "Error: Could not open file " << fileName << " for reading.\n";
        return;
    }
```

Thank You For Listening To Our Presentation