6. Objects and Classes

6.1. Classes

6.2. Constructors and Destructors

6.3. `this` and initializing `const` attributes

6.4. `static` members

6.5. The `string` Class

6.6. The `ifstream` and `ofstream` Classes

6.7. The `tuple` Class

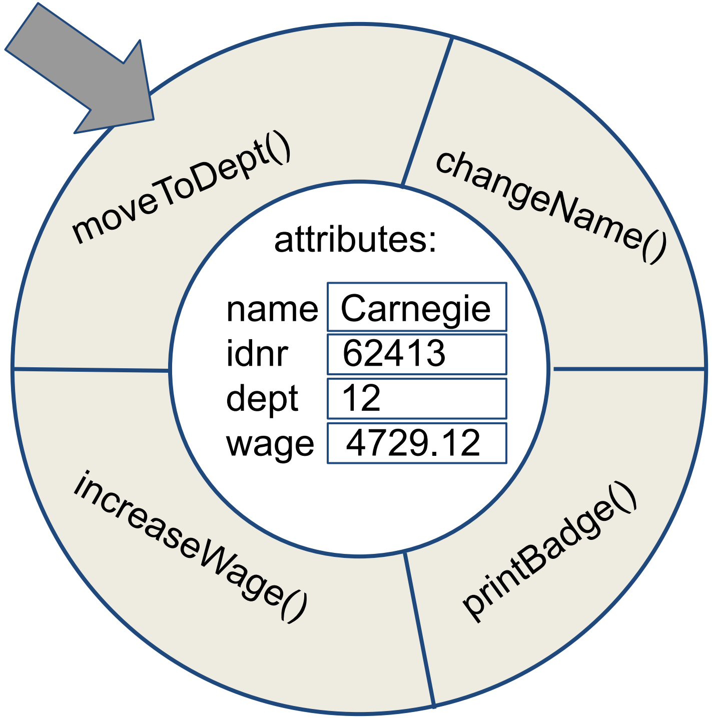
6.1. Classes and Objects

- A **class** essentially defines a new type, containing:
  - a collection of variables (data members, attributes)
  - a set of related operations (member functions, methods)
- An **object** is an instance (an entity) of a class
  - it is called object, since it usually models a real-world object
  - a class then can be viewed as a model or a blueprint for a certain class of objects

6.1. Classes and Objects

Encapsulation: The bundling together of all information, capabilities, and responsibilities (data and functions) into one single object:

```
// create a new employee:
Employee user("Carnegie", 62413,
              12, 4729.12);

// move the user to department 17:
user.movetoDept(17);
// change the user's name:
user.changeName("Carnegie-Mellon");
```

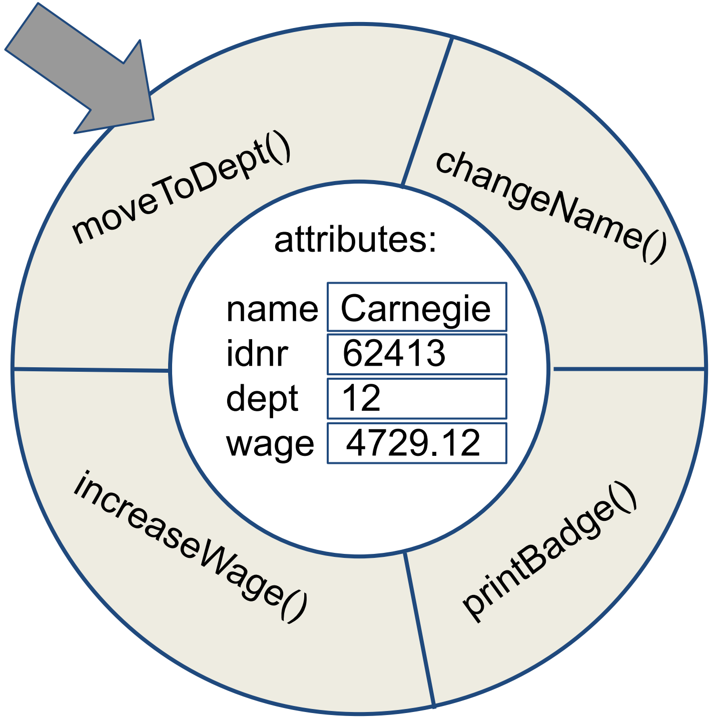


6.1. Classes and Objects

Encapsulation has two properties:

1) Data protection: Attributes are **private**, access to attributes goes through the available methods

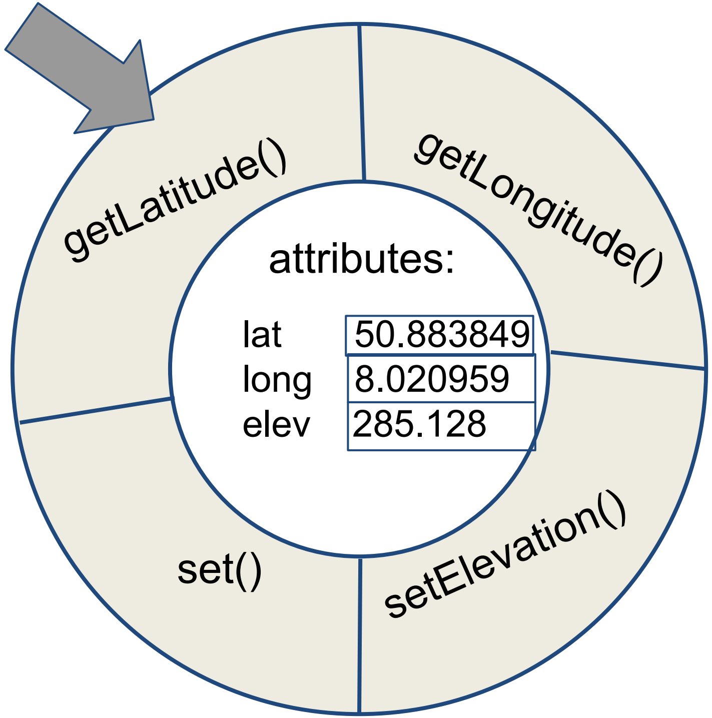
2) Information hiding: Internal implementation is hidden from external code, only the **public** interface of the class is accessible



6.1. Classes and Objects

Access to object's attributes goes through the available methods. Example:

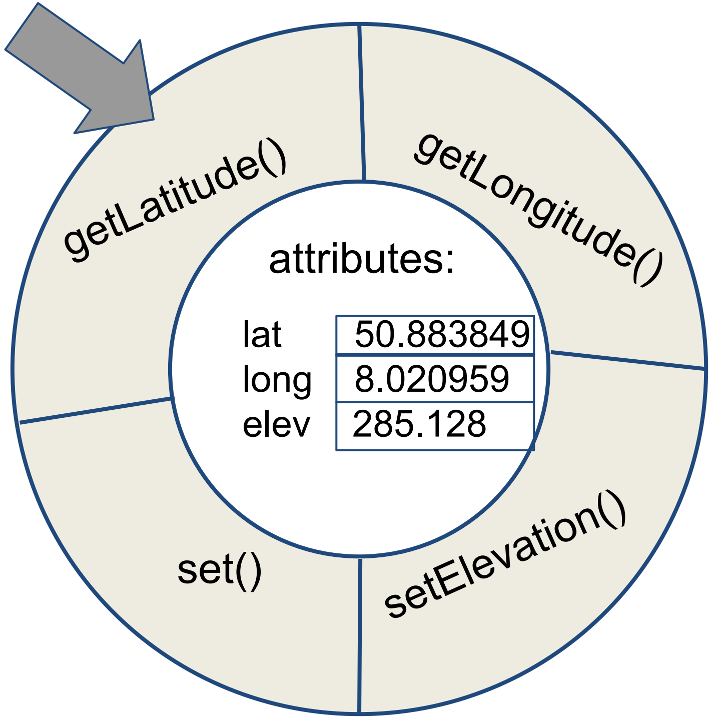
```
GPSCoord here; // GPS coordinate object

// we can modify latitude and
// longitude only together:
here.set(50.883849, 8.020959);

// we can modify the elevation:
here.setElevation(285.128);
// we can retrieve these attributes:
double lat = here.getLatitude();
double lng = here.getLongitude();
// .. but not elevation
```



6.1. Classes and Objects

Declaring a class GPSCoord:

```
// GPS coordinate class:
class GPSCoord {
 private:
  // latitude, longitude, elevation:
  double lat, long, elev;
 public:
  // set latitude and longitude:
  void   set(double la, double lo);
  void   setElevation(double val);
  double getLatitude();
  double getLongitude();
};
```



6.1. Classes and Objects

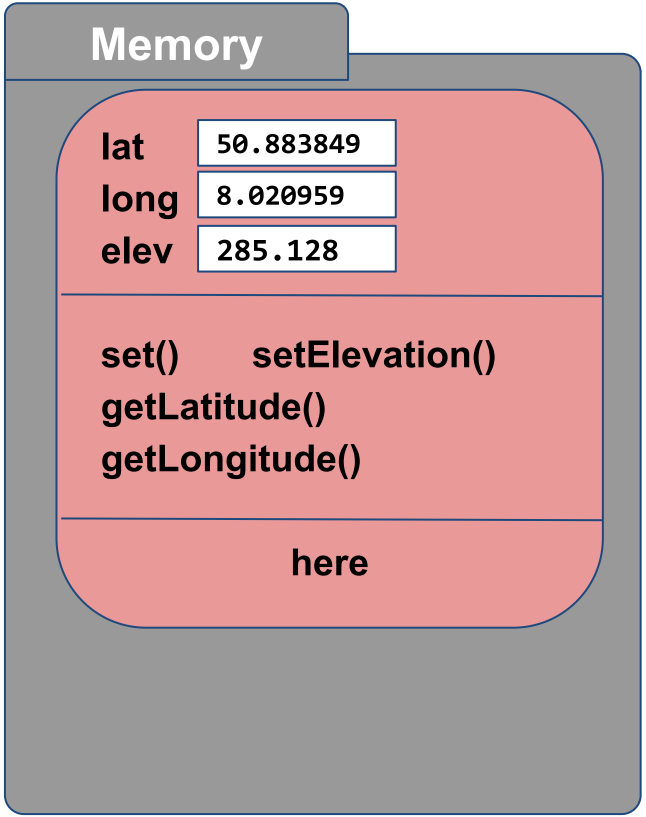
Implementing the methods for the class GPSCoord:

```
void GPSCoord::set(double la, double lo){
  lat = la; long = lo;
}
void GPSCoord::setElevation(double val){
  elev = val;
}
double GPSCoord::getElevation(){
  return elev;
}
double GPSCoord::getLatitude(){
  return lat;
}
```

6.1. Classes and Objects

Creating an object of the class GPSCoord:

```
GPSCoord here; // GPS coordinate object

// we can modify latitude and
// longitude only together:
here.set(50.883849, 8.020959);

// we can modify the elevation:
here.setElevation(285.128);
// we can retrieve these attributes:
double lat = here.getLatitude();
double lng = here.getLongitude();
// .. but not elevation
```

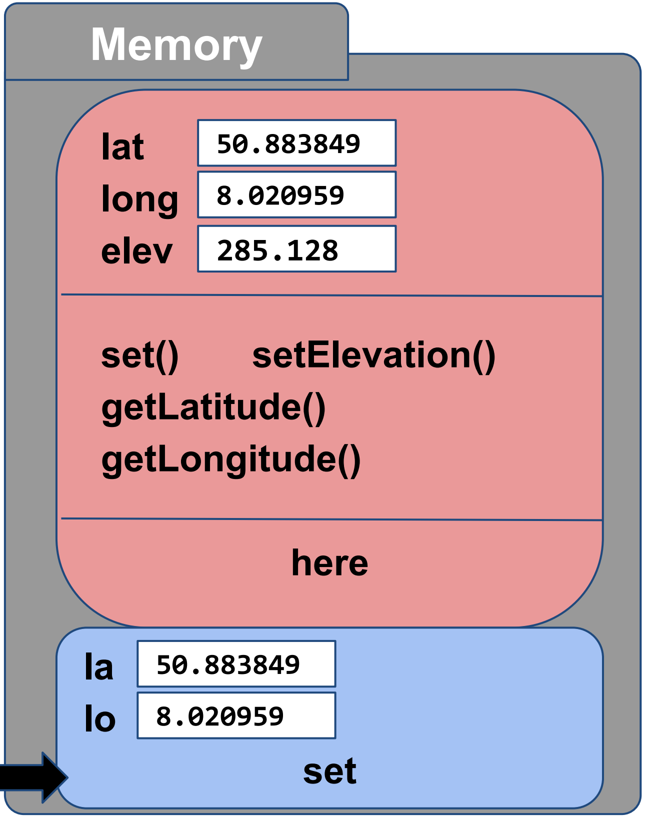


6.1. Classes and Objects

An object's method (member function) has access to *all* its attributes (variables) and methods

```
void GPSCoord::set(double la, double lo){
  lat = la; long = lo;  // defaults
  // call GPSCoord methods to update:
  lat  = getLatitude();
  long = getLongitude();
  setElevation(285.128);
}
```

```
GPSCoord here; // GPS coordinate object
// set latitude and longitude:
here.set(50.883849, 8.020959);
```



6.1. Classes and Objects

❗ Note new suggested indentation & syntax for classes. One space should come before private or public, and a colon (':') after these keywords. Example:

```
// GPS coordinate class:
class GPSCoord {
◇private:
◇◇// latitude, longitude, elevation:
◇◇double lat, long, elev;
◇public:
◇◇// set latitude and longitude:
◇◇void set(double la, double lo);
◇◇void etElevation(double val);
◇◇double getLatitude();
◇◇double getLongitude();
};  // mind the semicolon after the declaration
```

6.1. Classes and Objects: Minor notes

Attributes could also be public (but usually aren't)
Methods can also be implemented in the class declaration

```
class Test {
 private:
  int attribute1;  // a private attribute
 public:
  bool attribute2;  // a public attribute
  void method1(int parameter) { attribute1 = parameter; };   // methods implemented
  void method2() { std::cout << attribute1 << std::endl; };  // in class declaration
};

int main(){
  Test myTest;  // create an object of class Test
  myTest.attribute2 = false;  // we can access public attributes
  myTest.method1(21); // we can call public methods
  return 0;
}
```

6.1. Classes and Objects: Minor notes

The class declaration is usually in the file **`className.h`**, the class' method implementations in the file **`className.cpp`**

Test.h

```
class Test {
 private:
  int attribute1;
 public:
  bool attribute2;
  void method1(int parameter);
  void method2();
};
```

Test.cpp

```
#include "Test.h"
void Test::method1(int parameter) {
  attribute1 = parameter;
};
void Test::method2() {
  std::cout << attribute1 << std::endl;
};
```

mainTest.cpp

```
#include "Test.h"
int main(){
  Test myTest;
  myTest.attribute2 = false;
  myTest.method1(21);
  return 0;
}
```

6.2. Constructors and Destructors

Reminder: Variables can be declared and later initialized, or they can be immediately initialized within the declaration:

```
char mySymbol = '?';
```

This declaration and initialization can be done for classes' objects as well:

```
GPSCoord myLocation(50.88385, 8.02096, 285.128);  // coordinates of Siegen
Employee user("Carnegie", 62413, 12, 4729.12);  // employee Carnegie
SizedSymbol bigQuestion('?', 14);  // a SizedSymbol '?' with size 14
```

This requires a special method with the class' name: The constructor

6.2. Constructors and Destructors

A constructor has no return value (not even void) and is automatically called

```
class Test {
 private:
  int attribute1;
 public:
  Test(int parameter) {  // this is a constructor for class Test
    attribute1 = parameter;
  };
  void method2() {
    std::cout << attribute1 << std::endl;
  };
};

int main(){
  Test myTest(21);    // object's constructor initializes is automatically called
  myTest.method2();   // this will print out '21' to the terminal
  return 0;
}
```

6.2. Constructors and Destructors

Constructors can be overloaded (distinguished by their parameters):

```
class Test {
 private:
  int attribute1, attribute2;
 public:
  // multiple constructors for class Test:
  Test() { attribute1 = 0; }; // this is a default constructor
  Test(int parameter) { attribute1 = parameter; };
  Test(int parameter1, int parameter2) { attribute1 = parameter2; };
  void method2() { std::cout << attribute1 << std::endl; };
};

int main(){
  Test myTest(4, 12);  // object of class Test has attribute1 initialized to 12
  myTest.method2();  // this will print out '12' to the terminal
  return 0;
}
```

6.2. Constructors and Destructors

- A class' default constructor is a constructor without parameters
- A class without declared constructors results in the compiler automatically generating a default constructor
- A default constructor is invoked when an object is created, but not initialized

```
class Test {
 private:
  int attribute1;
 public:
  // Test() {}  --> an automatically generated constructor would look like this
  void method2() { std::cout << attribute1 << std::endl; };
};

int main(){
  Test myTest;  // this object is initialized with empty default constructor
  return 0;
}
```

6.2. Constructors and Destructors

A destructor is automatically called whenever an object is destroyed:

```
bool myFunction(){
  Test myTest(17);  // this object is created and initialized here
  return false;     // myTest's destructor is called when function returns
}
```

This destructor is another special method with the same name as the class, starting with a ~:

```
Test::~Test() {  // this is the destructor for class Test
  // here come statements for application-specific clean up
}
```

6.2. Constructors and Destructors

Example 00 (difficulty level: 🌶️)

```
/**
  Write a program that declares, implements, and uses a class with no attributes.
  The class should print 'hello' to the terminal when its object is created and
  'bye' when its object is removed from memory.
*/
#include <iostream>     // terminal input and output classes and objects

// write the class here

int main() {
  // create a class object here

  return 0;
}
```

6.2. Constructors and Destructors

Example 01 (difficulty level: 🌶️🌶️)

```
/**
  Write a program that declares, implements, and uses a class with two attributes,
  a boolean called 'flag' and an integer called 'number', which can only be changed
  or read through a constructor. The class should also have a method 'get' with no
  parameters, which returns the integer 'number' only if 'flag' is true, and
  otherwise 0.
*/
// write the class here
int main() {
  int returnValue;
  // create a class object here
  // and use its get method
  return returnValue;
}
```

6.2. Constructors and Destructors: Maze Game v.4.00

Change the module into a class Maze, with **`mazeGame.cpp`** looking this way:

mazeGame.cpp

```
/* Fourth draft of Maze Game: drawing is in class "Maze" */
#include "Maze.h"  // everything related to the maze
int main() {
  auto c = ' ';         // used for user key input
  Maze maze(10, 5);     // initialize the maze and put player at (10, 5)
  while ( c != 'q' ) {  // as long as the user doesn't press q ..
    maze.draw('@', 3);     // draw player as a '@' with color pair 3
    c = getch();           // capture the user's pressed key
    switch (c) {
      case 'w': maze.up(); break;  // go up
      case 's': maze.down(); break;  // go down
      case 'a': maze.left(); break;  // go left
      case 'd': maze.right(); break;  // go right
    }
  }
  return 0;
}
```

6.3. **`this`** and initializing **`const`** attributes

Class methods often reuse attribute names, leading to a problem:

```
class Maze {
 public:
  Maze(int16_t x, int16_t y);
  ...
 private:
  int16_t x, y;
  ...
};
```

```
Maze::Maze(int16_t x, int16_t y) {
 // how to assign the values of
 // constructor parameters x and y
 // to class attributes x and y?
   ...
}
Maze::Maze() {
 int16_t x, int16_t y;
 // how to assign the values of
 // constructor parameters x and y
 // to local variables x and y?
   ...
}
```

6.3. **this** and initializing **const** attributes

One solution that works in all the class' methods is to use **this**

```
Maze::Maze(int16_t x, int16_t y) {
   this->x = x;  // attribute x = value of constructor parameter x
   this->y = y;  // attribute y = value of constructor parameter y
   ...
}
```

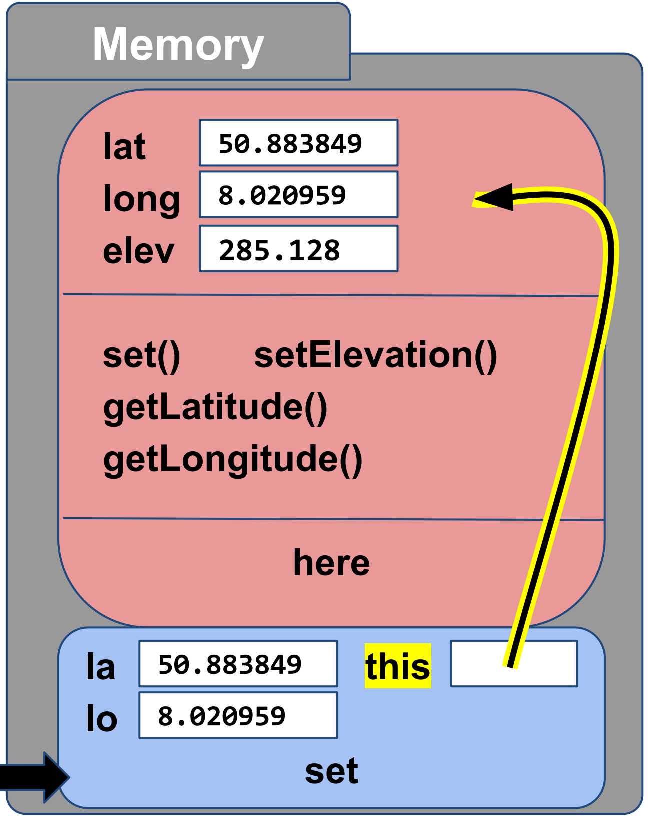
- Note: Each object gets its own copy of data members, all objects share a single copy of the class' methods
- **this** is an implicit pointer (see later) that is passed as a hidden parameter for all the class' methods, and is there available as another local variable

6.3. **`this`** and initializing **`const`** attributes

`this` is passed as a hidden parameter for all the class' methods, as an extra (pointer) variable

```
void GPSCoord::set(double lat, double long){
   this->lat = lat;
   this->long = long;
   this->lat  = getLatitude();
   this->long = getLongitude();
   setElevation(285.128);
}
```

```
GPSCoord here; // GPS coordinate object
// set latitude and longitude:
here.set(50.883849, 8.020959);
```



6.3. **`this`** and initializing **`const`** attributes

Another (shorter) solution for constructors is to use this initialization syntax:

```
Maze::Maze(int16_t x, int16_t y) : x(x), y(y) {
  // attributes x and y have now the same value as constructor
  // parameters x and y
}
```

This *member initializer list* syntax in constructors is not an assignment but a real initialization of the attribute. Curly braces can also be used:

```
Maze::Maze(int16_t x, int16_t y) : x{x}, y{y} {
  // attributes x and y have now the same value as constructor
  // parameters x and y
}
```

6.3. **this** and initializing **const** attributes

This syntax addresses the problem of initializing a class' **const** attribute:

```
class Maze {
  ...
 private:
  const int16_t mazeXlen;
  const int16_t mazeYlen;
  ...
};
```

```
Maze::Maze(int16_t x, int16_t y) {
  // we cannot assign to const:
  mazeXlen = 15; // compiler error
  mazeYlen = 10; // compiler error
  ...
}
```

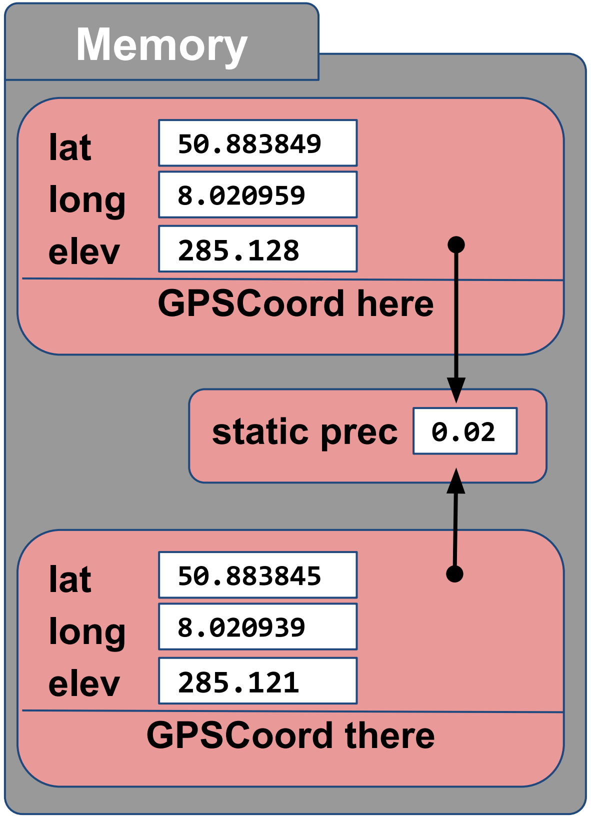
This syntax is not an assignment but an initialization, and thus works:

```
Maze::Maze(int16_t x, int16_t y) : mazeXlen(15), mazeYlen(10) {
  // mazeXlen is now 15, mazeYlen 10
  ...
}
```

6.4. **`static`** members

- Each object has its own class attributes
- All objects share one copy of the class' methods
- **`static`** attributes, or [static members](), are stored once across all objects. *Changing the through one object will change it for all objects.*

```
GPSCoord here;
GPSCoord there;

there.prec = 0.02;
// the following will print out 0.02:
std::cout << here.prec;
```



6.4. `static` members

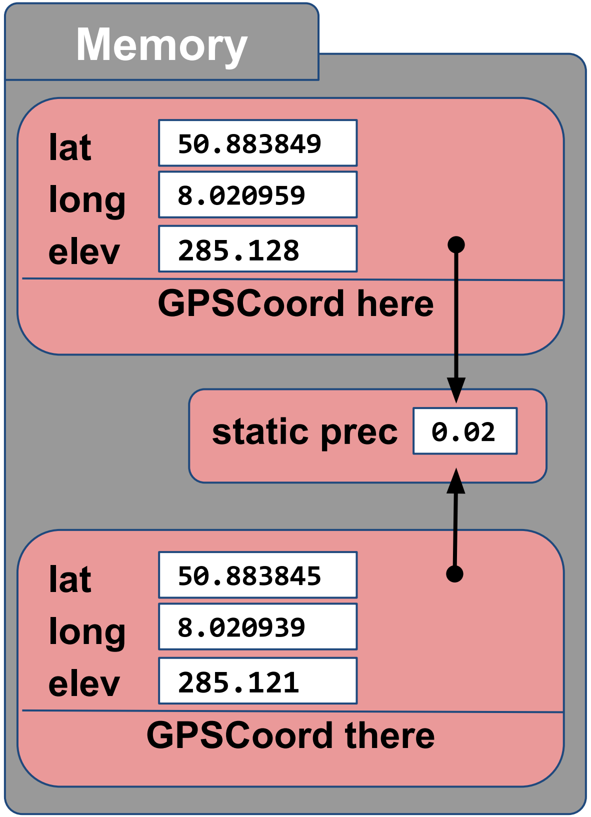
- They exist even if no objects of the class have been defined
- Static class members are declared in the class declaration, and are defined in the implementation / source file:

GPSCoord.h

```
class GPSCoord {
 public:
  // declaring a precision
  // attribute for all
  // GPSCoord objects:
  static double prec;
 ...
};
```

GPSCoord.cpp

```
// precision definition:
double GPSCoord::prec;

 ...
```

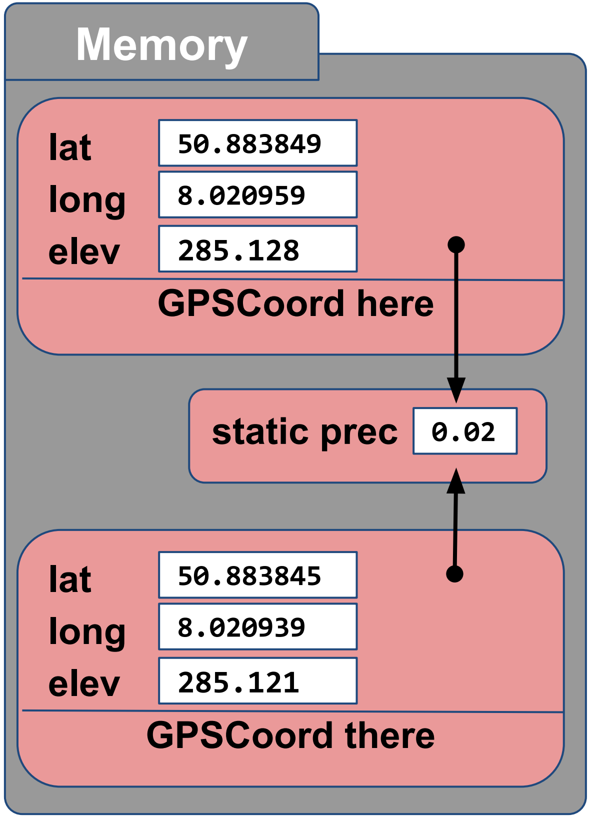


6.4. `static` members

This is not the same as **`static`** variables:

- Local variables that are declared as **`static`** are stored as global variables in the static data segment in memory
- Their size has to be known at compile time (e.g., for arrays)

```
void GPSCoord::set(){
   // this variable will stay in memory and keep
   // and keep its value after the method finishes:
   static double prec = 0.01;
   setElevation(285.128, prec);
}
```



6.5. The **string** Class

iostream has a class called [**std::string**](), helping in dealing with strings:

```
#include <iostream>      // terminal input and output classes and objects
int main() {
  std::string myFirstName("John");  // initialize with constructor
  std::string myLastName = "Doe";  // initialize with assignment
  std::string myString = myFirstName + myLastName;  // concatenation

  std::cout << myString << ", length = " ;
  std::cout << myString.length() << std::endl;  // returns myString length

  std::cout << " Do found at = ";
  std::cout << myString.find("Do") << std::endl;  // return position of "Do"

  std::cout << myString.compare(4, 3, "Do");  // is substring at position 4
  std::cout << std::endl;                    // and length 3 the same as "Do"?
  return 0;
}
```

6.5. The **string** Class

iostream has a class called [**std::string**](), helping in dealing with strings.
Several **std::string** alternatives use instead of **char**:

[**std::wstring**]()	uses **wchar_t** for [wide strings]()
std::u8string (since C++20)	uses **char8_t**
std::u16string (since C++11)	uses **char16_t**
std::u32string (since C++11)	uses **char32_t**

```
#include <iostream>     // terminal input and output classes and objects
int main() {
  std::wcout.imbue(std::locale("en_US.UTF-8"));
  std::string  myString  = "¡Hola! 日本 שלום 你好 مرحبا";  // UTF-8
  std::wstring mywString = L"¡Hola! 日本 שלום 你好 مرحبا";  // wide chars
  std::cout  << myString << sizeof(myString[0]) << std::endl;
  std::wcout << mywString << sizeof(mywString[0]) << std::endl; // ! note wcout
  return 0;
}
```

6.6. The **ifstream** and **ofstream** Classes

The module **fstream** contains a class [**std::ifstream**] for reading from a file:

fileTest.cpp

```
#include <iostream>    // terminal input/output classes & objects
#include <fstream>     // input file stream class std::ifstream

int main() {
  std::ifstream myFile("fileTest.cpp");  // initialize with constructor

  char c;
  while (myFile.get(c)) {   // get a character from the file, move to next
    std::cout << c;         // and output it to the terminal
  }

  return 0;
}
```

6.6. The **`ifstream`** and **`ofstream`** Classes

The module **`fstream`** also contains a class [**`std::ofstream`**]() for writing to a file:

copyTest.cpp

```
#include <fstream> // in/output file stream classes

int main() {
  std::ifstream myFile("copyTest.cpp");    // initialize input and output file
  std::ofstream myFileCopy("copyTest_copy.cpp");  // streams with constructors

  char c;
  while (myFile.get(c)) {   // get a character from the input file stream
    myFileCopy << c;        // and output it to the output file stream
  }

  return 0;
}
```

6.7. The **`tuple`** Class

A **`std::tuple`** is a fixed-sized collection values of various data types (preview of what is still to come, as it uses templates under the hood):

```
#include <iostream> // std::cout, std::endl, and tuples functionality
int main() {
  auto myUser = std::make_tuple("James", "Smith", 187.2);  // auto = std::tuple
  // get with index-based access:
  std::cout << std::get<0>(myUser) << " " << std::get<1>(myUser);
  // get with type-based access:
  std::cout << ":" << std::get<double>(myUser) << std::endl;
  return 0;
}
```

- `get<0>(myUser)` accesses first element (hence index-based access, since C++11)
- `get<double>(myUser)` accesses the double (hence type-based access, since C++14) (works only if 1 tuple element has this type, otherwise the compiler reports an error)

6.7. The **`tuple`** Class

With decomposition declarations or [structured bindings](#) (since C++17), you can unpack the contents of the tuple into individual variables:

```
#include <iostream> // std::cout, std::endl, and tuples functionality
int main() {
  auto myUser = std::make_tuple("James", "Smith", 187.2);  // auto = std::tuple
  auto [fname, lname, height] = myUser;  // decomposition declaration, C++17
  std::cout << fname << " " << lname << ":" << height << std::endl;
  return 0;
}
```

note that the first `auto` above can deduce myUser as an **std::tuple** object