C++ Programming

Instructor: Rita Kuo
Office: CS 520E
Phone: Ext. 4405
E-mail: rita.kuo@uvu.edu

Mapping zyBooks Chapters

Topic	zyBooks Chapter
Class and Object	8.5
Class in C++	8.6, 8.7, 8.8, 8.10, 8.15
Initialization and Constructors	8.11, 8.12
Special Member Functions	
Static Data Members and Functions	8.13
Inline Functions	8.9
Overloading	8.18, 8.19, 8.20
Namespaces	8.22
Unit Testing (Class)	8.17

Self-study Chapters: 8.14, 8.16, 8.21, 8.23

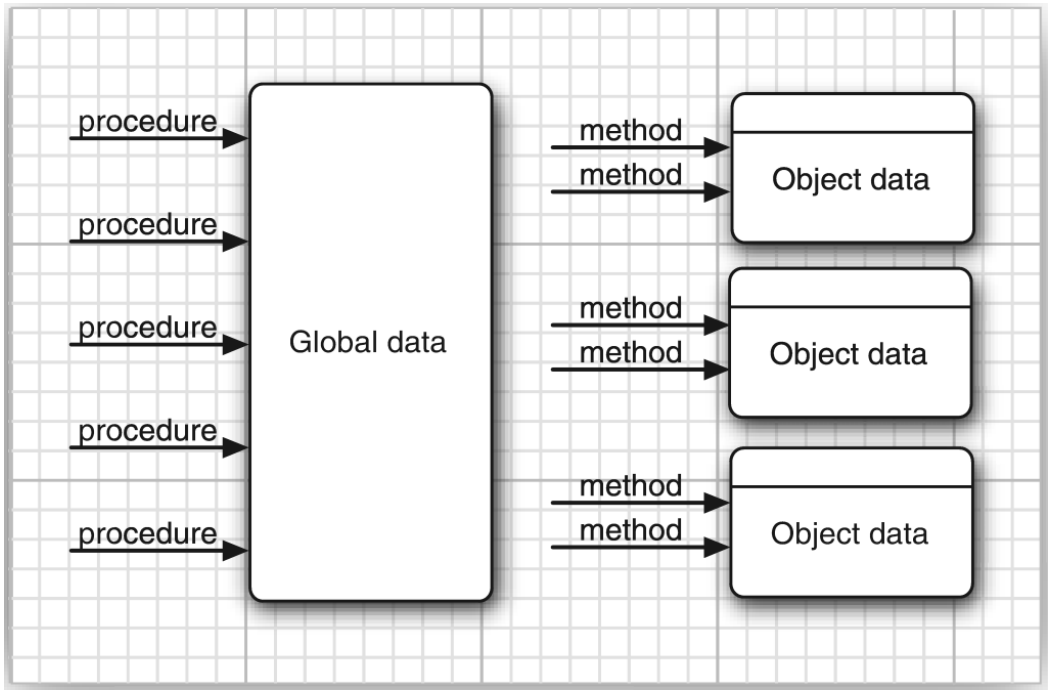
Class and Object

Object-Oriented Programming

- Traditional Structured Programming
  - Design a set of procedures to solve a problem → Algorithm
  - Find appropriate ways to store the data → Data Structure
- Object-Oriented Programming (OOP)
  - Decide the data → Object
  - Look at the algorithm to operate on the data → Algorithm

Object-Oriented Programming

- OOP is more appropriate for solving large problems
  - Example in web browser implementation:
    - Structure Programming: 2000+ procedure
    - Object-Oriented Programming: 100+ classes with average of 20 methods per class
  - Easier to handle the problem and find bugs



Review - Structure

- Define a structure
  - `struct` *structure_name* {
    member_type1 member_name1;
    member_type2 member_name2;
    ...
    };
  - Example

```
struct point_t {
  double x;
  double y;
};
```

Review - Structure

- Declaring a structure variable
  - `struct` *structure_name variable_name*`;`
  - Example
    ```
    struct point_t pt1;
    struct point_t pt2;
    ```
- Initializing Structure Variables
  - Prepare a list of values to be stored in the structure and enclose it in braces
  - Example:
    ```
    struct point_t pt1 = {200.0, 200.0};
    ```
  - The values in the initializer must appear in the same order as the members of the structure.

Review - Structure

- Accessing Structure Members
  - Structure member operator: ., also called the dot operator
  - Example:
    ```
    cout << "(" << pt1.x << ", " << pt2.x << ")";
    ```
  - Designated Initializers
    ```
    struct point_t pt1 = {.y = 200.0, .x = 100.0};
    ```
- Where to define structures?
  - Generally defined in a header file along with function prototype
  - Can defined them at the top of `.c` file

https://stackoverflow.com/questions/228684/

Object-Oriented Programming

- Anything in real life can be considered as an object
  - Object: Car
  - Attributes: speed, location
  - Behavior: speed up, slow down
- Objects in Programming Language
  - Attributes → Instance Variables
  - Behavior → Methods
- Object “Car” in Programming Language
  - Method: Speed Up
    → Increase “speed” variable, change “location” variable
  - Method: Slow Down
    → Decrease “speed” variable, change “location” variable

Object-Oriented Programming

- Class vs. Instance
  - Class: The blueprint of the car
  - Instance: A completed car built based on the blueprint
- Key Characteristics of Objects
  - Behavior: what you can do with this object, or what methods you can apply to it.
  - State: how the object react when you invoke those methods.
  - Identity: How the object is distinguished from other that may have the same behavior and state.

Object-Oriented Programming Major Concepts

- Encapsulation
  - Restrict access to methods and attributes in a class.
  - Hide the complex details from the users, and prevent data being modified by accident
- Inheritance
  - Define a class that inherits all the methods and attributes from another class
  - Makes the OOP code more modular, easier to reuse and build a relationship between classes
- Polymorphism
  - Use a single interface with different underlying forms such as data types or classes

https://pythonnumericalmethods.berkeley.edu/notebooks/chapter07.03-Inheritance-Encapsulation-and-Polymorphism.html

Class in C++

Define a Class

- Using the `class` keyword
- Example

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

class Restaurant {
  public:
    void Print()
    {
      cout << "Restaurant and Rating" << endl;
    }
};

int main()
{
  Restaurant r;
  r.Print();
  return 0;
}
```

Define a Class

- Using the `class` keyword
- Example

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

class Restaurant {
  public:
    void Print()
    {
      cout << "Restaurant and Rating" << endl;
    }
};

int main()
{
  Restaurant r;
  r.Print();
  return 0;
}
```

class name: Usually use Pascal case – the first letter of each compound word in a variable is capitalized (e.g., GradeBook)

class definition

Every class's body is enclosed in a pair of left and right braces ({ and }) and terminates with a semicolon

Define a Class

- Using the `class` keyword
- Example

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

class Restaurant {
  public:
    void Print()
    {
      cout << "Restaurant and Rating" << endl;
    }
};

int main()
{
  Restaurant r;
  r.Print();
  return 0;
}
```

Access specifier: always followed by a colon (:)

A member function in the Restaurant class: It is available to the public due to it is appears after the public access specifier

Define a Class

- Using the `class` keyword
- Example

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

class Restaurant {
  public:
    void Print()
    {
      cout << "Restaurant and Rating" << endl;
    }
};

int main()
{
  Restaurant r;
  r.Print();
  return 0;
}
```

r: a variable which is an instance of the Restaurant class

Member access operator (dot operator, .): call the member function – Print – using variable r

Define the Member Function Outside the Class

- Revise the program as follow:

```
class Restaurant {
  public:
    void Print();
};

void Restaurant::Print()
{
  cout << "Restaurant and Rating" << endl;
}

int main()
{
  Restaurant r;
  r.Print();
  return 0;
}
```

Function prototype

Binary scope resolution operator: ties each member function to the class – `Restaurant` in this example

Separate Files

- Create the files for the class `Restaurant`

Restaurant.h

```
#ifndef RESTAURANT_H
#define RESTAURANT_H

#include <iostream>
using namespace std;

class Restaurant {
  public:
    void Print();
};

#endif
```

Restaurant.cpp

```
#include "Restaurant.h"

void Restaurant::Print()
{
  cout << "Restaurant and Rating" << endl;
}
```

Separate Files

- Create a file to use the `Restaurant` class

Favoriate.cpp

```
#include <iostream>
#include "Restaurant.h"
using namespace std;

int main()
{
  Restaurant r;
  r.Print();
  return 0;
}
```

Review - Object-Oriented Programming Major Concepts

- Encapsulation
  - Restrict access to methods and attributes in a class.
  - Hide the complex details from the users, and prevent data being modified by accident
- Inheritance
  - Define a class that inherits all the methods and attributes from another class
  - Makes the OOP code more modular, easier to reuse and build a relationship between classes
- Polymorphism
  - Use a single interface with different underlying forms such as data types or classes

https://pythonnumericalmethods.berkeley.edu/notebooks/chapter07.03-Inheritance-Encapsulation-and-Polymorphism.html

Data Members

- Update the `Restaurant` class as follow:

Restaurant.h

```
class Restaurant {
  public:
    void Print();

  private:
    string name = "NoName";
    int rating = -1;
};
```

Data member initialization

Restaurant.cpp

```
void Restaurant::Print()
{
  cout << name << "--" << rating << endl;
}
```

Favoriate.cpp

```
int main()
{
  Restaurant r;
  r.Print();
  return 0;
}
```

```
~/CS-2370-Exercise$ ./main
NoName---1
~/CS-2370-Exercise$
```

Data Members

- Update the `main` function as follow:

```
int main()
{
  Restaurant r;
  cout << r.name << "--" << r.rating << endl;
  return 0;
}
```

Access Specifiers

- `public`
  - Members are accessible from outside the class
- `private`
  - Members cannot be accessed (or viewed) from outside the class
  - Declaring data member with access specifier `private` is known as **data hiding** → name and rating are **encapsulated** (hidden) in the object
  - Have to implement member functions to access the private member data
- `protected`
  - Members cannot be accessed from outside the class, however, they can be accessed in inherited classes. (will be discussed later)

https://www.w3schools.com/cpp/cpp_access_specifiers.asp

Accessors and Mutators

- A class's `private` data members can be manipulated only by member functions of that class
- Any statement that call the object's member function from outside the object
  → Call the class's `public` member functions to request the class's services for particular objects of the class
- Accessors (or getters)
  - Public member functions to allow clients of the class to get (i.e., obtain the value of) private data members
  - The member function names usually begin with `Get`. E.g., `GetName`
- Mutators (or setters)
  - Public member functions to allow clients of the class to set (i.e., assign values to) private data members
  - The member function names usually begin with `Set`. E.g., `SetRating`

Accessors and Mutators

- Add setters of the `Restaurant` class

Restaurant.h

```
class Restaurant {
  public:
    void SetName(string restaurantName);
    void SetRating(int userRating);
    void Print();

  private:
    string name;
    int rating;
};
```

Accessors and Mutators

- Add setters of the `Restaurant` class

Restaurant.cpp

```
void Restaurant::SetName(string restaurantName) {
  name = restaurantName;
}

void Restaurant::SetRating(int userRating) {
  rating = userRating;
}

void Restaurant::Print()
{
  cout << name << "--" << rating << endl;
}
```

Accessors and Mutators

- Add setters of the `Restaurant` class

Favoriate.cpp

```
int main()
{
  Restaurant favLunchPlace;
  Restaurant favDinnerPlace;

  favLunchPlace.SetName("Central Deli");
  favLunchPlace.SetRating(4);

  favDinnerPlace.SetName("Friends Cafe");
  favDinnerPlace.SetRating(5);

  cout << "My favorite restaurants: " << endl;
  favLunchPlace.Print();
  favDinnerPlace.Print();

  return 0;
}
```

Implicit and Explicit Parameters

- There are two parameters when calling `SetRating` method:

```
void Restaurant::SetRating(int userRating) {
  rating = userRating;
}
```

```
favLunchPlace.SetRating(4);
```

Implicit Parameter

Explicit Parameter

  - This call executes:

```
favLunchPlace.rating = 4;
```

Implicit and Explicit Parameters

- There are two parameters when calling SetRating method:

```
void Restaurant::SetRating(int rating) {
  this->rating = rating;
}
```

- The keyword this refers to the implicit parameter
- Clearly distinguishes between instance fields and local variables
- Arrow operator (->): Member selection via pointer. It is similar to the dot (.) operator. (Will be discussed in the pointer again)

```
favLunchPlace.SetRating(4);
```

Implicit Parameter

Explicit Parameter

- This call executes:

```
favLunchPlace.rating = 4;
```

Review - Access Specifiers

- `public`
  - Members are accessible from outside the class
- `private`
  - Members cannot be accessed (or viewed) from outside the class
  - Declaring data member with access specifier `private` is known as **data hiding** → name and rating are **encapsulated** (hidden) in the object
  - Have to implement member functions to access the private member data
- `protected`
  - Members cannot be accessed from outside the class, however, they can be accessed in inherited classes. (will be discussed later)

https://www.w3schools.com/cpp/cpp_access_specifiers.asp

Review - Accessors and Mutators

- A class's `private` data members can be manipulated only by member functions of that class
- Any statement that call the object's member function from outside the object
  → Call the class's `public` member functions to request the class's services for particular objects of the class
- Accessors (or getters)
  - Public member functions to allow clients of the class to get (i.e., obtain the value of) private data members
  - The member function names usually begin with `Get`. E.g., `GetName`
- Mutators (or setters)
  - Public member functions to allow clients of the class to set (i.e., assign values to) private data members
  - The member function names usually begin with `Set`. E.g., `SetRating`

Utility Functions (Helper Functions)

- A private member function that supports the operation of the class's other member function
- Are not intended to be used by clients of a class

```
class MyClass {
   public:
      void Fct1();
   private:
      int numA;
      int FctX();
};

void MyClass::Fct1() {
   numA = FctX();
   ...
}

int MyClass::FctX() {
   ...
}
```

OK

```
int main() {
   MyClass SomeObj;

   SomeObj.Fct1();

   ...

   SomeObj.FctX();

   return 0;
}
```

OK

Error

Initialization and Constructors

Review - Data Members

- Update the `Restaurant` class as follow:

Restaurant.h

```
class Restaurant {
  public:
    void Print();

  private:
    string name = "NoName";
    int rating = -1;
};
```

Data member initialization

Restaurant.cpp

```
void Restaurant::Print()
{
  cout << name << "--" << rating << endl;
}
```

Favoriate.cpp

```
int main()
{
  Restaurant r;
  r.Print();
  return 0;
}
```

```
~/CS-2370-Exercise$ ./main
NoName---1
~/CS-2370-Exercise$
```

Constructor

- A special class member function called automatically when a variable of the class type is declared
  - Has the same name as the class
  - Cannot return values, so they cannot specify a return type (not even void)
  - Are declared public normally
- Default constructor: a constructor callable without arguments
  - If a class does not explicitly include any constructor, the compiler provides a default constructor
    - Create an instance of the class
    - Implicitly calls each data member's default constructor
  - If you define a constructor with arguments, C++ will not implicitly create a default constructor for that class

Constructor

- Default constructor: a constructor callable without arguments

Restaurant.h

```
class Restaurant {
  public:
    Restaurant();
    void SetName(string name);
    void SetRating(int userRating);
    void Print();

  private:
    string name;
    int rating;
};
```

Constructor in Restaurant.cpp

```
Restaurant::Restaurant() {
  name = "NoName";
  rating = -1;
}
```

Favoriate.cpp

```
int main()
{
  Restaurant r;
  r.Print();

  return 0;
}
```

```
~/CS-2370-Exercise$ ./main
NoName---1
~/CS-2370-Exercise$
```

Review - Constant Variables

- A constant is a value or an identifier whose value cannot be altered in a program.
- Declare a constant variable: use `const` keyword
- Examples:

```
const double SPEED_OF_SOUND   = 761.207;
const double SECONDS_PER_HOUR = 3600.0;
```

- A common convention is to name constant variables using upper case letters with words separated by underscores.

Review - Passing an Array to a Function

- Prevent the function to modify the array parameter: using `const`
- Example

```
double CalculateAverage(const double scoreVals[], int numVals) {
   int i;
   double scoreSum = 0;

   for (i = 0; i < numVals; ++i) {
      scoreSum = scoreSum + scoreVals[i];
   }

   return scoreSum / numVals;
}
```

Constant Member Function

- Syntax of defining a constant member function
  - For function declaration within a class
    ```
    <return_type> <function_name>() const;
    ```
  - For function definition within the class declaration.
    ```
    <return_type> <function_name>() const
    {
            //function body
    }
    ```
  - For function definition outside the class.
    ```
    <return_type> <class_name> : : <function_name>() const
    {
            //function body
    }
    ```
- Not to allow the function to modify the object on which they are called
- It is recommended to practice to make as many functions `const` as possible so that accidental changes to objects are avoided

https://www.geeksforgeeks.org/const-member-functions-c/

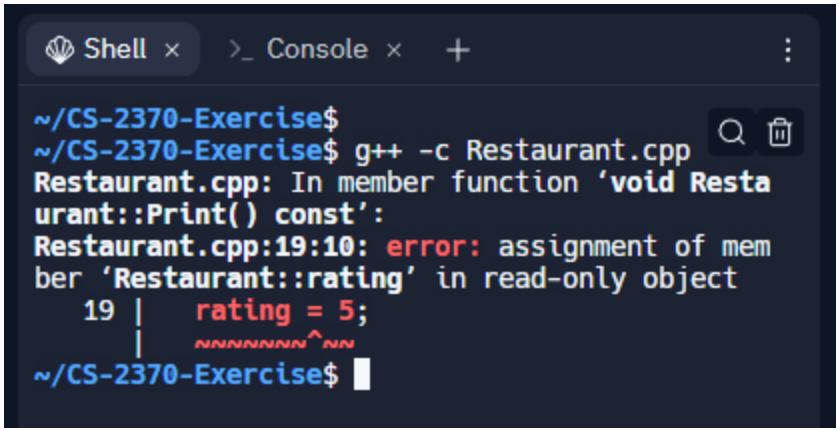
Constant Member Function

- Example: update the `Print` function as a constant member function

Restaurant.h

```
class Restaurant {
  public:
    Restaurant();
    void SetName(string name);
    void SetRating(int userRating);
    void Print() const;

  private:
    string name;
    int rating;
};
```



Print() in Restaurant.cpp

```
void Restaurant::Print() const
{
  rating = 5;
  cout << name << "--" << rating << endl;
}
```

Cause the error. A constant member function can not update the object which they are called

Constant Member Function

- Update the `Print()` method again

Restaurant.h

```
class Restaurant {
  public:
    Restaurant();
    void SetName(string name);
    void SetRating(int userRating);
    void Print() const;

  private:
    string name;
    int rating;
};
```

Print() in Restaurant.cpp

```
void Restaurant::Print() const
{
  cout << name << "--" << rating << endl;
}
```

Constant Object (Instance)

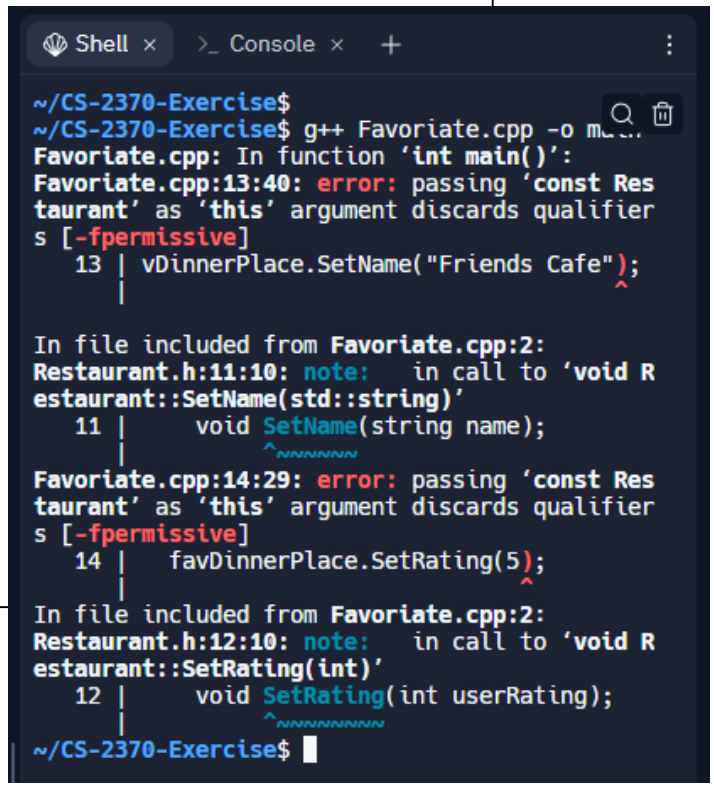
- Define an object is not modifiable by using `const` keyword
- Example:

```
int main()
{
  Restaurant favLunchPlace;
  const Restaurant favDinnerPlace;

  favLunchPlace.SetName("Central Deli");
  favLunchPlace.SetRating(4);

  favDinnerPlace.SetName("Friends Cafe");
  favDinnerPlace.SetRating(5);

  return 0;
}
```



Constant Member Data

- Add a constant member data in the `Restaurant` class

Restaurant.h

```
#include <iostream>
#include <cstdlib>
using namespace std;

class Restaurant {
  public:
    Restaurant();
    void SetName(string name);
    void SetRating(int userRating);
    void Print() const;

  private:
    string name;
    int rating;
    const int id = rand();
};
```

Constant Member Data

- Add a constant member data in the `Restaurant` class

Print() in Restaurant.cpp

```
void Restaurant::Print() const
{
  cout << id << ": " << name << "--" << rating << endl;
}
```

Favoriate.cpp

```
int main()
{
  Restaurant favLunchPlace, favDinnerPlace;

  favLunchPlace.SetName("Central Deli"); favLunchPlace.SetRating(4);
  favDinnerPlace.SetName("Friends Cafe"); favDinnerPlace.SetRating(5);

  cout << "My favorite restaurants: " << endl;
  favLunchPlace.Print(); favDinnerPlace.Print();

  return 0;
}
```

Constant Member Data

- Move the initializer to the default constructor

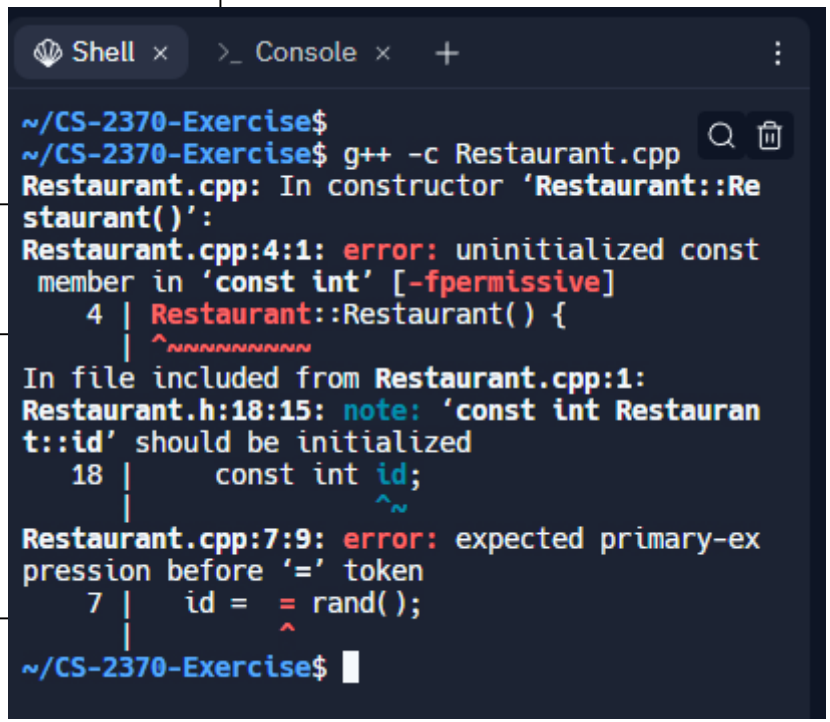
Restaurant.h

```
class Restaurant {
  ...
  private:
    string name;
    int rating;
    const int id;
};
```

The in-class initialization occurs before the constructor runs

Constructor in Restaurant.cpp

```
Restaurant::Restaurant() {
  name = "NoName";
  rating = -1;
  id =  = rand();
}
```



Constant Member Data

- Using constructor initializer list instead
  - An alternative approach for initializing data member in a constructor
  - Coming after a colon and consisting of a comma-separated list of `variableName(initValue)` items

Constructor in Restaurant.cpp

```
Restaurant::Restaurant() : name("NoName"), rating(-1), id(rand()) {

}
```

Constant Member Data

- Initializing them in the body of the constructor means that they get "initialized" twice
  - first they are automatically default-initialized
  - and then they are reassigned by your assignment statements in the constructor body

```
#include <iostream>
#include <vector>
using namespace std;

class SampleClass {
   public:
      SampleClass();
      void Print() const;

   private:
      vector<int> itemList;
};

SampleClass::SampleClass() {
   // itemList gets default constructed, size 0
   itemList.resize(2);
}
```

```
#include <iostream>
#include <vector>
using namespace std;

class SampleClass {
   public:
      SampleClass();
      void Print() const;

   private:
      vector<int> itemList;
};

SampleClass::SampleClass() : itemList(2) {
   // itemList gets constructed with size 2
}
```

  - Member objects requiring constructor arguments should ALWAYS appear in the initializer list

Special Member Functions

Special Member Functions

- Under *certain conditions*, the followings will be automatically generated by the compiler (“Synthesized”):
  - Default constructor (no-arg constructor)
  - Copy constructor (takes another instance as a parameter)
  - Copy-Assignment operator (overwrites an existing object)
  - Destructor
- Synthesized default constructor
  - The constructor called when objects of a class are declared, but are not initialized with any arguments
  - If a class definition has no constructors, the compiler assumes the class to have an implicitly defined default constructor

https://cplusplus.com/doc/tutorial/classes2/

Special Member Functions

- Synthesized copy constructor
  - When an object is passed a named object of its own type as argument, its copy constructor is invoked in order to construct a copy.
  - If a class has no custom copy nor move constructors (or assignments) defined, an implicit copy constructor is provided.
  - Will be discussed regarding how to define the copy constructor in the Pointers chapter.
- Synthesized copy-assignment operator
  - Assigns all members from an existing object to another existing object
  - Is defined implicitly if a class has no custom copy nor move assignments (nor move constructor) defined
  - Will be discussed regarding how to define the copy-assignment constructor in the Pointers chapter

https://cplusplus.com/doc/tutorial/classes2/

Special Member Functions

- Synthesized destructor
  - Responsible for the necessary cleanup needed by a class when its lifetime ends.
  - Is defined implicitly if a class has no customized destructor defined
  - Destroys each non-`static` member in the reverse order from that in which the object was created. In consequence
  - Does not delete the object pointed to by a pointer member.

Static Data Members and Functions

Review - Object-Oriented Programming

- Class vs. Instance
  - Class: The blueprint of the car
  - Instance: A completed car built based on the blueprint
- Key Characteristics of Objects
  - Behavior: what you can do with this object, or what methods you can apply to it.
  - State: how the object react when you invoke those methods.
  - Identity: How the object is distinguished from other that may have the same behavior and state.

Static Data Members

- The `static` variable in C language
  - Is allocated in memory only once during a program's execution
  - Reside in the program's static memory region for the entire program
  - Example:

```
#include <stdio.h>

int function1() {
    static int count1 = 0;     count1++;
    printf("count of function 1 is %d\n", count1);
}

int function2() {
    int count2 = 0;      count2++;
    printf("count of function 2 is %d\n", count2);
}

int main() {
    function1();
    function1();
    function2();
    function2();
    function2();
}
```

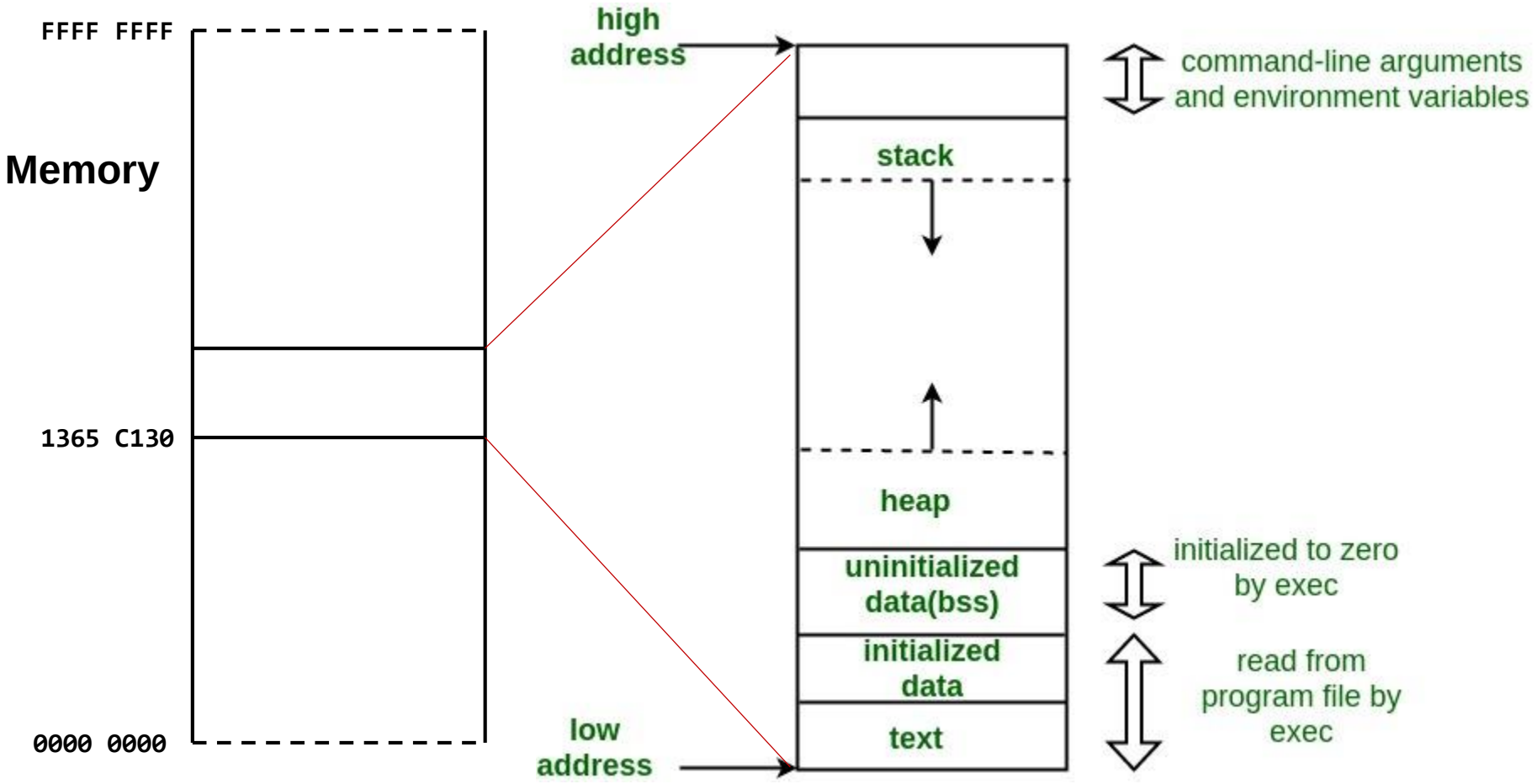
```
count of function 1 is 1
count of function 1 is 2
count of function 2 is 1
count of function 2 is 1
count of function 2 is 1


...Program finished with exit code 0
Press ENTER to exit console.
```

https://www.scaler.com/topics/static-variables-in-c/

Review - Memory Layout

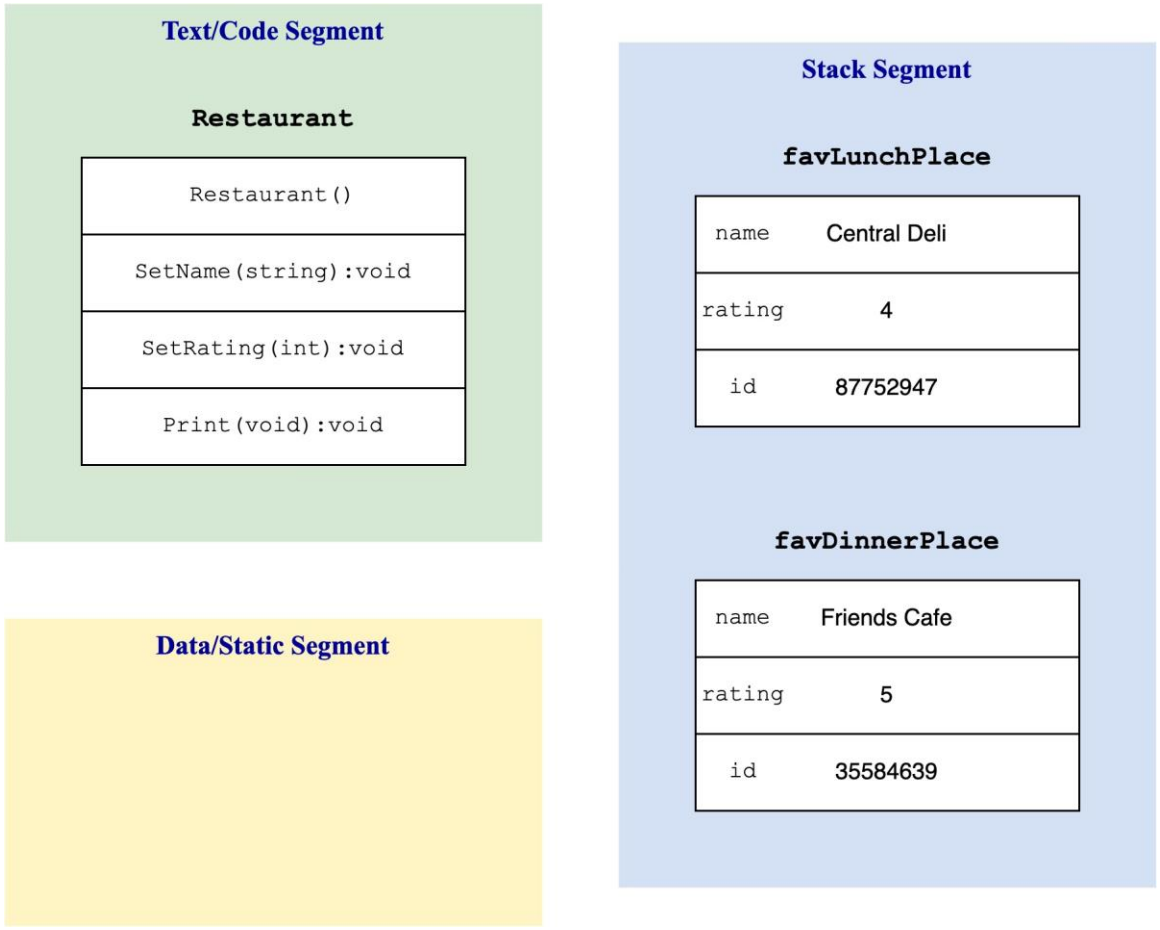
- Memory
  - Variables correspond to locations in the computer's memory



https://www.geeksforgeeks.org/memory-layout-of-c-program/

Static Data Members

- The memory layout in the previous program



Static Data Members

- Adding a static data member in the `Restaurant` class

Restaurant.h

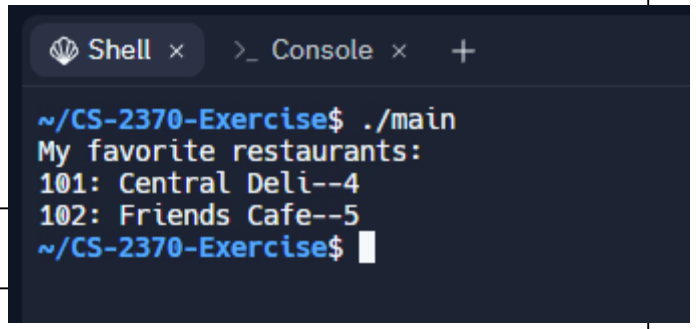
```
class Restaurant {
  public:
    Restaurant();
    void SetName(string name);
    void SetRating(int userRating);
    void Print() const;

  private:
    string name;
    int rating;
    const int id;
    static int nextId;
};
```

Restaurant.cpp

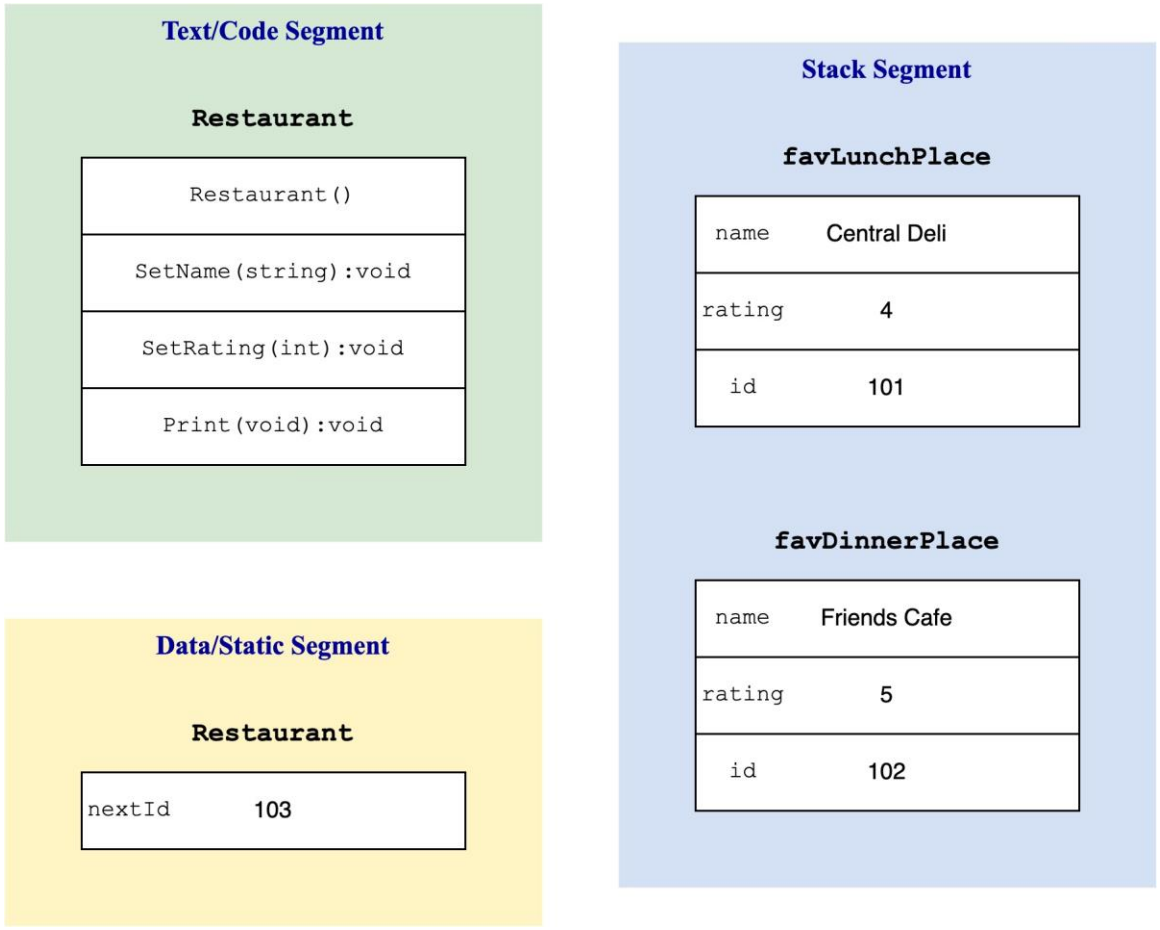
```
int Restaurant::nextId = 101;

Restaurant::Restaurant() : name("NoName"), rating(-1), id(nextId) {
  nextId++;
}
```



Static Data Members

- The memory layout with the static data member



Static Member Functions

- A class function that is independent of class objects/instances
- Usage
  - Access and mutate private static data members from outside the class
  - Can only access a class' static data members
    → the `this` parameter is not passed to the static member function
  - Can be used even if no objects of that class have been instantiated

Static Member Functions

- Example
  - Add a static member function to access the static data member without any object/instance instantiated

Restaurant.h

```
class Restaurant {
  public:
    ...
    static int GetNextId();

  private:
    ...
    static int nextId;
};
```

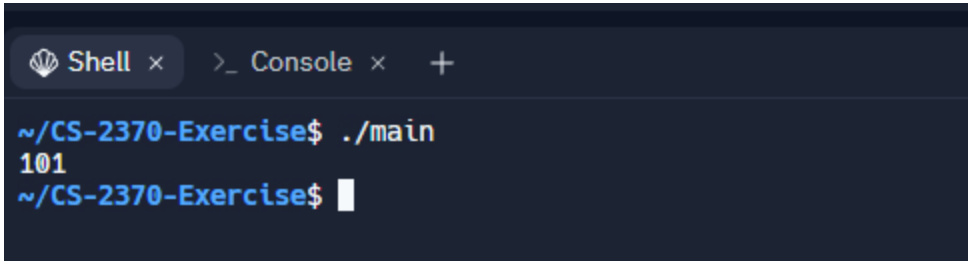
Restaurant.cpp

```
int Restaurant::GetNextId() {
  return nextId;
}
```

Favoriate.cpp

```
int main()
{
  cout << Restaurant::GetNextId() << endl;

  return 0;
}
```



Static Member Functions

- Add a static member function to access a non-static data member

Restaurant.h

```
class Restaurant {
  public:
    ...
    static int GetId();

  private:
    ...
    const int id;
};
```

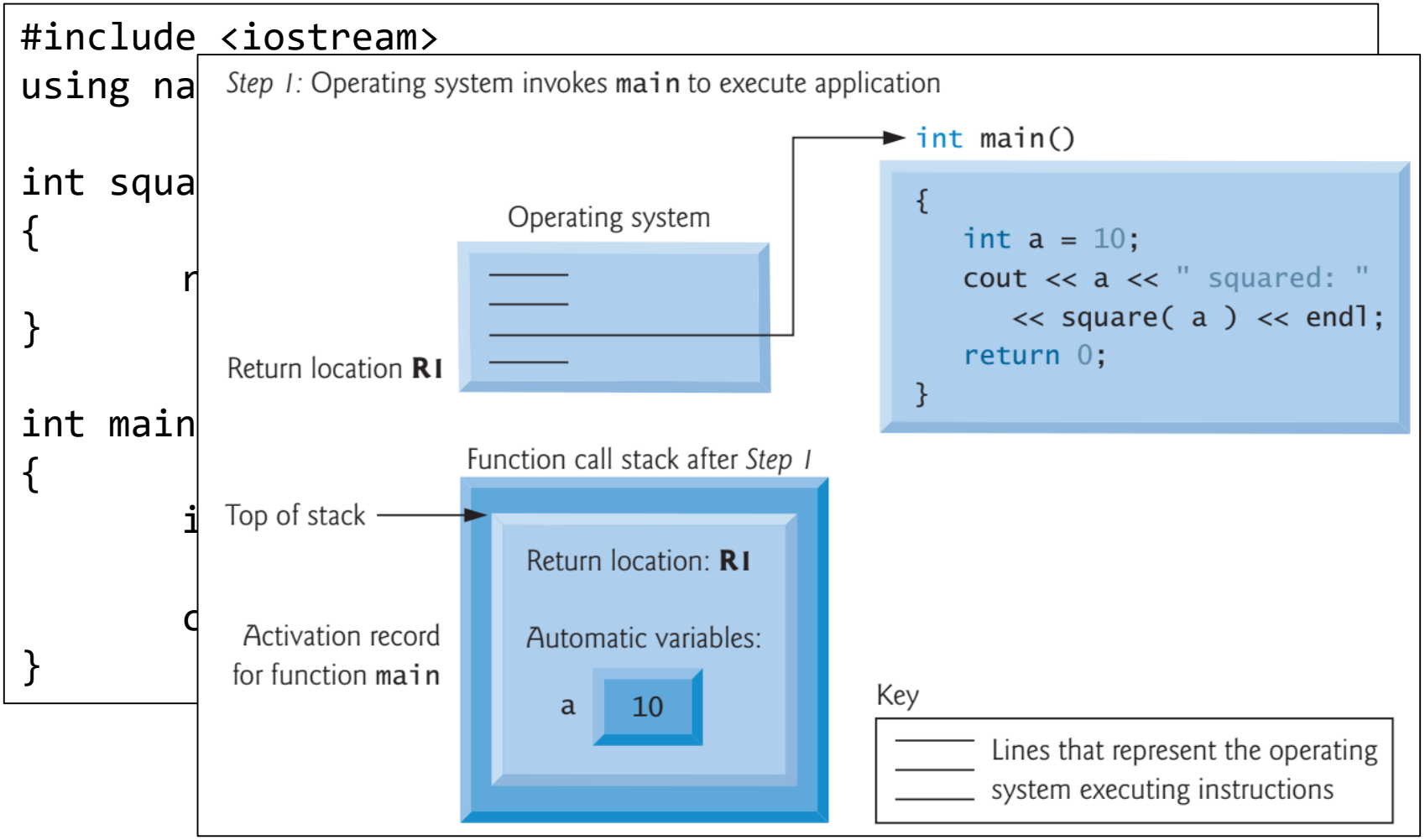
Restaurant.cpp

```
int Restaurant::GetId() {
  return id;
}
```

Inline Functions

Review - Function Call Stack and Stack Frames

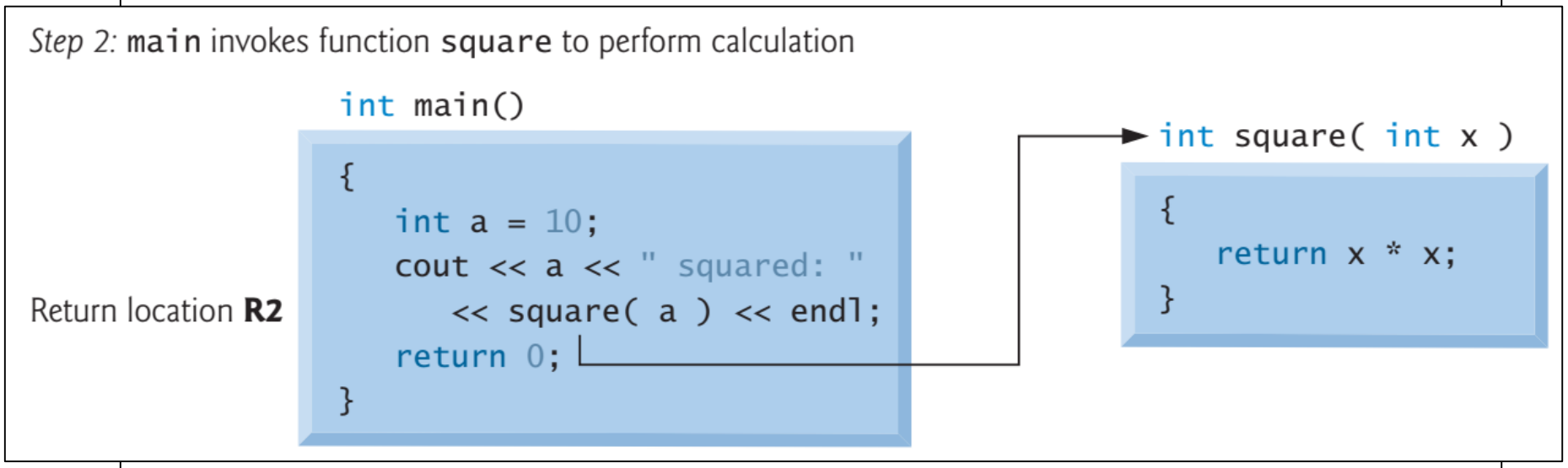
- Example



Review - Function Call Stack and Stack Frames

- Example

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




```
        cout << a << " squared: " << << endl;
}
```

Review - Function Call Stack and Stack Frames

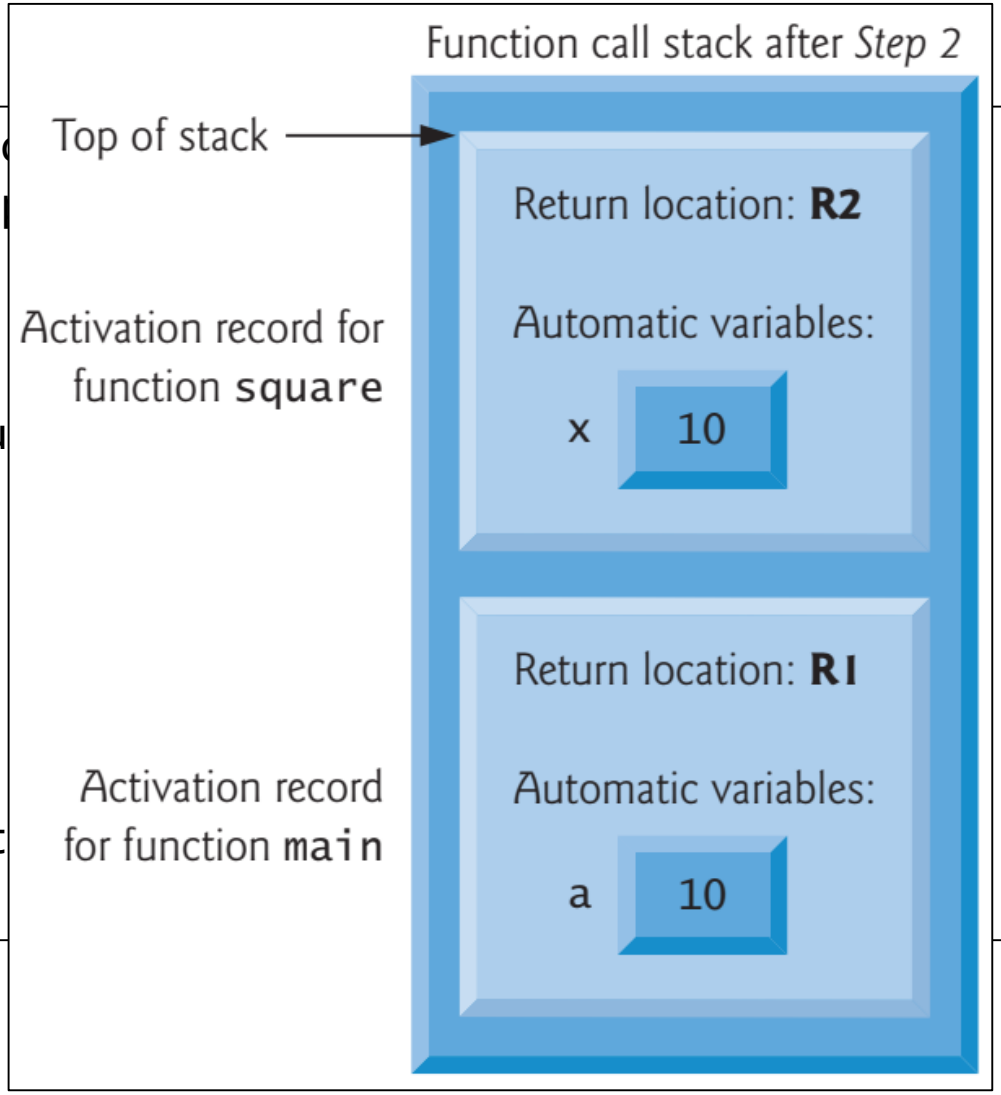
- Example

```
#include <i
using namesp

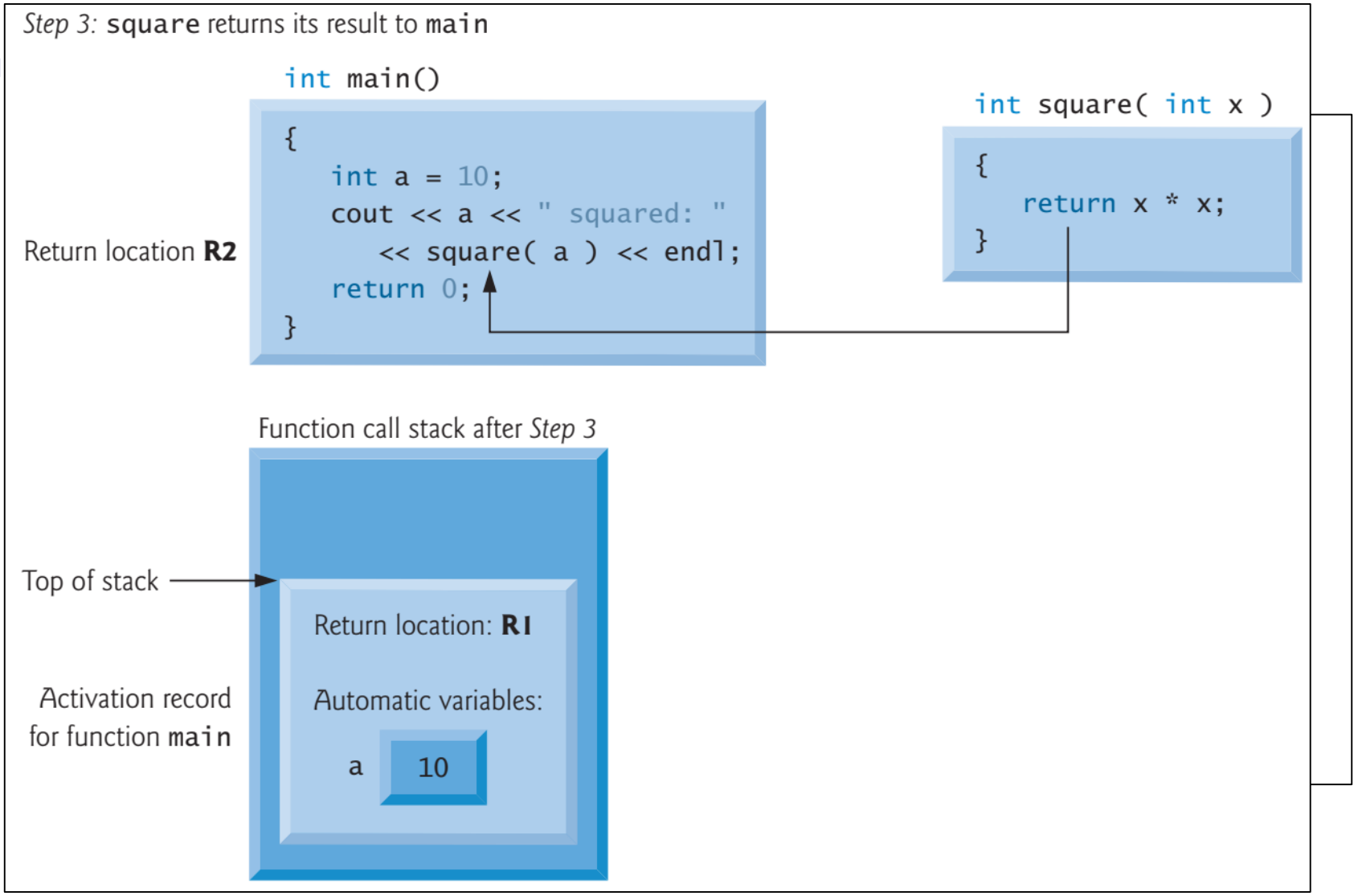
int square(
{
        retu
}

int main()
{
        int

        cout
}
```



Review - Function Call Stack and Stack Frames



Inline Functions

- When the program executes the function call instruction, the CPU
  - Stores the memory address of the instruction following the function call
  - Copies the arguments of the function on the stack
  - Transfers control to the specified function
- After the function call, the CPU
  - Stores the function return value in the predefined memory location/register
  - Returns controls to the calling function
- The process become overhead if the execution time of function is less than a switching time from the caller function to callee
  - Large function: the overhead is usually insignificant
  - Small, commonly-used function: the time needed to make the function call is often a lot more than the time needed to actually execute the function's call
    → execution time of small function is less than the switching time

https://www.geeksforgeeks.org/inline-functions-cpp/

Inline Functions

- Inline function
  - Reduce the function call overhead
  - The function is expended in line when it is called:
    the whole code of the inline function gets inserted or substituted at the point of inline function call
  - Is performed by the C++ compiler at compile time
- Syntax
  - ```
    inline return-type function-name(parameters)
    {
            // function code
    }
    ```

https://www.geeksforgeeks.org/inline-functions-cpp/
    ```

Inline Functions

- Example

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

inline int cube(int s)
{
    return s*s*s;
}

int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
}
```

https://www.geeksforgeeks.org/inline-functions-cpp/

Inline Functions

- Inlining is only a request to the compiler, not a command.
- Compiler may not perform inlining in such circumstances like:
  - If a function contains a loop. (for, while, do-while)
  - If a function contains static variables.
  - If a function is recursive.
  - If a function return type is other than void, and the return statement doesn’t exist in function body.
  - If a function contains switch or goto statement.

https://www.geeksforgeeks.org/inline-functions-cpp/

Inline Member Functions

- All the functions defined inside the class are implicitly inline.

```
class S
{
public:
    inline int square(int s) // redundant use of inline
    {
        // this function is automatically inline
        // function body
    }
};
```

Review - Define a Class

- Using the `class` keyword
- Example

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

class Restaurant {
  public:
    void Print()
    {
      cout << "Restaurant and Rating" << endl;
    }
};

int main()
{
  Restaurant r;
  r.Print();
  return 0;
}
```

The Print function is an inline function implicitly

Inline Member Functions

- Explicitly declare inline functions in the class
  - Declare the function inside the class
  - Define the function outside the class using inline keyword.
  - Placed in the header file (https://stackoverflow.com/questions/4769479/) if it has
- Example

Restaurant.h

```
class Restaurant {
  public:
    void Print() const;
    …
};

inline void Restaurant::Print() const
{
  cout << id << ": " << name << "--" << rating << endl;
}
```

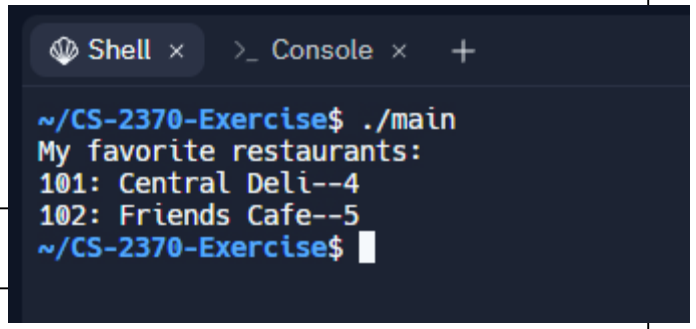
Review - Static Data Members

- Adding a static data member in the `Restaurant` class

Restaurant.h

```
class Restaurant {
  public:
    Restaurant();
    void SetName(string name);
    void SetRating(int userRating);
    void Print() const;

  private:
    string name;
    int rating;
    const int id;
    static int nextId;
};
```



Restaurant.cpp

```
int Restaurant::nextId = 101;

Restaurant::Restaurant() : name("NoName"), rating(-1), id(nextId) {
  nextId++;
}
```

Inline Statement

- static data member can be initialized in the heading file with inline statements.
- Update the Restaurant class as follow:

Restaurant.h

```
class Restaurant {
  ...
  private:
    string name;
    int rating;
    const int id;
    inline static int nextId = 101;
};
```

Restaurant.cpp

```
int Restaurant::nextId = 101;
```

Remove this line

```
Restaurant::Restaurant() : name("NoName"), rating(-1), id(nextId) {
  nextId++;
}
```

Overloading

Review - Function Signatures

- In regular function
  - The name and the parameter-type-list of a function
- In a class member
  - The name and the parameter-type-list of a function
  - The class, concept, concept map, or the namespace
- Functions in the same scope must have unique signatures
- The compiler uses to perform overload resolution

https://stackoverflow.com/questions/2322736/what-is-the-difference-between-function-declaration-and-signature

Review - Function Overloading

- Several functions are defined with same name
  - Have different signatures
    → Have different number, types, and order of the arguments
- Example

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

void PrintDate(int currDay, int currMonth, int currYear) {
   cout << currMonth << "/" << currDay << "/" << currYear;
}

void PrintDate(int currDay, string currMonth, int currYear) {
   cout << currMonth << " " << currDay << ", " << currYear;
}

int main() {

   PrintDate(30, 7, 2012);
   cout << endl;

   PrintDate(30, "July", 2012);
   cout << endl;

   return 0;
}
```

```
7/30/2012
July 30, 2012
```

Review - Function Overloading

- Function Overloading
  - A program has two functions with the same name but differing in the number or types of parameters
- How the compiler differentiates overloaded function
  - By their signatures, combining with a function's name and its parameter types (in order)
  - Name mangling or name decoration
    Encodes each function identifier with the number and types of its parameters → enable type-safe linkage
  - Type-safe linkage: ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters

Constructor Overloading

- Provide different initialization values when creating a new object
- Define multiple constructors differing in parameter types
- Example

```
class Restaurant {
   public:
      Restaurant();
      Restaurant(string initName, int initRating);

   ...
};

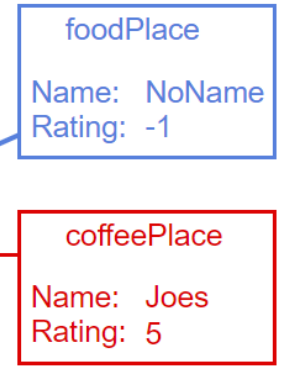
// Default constructor
Restaurant::Restaurant() {
   name = "NoName";
   rating = -1;
}

// Another constructor
Restaurant::Restaurant(string initName, int initRating) {
   name = initName;
   rating = initRating;
}

int main() {
   Restaurant foodPlace;              // Calls default constructor

   Restaurant coffeePlace("Joes", 5); // Calls another constructor

   ...
}
```



Review - Constructor

- A special class member function called automatically when a variable of the class type is declared
  - Has the same name as the class
  - Cannot return values, so they cannot specify a return type (not even void)
  - Are declared public normally
- Default constructor: a constructor callable without arguments
  - If a class does not explicitly include any constructor, the compiler provides a default constructor
    - Create an instance of the class
    - Implicitly calls each data member's default constructor
  - If you define a constructor with arguments, C++ will not implicitly create a default constructor for that class

Constructor Overloading

- If any constructor defined, should define default
  - If a programmer defines any constructor, the compiler does not implicitly define a default constructor
  - The programmer should explicitly define a default constructor so that a declaration like `MyClass x;` remains supported.

```
class Restaurant {
   public:
      Restaurant(string initName, int initRating);

      // No other constructors
      ...
};

int main() {
   Restaurant foodPlace;
   ...
}
```

```
tmp1.cpp:37:15: error: no matching
constructor for initialization of
      'Restaurant'
   Restaurant foodPlace;
```

Constructor Overloading

- Constructors with default parameter values
  - If the default parameters allow the constructor to be called without arguments, then that constructor can serve as the default constructor.
- Example

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

class Restaurant {
   public:
      Restaurant(string initName = "NoName", int initRating = -1);
      void Print();

    private:
      string name;
      int rating;
};

Restaurant::Restaurant(string initName, int initRating) {
   name = initName;
   rating = initRating;
}
```

Operator Overloading

- Redefine the functionality of built-in operators like +, -, and *, to operate on programmer-defined objects

Without operator overloading

```
TimeHrMn time1(3, 22);
TimeHrMn time2(2, 50);
TimeHrMn timeTot;
timeTot.hours = time1.hours + time2.hours;
timeTot.minutes = time1.minutes + time2.minutes;

timeTot.Print();
```

Console:

H:5, M:72

With operator overloading

```
TimeHrMn time1(3, 22);
TimeHrMn time2(2, 50);
TimeHrMn timeTot;
timeTot = time1 + time2;

timeTot.Print();
```

Console:

H:5, M:72

- Example
  - <<: is used both as the stream insertion operator and as the bitwise left-shift operator

Operator-Overloading Function

- The function name starts with the keyword `operator` followed by the symbol for the operator being overloaded
  - Example:
    `operator+` would be used to overload the addition operator (`+`)
- When operators are overloaded as member functions, they must be `non-static` → they must be called on an object of the class and operate on the object
- Operators that can not be overloaded

Operators that cannot be overloaded			
.	.*	::	?:

Fig. 11.2 | Operators that cannot be overloaded.

Operator-Overloading Function

- Define `TimeHrMn` class without operator-overloading functions

TimeHrMn.h

```
#ifndef TIMEHRMN_H
#define TIMEHRMN_H

#include <iostream>
using namespace std;

class TimeHrMn {
  public:
    TimeHrMn(int hours = 0, int minutes = 0);
    void Print() const;
  private:
    int hours;
    int minutes;
};

#endif
```

Operator-Overloading Function

- Define `TimeHrMn` class without operator-overloading functions

TimeHrMn.cpp

```
#include "TimeHrMn.h"

TimeHrMn::TimeHrMn(int hours, int minutes)
{
  int carry = 0;
  if (minutes < 0) {
    this->minutes = 0;
  } else if (minutes >= 0 && minutes < 60) {
    this->minutes = minutes;
  } else {
    this->minutes = minutes % 60;
    carry = minutes / 60;
  }
  this-> hours = hours + carry;

}

void TimeHrMn::Print() const
{
  cout << "H: " << hours << ", M: " << minutes << endl;
}
```

Operator-Overloading Function

- Define `TimeHrMn` class without operator-overloading functions

TimeHrMnTest.cpp

```
#include <iostream>
#include "TimeHrMn.h"
using namespace std;

int main()
{
  TimeHrMn time1(3, -5);
  TimeHrMn time2(2, 80);

  time1.Print();
  time2.Print();

  return 0;
}
```

Operator-Overloading Function

- Overload + operator in the `TimeHrMn` class

TimeHrMn.h

```
class TimeHrMn {
  public:
    TimeHrMn(int hours = 0, int minutes = 0);
    TimeHrMn operator+(TimeHrMn rhs);
    void Print() const;
  private:
    int hours;
    int minutes;
};
```

TimeHrMnTest.cpp

```
int main()
{
  TimeHrMn time1(3, 35);
  TimeHrMn time2(2, 40);
  TimeHrMn time3 = time1 + time2;

  time1.Print();
  time2.Print();
  time3.Print();

  return 0;
}
```

TimeHrMn.cpp

```
TimeHrMn TimeHrMn::operator+(TimeHrMn rhs)
{
  TimeHrMn timeTotal(this->hours + rhs.hours, this->minutes + rhs.minutes);

  return timeTotal;
}
```

Review - Implicit and Explicit Parameters

- There are two parameters when calling `SetRating` method:

```
void Restaurant::SetRating(int rating) {
  this->rating = rating;
}
```

- The keyword this refers to the implicit parameter
- Clearly distinguishes between instance fields and local variables
- Arrow operator (`->`): Member selection via pointer. It is similar to the dot (`.`) operator. (Will be discussed in the pointer again)

```
favLunchPlace.SetRating(4);
```

Implicit Parameter

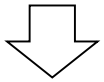
Explicit Parameter

- This call executes:

```
favLunchPlace.rating = 4;
```

Operator-Overloading Function

```
TimeHrMn time3 = time1 + time2;
```



```
TimeHrMn time3 = time1.operator+(time2);
```

Implicit Parameter

Explicit Parameter

```
TimeHrMn TimeHrMn::operator+(TimeHrMn rhs)
{
  TimeHrMn timeTotal(hours + rhs.hours, minutes + rhs.minutes);

  return timeTotal;
}
```

Operator-Overloading Function

- Add the second `operator+` method in the `TimeHrMn` class
  - A programmer can define several functions that overload the same operator

TimeHrMn.h

```
class TimeHrMn {
  public:
    TimeHrMn(int hours = 0, int minutes = 0);
    TimeHrMn operator+(TimeHrMn rhs);
    TimeHrMn operator+(int hours);
    ...
};
```

TimeHrMnTest.cpp

```
int main()
{
  TimeHrMn time1(3, 35);
  TimeHrMn time2 = time1 + 5;
  TimeHrMn time3 = time1 + time2;

  time1.Print();
  time2.Print();
  time3.Print();

  return 0;
}
```

TimeHrMn.cpp

```
TimeHrMn TimeHrMn::operator+(int hours)
{
  TimeHrMn timeTotal(this->hours + hours, this->minutes);
  return timeTotal;
}
```

Operator-Overloading Function

- Operator-overloading on non-member functions
  - Overload the operator ==

TimeHrMn.h

```
class TimeHrMn {
  public:
    TimeHrMn(int hours = 0, int minutes = 0);
    TimeHrMn operator+(TimeHrMn rhs);
    TimeHrMn operator+(int hours);
    int GetHours() const {return this->hours;};
    int GetMinutes() const {return this->minutes;};
    void Print() const;
  private:
    int hours;
    int minutes;
};

bool operator==(const TimeHrMn& lhs, const TimeHrMn& rhs);
```

Getters are required since `operator==` is not a member function of the `TimeHrMn` class

TimeHrMn.cpp

```
bool operator==(const TimeHrMn& lhs, const TimeHrMn& rhs)
{
  return ((lhs.GetHours() == rhs.GetHours()) && (lhs.GetMinutes() == rhs.GetMinutes()));
}
```

Operator-Overloading Function

- Operator-overloading on non-member functions
  - Overload the operator ==

TimeHrMnTest.cpp

```
int main()
{
  TimeHrMn time1(3, 35);
  TimeHrMn time2 = time1 + 0;
  TimeHrMn time3 = time1 + time2;

  time1.Print();
  time2.Print();
  time3.Print();

  cout << (time1 == time2) << endl;
  cout << (time1 == time3) << endl;

  return 0;
}
```

Operator-Overloading Function

- Application: calling `sort()` function in `algorithm` library
- To use `sort()`, the program must
  - Add `#include <algorithm>` to enable the use of `sort()`.
  - Overload the `<` operator for the programmer-defined class.
  - Call the `sort()` function as `sort(myVector.begin(), myVector.end())`

Operator-Overloading Function

- Application: calling `sort()` function in `algorithm` library

TimeHrMn.h

```
class TimeHrMn {
  public:
    void SetTime(int hours, int minutes) {this->hours = hours; this->minutes = minutes;}
    ...
};

bool operator==(const TimeHrMn& lhs, const TimeHrMn& rhs);
bool operator<(const TimeHrMn& lhs, const TimeHrMn& rhs);
```

TimeHrMn.cpp

```
bool operator<(const TimeHrMn& lhs, const TimeHrMn& rhs)
{
  if (lhs.GetHours() < rhs.GetHours()) {
    return true;
  } else if (lhs.GetHours() > rhs.GetHours()) {
    return false;
  } else {
    if (lhs.GetMinutes() < rhs.GetMinutes())
      return true;
    else
      return false;
  }
}
```

Operator-Overloading Function

- Application: calling `sort()` function in `algorithm` library

TimeHrMnTest.cpp

```
#include <algorithm>

...

int main()
{
  vector<TimeHrMn> times;
  TimeHrMn newTime;

  newTime.SetTime(5, 20);        times.push_back(newTime);
  newTime.SetTime(3, 45);        times.push_back(newTime);
  newTime.SetTime(6, 0);         times.push_back(newTime);

  for (TimeHrMn t : times)
    t.Print();

  sort(times.begin(),times.end());
  cout << "After sort()" << endl;

  for (TimeHrMn t : times)
    t.Print();

  return 0;
}
```

Review - Operator-Overloading Function

- Add the second `operator+` method in the `TimeHrMn` class
  - A programmer can define several functions that overload the same operator

TimeHrMn.h

```
class TimeHrMn {
  public:
    TimeHrMn(int hours = 0, int minutes = 0);
    TimeHrMn operator+(TimeHrMn rhs);
    TimeHrMn operator+(int hours);
    ...
};
```

TimeHrMnTest.cpp

```
int main()
{
  TimeHrMn time1(3, 35);
  TimeHrMn time2 = time1 + 5;
  TimeHrMn time3 = time1 + time2;

  time1.Print();
  time2.Print();
  time3.Print();

  return 0;
}
```

TimeHrMn.cpp

```
TimeHrMn TimeHrMn::operator+(int hours)
{
  TimeHrMn timeTotal(this->hours + hours, this->minutes);
  return timeTotal;
}
```

Operator-Overloading Function

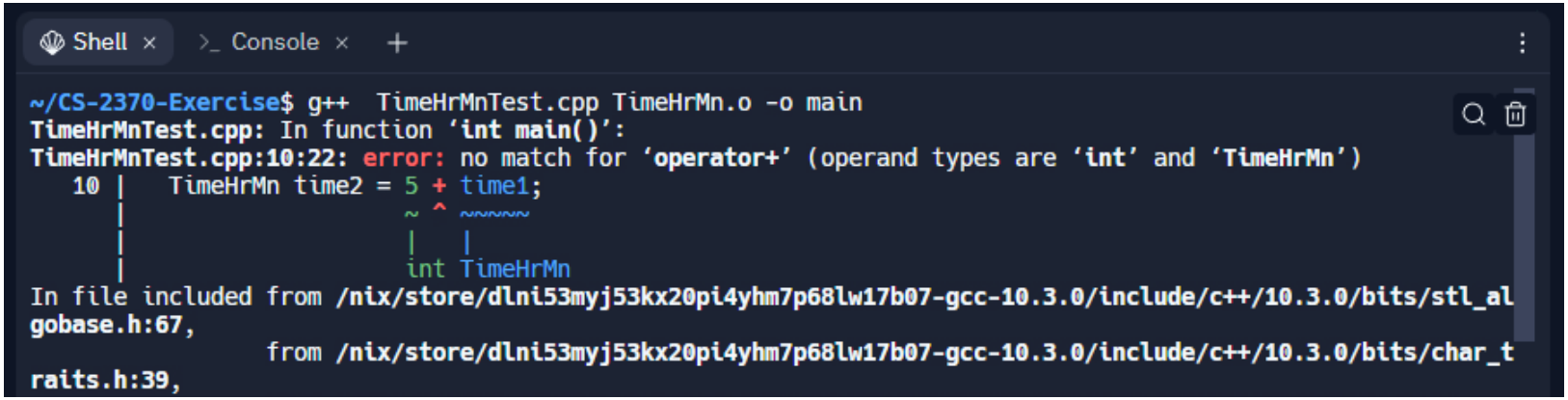
- Test the `TimeHrMn` class with the following program

TimeHrMnTest.cpp

```
int main()
{
  TimeHrMn time1(3, 35);
  TimeHrMn time2 = 5 + time1;
  TimeHrMn time3 = time1 + time2;

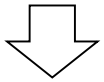
  time1.Print();
  time2.Print();
  time3.Print();

  return 0;
}
```



Review - Operator-Overloading Function

```
TimeHrMn time3 = time1 + time2;
```



```
TimeHrMn time3 = time1.operator+(time2);
```

Implicit Parameter

Explicit Parameter

```
TimeHrMn TimeHrMn::operator+(TimeHrMn rhs)
{
  TimeHrMn timeTotal(hours + rhs.hours, minutes + rhs.minutes);

  return timeTotal;
}
```

Operator-Overloading Function

- `TimeHrMn time2 = time1 + 5;`

`time1.operator+(5)`

`TimeHrMn.operator+(int)`

```
class TimeHrMn {
  public:
    TimeHrMn(int hours = 0, int minutes = 0);
    TimeHrMn operator+(TimeHrMn rhs);
    TimeHrMn operator+(int hours);
    ...
};
```

Defined in the `TimeHrMn` class

- `TimeHrMn time2 = 5 + time1;`

`5.operator+(time1)`

`int.operator+(TimeHrMn)`

Is not defined in any class

Operator-Overloading Function

- Overload + operator as a non-member function and test the program again

TimeHrMn.h

```
class TimeHrMn {
  ...
};

TimeHrMn operator+(int lhs, const TimeHrMn& rhs);
bool operator==(const TimeHrMn& lhs, const TimeHrMn& rhs);
bool operator<(const TimeHrMn& lhs, const TimeHrMn& rhs);
```

TimeHrMn.cpp

```
TimeHrMn operator+(int hours, const TimeHrMn& rhs)
{
  TimeHrMn timeTotal(rhs.GetHours() + hours, rhs.GetMinutes());
  return timeTotal;
}
```

Friend Functions

- A friend function of a class is defined outside that class's scope, yet has the right to access the non-`public` (and `public`) members of the class
- The `friend` declaration can be appear anywhere in the class
- Update the overloaded + operator non-member function as a friend function

TimeHrMn.h

```
class TimeHrMn {
  friend TimeHrMn operator+(int lhs, const TimeHrMn& rhs);
  ...
};

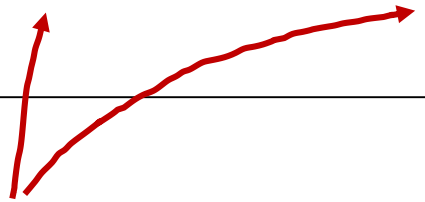
TimeHrMn operator+(int lhs, const TimeHrMn& rhs);
bool operator==(const TimeHrMn& lhs, const TimeHrMn& rhs);
bool operator<(const TimeHrMn& lhs, const TimeHrMn& rhs);
```

Friend Functions

- Update the overloaded + operator non-member function as a friend function

TimeHrMn.cpp

```
TimeHrMn operator+(int hours, const TimeHrMn& rhs)
{
  TimeHrMn timeTotal(rhs.hours + hours, rhs.minutes);
  return timeTotal;
}
```



The friend functions are able to access non-public attributes

Namespaces

Review - The First Program

- A simple C++ program form

```
directives

int main()
{
    statements
}
```

- Example

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

int main() {
  int wage;

  wage = 20;

  cout << "Salary is ";
  cout << wage * 40 * 52;
  cout << endl;

  return 0;
}
```

Namespace

- Allow us to group named entities that otherwise would have global scope into narrower scopes
- Multiple namespace blocks with the same name are allowed.

std

- The namespace defined in **<iostream>**
- **cout** is an object defined in the **std** namespace

If without **using namespace std**

- **cout << "Salary is ";** should be revised to **std::cout << "Salary is ";**

https://www.geeksforgeeks.org/namespace-in-c/

namespaces

- Naming conflict
  - A variable of one scope overlaps (i.e., collide) with a variable of the same name in a different scope
  - Frequently occurred in third-libraries: using the same names for global identifiers (such as functions)

main.cpp

```
#include "auditorium.h"
#include "airplane.h"

int main() {
   Seat concertSeat;
   Seat flightSeat;

   // ...

   return 0;
};
```

auditorium.h

```
class Seat {
   ...
};
```

airplane.h

```
class Seat {
   ...
};
```

namespaces

- Solution in C++: `namespaces`
  - Defines a scope in which identifiers and variables are placed
- Use a `namespace` member: using scope resolution operator (`::`)
  - Syntax: *MyNameSpace*`::`*member*
    Example: `std::cout`
  - A using directive appears before the name is used:
    `using namespace` *MyNameSpace*`;`
    Example: `using namespace std;`
    Example: `using std::cout;`
- Not all namespaces are guaranteed to be unique.
  - Two third-party vendors might inadvertently use the same identifiers for their namespace names

namespaces

- Defining namespaces
  - The body of a namespace is delimited by braces ({})

main.cpp

```
#include "auditorium.h"
#include "airplane.h"

int main() {
    auditorium::Seat concertSeat;
    airplane::Seat flightSeat;

    // ...

    return 0;
};
```

auditorium.h

```
namespace auditorium {
    class Seat {
        ...
    };
}
```

airplane.h

```
namespace airplane {
    class Seat {
        ...
    };
}
```

`std` namespace

- All items in the C++ standard library are part of the `std` namespace (short for standard)
- `using namespace std` is considered bad practice
  - https://dev.to/77bala7790/c-best-practice-1-don-t-simply-use-using-namespace-std-4n6b

Unnamed (Anonymous) Namespace

- Are accessible in the current translation unit (a `.cpp` file and the files it includes)
- Has an implicit using directive
  - Its members appear to occupy the global namespace
  - Is accessible directly and do not have to be qualified with a namespace name
- Example

ns.h

```
namespace {
  int test = 5;
}
```

main.cpp

```
#include <iostream>
#include "ns.h"

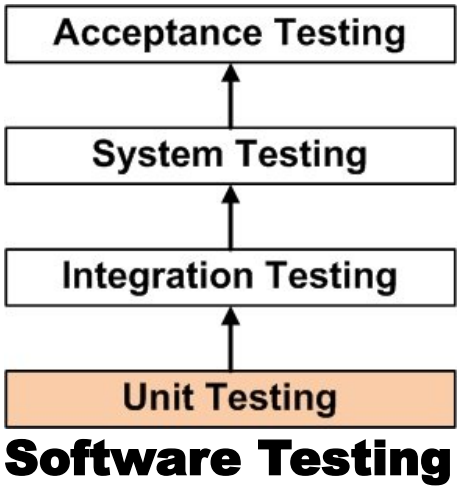
int main() {
  std::cout << test << std::endl;
}
```

Additional Reading for `namespaces`

- https://www.geeksforgeeks.org/namespace-in-c/
- https://www.geeksforgeeks.org/namespace-in-c-set-2-extending-namespace-and-unnamed-namespace/
- https://www.geeksforgeeks.org/namespace-c-set-3-creating-header-nesting-aliasing-accessing/

Unit Testing (Class)

Review - Unit Testing



- Definition: a software testing method
  - Individual units/components of a software are tested
- Unit
  - Is the smallest testable part of any software
  - Has one or a few inputs
  - Has a single output
- A unit in procedural programming
  - Maybe be an individual program, function, procedure, etc.
- A unit in object-oriented programming
  - A method, which may belong to a base/super class, abstract class, or derived/child class

http://softwaretestingfundamentals.com/unit-testing/

Unit Testing (Classes)

- Public member functions
  - Create an object then create testbenches
- Private member functions
  - Define a friend class
  - Test private member functions through the defined friend class.
- Defining a friend class example

```
class A
{
    int x;
          public:

    A()
    {
        x=10;
    }
    friend class B;    //friend class
};
```

```
class B
{
    public:
        void display(A &t)
        {
            cout<<endl<<"The value of x="<<t.x;
        }
};
```

https://www.geeksforgeeks.org/friend-class-function-cpp/

Review - Testbench

- A unit test is typically conducted by creating a testbench (test harness)
  - A separate program whose sole purpose is to check that a function returns correct output values for a variety of input values
  - Each unique set of input values is know and a test vector
  - Example

```
cout << "0:0, expecting 0, got: "    << HrMinToMin(0, 0)  << endl;
cout << "0:1, expecting 1, got: "    << HrMinToMin(0, 1)  << endl;
cout << "0:99, expecting 99, got: "  << HrMinToMin(0, 99) << endl;
cout << "1:0, expecting 60, got: "   << HrMinToMin(1, 0)  << endl;
cout << "5:0, expecting 300, got: "  << HrMinToMin(5, 0)  << endl;
cout << "2:30, expecting 150, got: " << HrMinToMin(2, 30) << endl;
```

```
0:0, expecting 0, got: 0
0:1, expecting 1, got: 1
0:99, expecting 99, got: 99
1:0, expecting 60, got: 0
5:0, expecting 300, got: 0
2:30, expecting 150, got: 30
```

Unit Testing (Classes)

- Features of a good testbench
  - Automatic checks.
    Ex: Values are compared, as in `testData.GetNum1() != 100`. For conciseness, only fails are printed.
  - Independent test cases.
    Ex: The test case for `GetAverage()` assigns new values, vs. relying on earlier values.
  - 100% code coverage:
    Every line of code is executed. A good testbench would have more test cases than below.
  - Includes not just typical values but also border cases:
    Unusual or extreme test case values like 0, negative numbers, or large numbers.

Unit Testing (Classes)

- Regression testing
  - Retest an item like a class anytime that item is changed
  - If previously-passed test cases fail, the item has "regressed“
- Erroneous unit tests
  - Test may fail even if the code being tested is correct

```
StatsInfo testData;
testData.SetNum1(20);
testData.SetNum2(30);
if (testData.GetAverage() != 35) {
   cout << "   FAILED GetAverage for 20, 30" << endl;
}
```

Wrong expected value

```
StatsInfo testData;
testData.SetNum1(20);
testData.SetNum1(30);
if (testData.GetAverage() != 25) {
   cout << "   FAILED GetAverage for 20, 30" << endl;
}
```

Test object's data not properly set