# Part VII

# Object-Oriented Programming

# Outline

# Object-Oriented Programming

- Object-Oriented Programming is a vital part of most complex programs these days
- Although not absolutely necessary for scientific programming, it will make for far more readable code and make code easier to read and debug
- The essential concepts in OOP are encapsulation and data-hiding.
- There are further concepts such as inheritance and polymorphism
- This is not a course in OOP, so only a brief introduction to the concepts will be given
- Much of the skill in using OOP is knowing how to partition concepts into objects/classes; this will not be covered here.

# Classes

- Simple objects were available in C as a `struct`. (short for structure)
- A `struct` is a self-contained package of other types.
- The C++ generalized form of these is a class.
- Provides names for its member-types
- Allows for easy copying/assignment of these objects
- For example, a `Date` class requires three members:

```
class Date{
public:
  int day;
  int month;
  int year;
};
```

- Within C++ a class is a new type, with all the type-safety features that implies.

# Using classes

```
void printDate(const Date& d){
  std::cout << d.day << "/" << d.month << "/" << d.year <<
    std::endl;
}

int main(void){
  Date myDeadline;
  myDeadline.day = 31;
  myDeadline.month = 8;
  myDeadline.year = 2021;

  printDate(myDeadline);
}
```

- Copying and assignment of classes is implemented automatically for simple data.
- Anything else, e.g. printing, checking for equality, arithmetic operations, etc. must be explicitly coded by the programmer.
- A specific variable of type `Date` is referred to as an *instance* of the `Date` class.

# Outline

# Member functions

We may want to be able to advance a `Date` by a single day:
Approach 1:

```
Date advance(const Date& d){
  Date newD = d;
  newD.day += 1;
  if( newD.day > numDaysInMonth[newD.month] ){
    newD.day = 1;
    newD.month += 1;
  }
  if( newD.month > 12 ){
    newD.month = 1;
    newD.year += 1;
  }
  return newD;
}

Date tomorrow = advance(today);
```

However, this approach will result in separate functions not obviously related to a `Date`.

# Member functions

- It might be better if we could do something like:

```
myDate.advance();
```

  which would advance `myDate` by one day.
- We can do this as follows:

```
class Date{
public:
  void advance();
};
void Date::advance(){
  day += 1;
}
```

  where the `advance` function knows about the current object that it is being called on.
- The `advance` function is a *member* of the `Date` class.
- The `day` referred to is specific to this object.
- Calling `advance()` on one `Date` object will not affect any other `Date` object

# Access control

- As we've written it so far, data members of a Date instance are *public*.
- i.e. any external function can access the members:

```
d.day = 35;
d.month = 13;
```

  providing invalid data that will cause problems later.
- It would be useful if all access to the object's members had to go through a `Date` member function.

# Private

- We can do this by making some of the members private:

```
class Date{
public:
  void advance();
private:
  int day;
  int month;
  int year
};

int main(void){
  Date d;
  d.day = 1; //  Compile-time error - day is private
  d.advance(); // Allowed since advance() is public
}
```

- In fact, class members are private by default, hence the "public" in previous slides.
- Members of a struct are public by default.

# Access functions

- However, we now cannot get any data into the `Date` object in the first place.
- We can add simple access functions:

```
class Date{
public:
  int getDay()const;
  void setDay(int);
};
int Date::getDay()const{
  return day;
}
void Date::setDay(int d){
  day = d;
}
```

- The advantage is that all access to `day` goes through one function, and can be checked for errors at this point, without having to remember to introduce checks elsewhere.

# Private member functions

- We may also wish to have private member functions:

```
class Date{
private:
  void checkCorrect()const;
};
```

- which could be called from within any other member function:

```
void Date::setDay(int d){
  day = d;
  checkCorrect();
}
```

- and the `checkCorrect` function ensures that the date stored is a valid one.

# Naming conventions

- Note that member functions cannot have the same names as member data.
- An appropriate naming convention should be used. Note that it is better for the interface to the class (its member functions) to have memorable names than the internal data.

```
class Date{
public:
  int day()const;
private:
  int m_day;
};
```

# Const-ness

- You may have noticed the `const` on the `getDay` function above.
- This indicates that the function does not change any member data of the `Date` object (except for any static members)
- Any attempt to so within the function is a compile-time error:

```
int Date::getDay()const{
  m_month = 1; // Compile error
  return m_day;
}
```

# Const-ness ctd

Also, if a `Date` object has been declared to be constant, then you cannot call non-const functions on it:

```
const Date myBirthday = d; // Where d holds a pre-defined Date
// The following is OK:
std::cout << myBirthday.getDay() << "/" <<
    myBirthday.getMonth() <<
std::endl;

 // Compile-error: setDay does not keep a Date object constant
myBirthday.setDay(12);
```

# Outline

# Constructors

Having to write:

```
Date d;
d.setDay(25);
d.setMonth(12);
d.setYear(2020);
```

is labourious. It would be easier if we could write:

```
Date d(25, 12, 2020);
```

The function that does this is called a constructor.

# Constructors

- To define a constructor:

```
class Date{
public:
  Date(int, int, int);
  // More function prototypes
};

Date::Date(int d, int m, int y){
  m_day = d;
  m_month = m;
  m_year = y;
}
```

- A constructor is a function with the same name as the object it refers to, and with no return type (not even `void`).
- A class may have multiple constructors taking different parameters, or even default parameters.

# Constructor initializer list

- You can also initialize data members outside the constructor function
- This uses an initializer list

```
Date::Date(int d, int m, int y)
      : m_day(d), m_month(m), m_year(y){
}
```

- This is the only way to initialize `const` members of a class, as they cannot be modified once a class instance has been constructed.
- (A `const` member could be used for a run-time-sized `Array` but whose size cannot be changed later.)

# Default constructor

- As soon as you create a constructor of your own, the compiler no longer automatically generates the default constructor.
- A default constructor is one that either takes no parameters, or all its parameters have default values (so it can be called with no parameters).
- An empty function body may be appropriate (but may leave data members uninitialized), or you may choose to explicitly initialize member data with default or nonsense values:

```
Date::Date(){
  day = 32;
  month = 13;
  year = 0;
}
```

- You can also disable the default constructor, to force explicit initialization of all `Dates` by:

```
class Date{ Date() = delete; };
```

# Destructors

- If you allocate memory in your constructor, using **new**, you should also **delete** it when the object is destroyed.
- This is done in the destructor:

```
class MyArray{
public:
  MyArray(int);
  ~MyArray();
private:
  double* data;
};
MyArray::MyArray(int n){
  data = new double[n];
}
MyArray::~MyArray(){
  delete[] data;
}
```

# Destruction

- The `data` of `arr` is deleted when `arr` goes out of scope, i.e. at the end of the function or block in which it is defined.

```
int f(){
  MyArray a(5);
  for(int i=0 ; i < 10 ; i++){
    MyArray b(10);
    // b.~MyArray is called at the end of every iteration
  }
  // a.~MyArray is called directly after this line
}
```

- The destructor takes no parameters.
- (The choice for destructor syntax comes from bitwise NOT.)

# Classes and header files

- In order to compile code using a class, the compiler needs to know the data it contains and its member functions.
- Therefore, `Date.H` should contain:

```
class Date{
public:
  int day();
private:
  int m_day;
};
```

- Then, any `.C` file that uses the `Date` class and has `#include "Date.H"` will compile correctly.
- A separate file `Date.C` should contain the member function definitions:

```
int Date::day(){ return m_day; }
```

- Compiling these and linking them as in the previous lecture will give a complete program.

# Heap construction/destruction of classes

- To allocate space on the heap for objects of a specific class:

```
MyArray* a = new MyArray(10);
```

- which allocates a single instance of `MyArray` and calls its constructor with an argument 10.
- This object then persists until the destructor is explicitly called using

```
delete a;
```

- To allocate an array of these, use:

```
MyArray* a = new MyArray[10];
delete[] a;
```

- This uses the default constructor to initialize the instances. A default constructor must be available; if not it is a compile-time error.

# Class pointer function call

- In order to call a member function of a class, given a pointer to an instance of that class, use:

```
Car* myCar = new Car;
myCar->setNumberPassengers(10);
```

- This is essentially identical to:

```
Car* myCar = new Car;
(*myCar).setNumberPassengers(10);
```

- (It would be different only if some of these operators were overloaded oddly.)

# Return by reference

- Suppose we have a large object stored within a class and need access to it from outside:

```
BigObject MyClass::data(){
  return myData;
}
```

- This may suggest bad design already; direct access to a class's internal data is usually not sensible.
- However, it is permissible to return a reference to an object from a function to avoid the expense of copying:

```
BigObject& MyClass::data(){
  return myData;
}
```

# Return by reference ctd

- The lack of copy is only preserved until the object is allocated to another variable:

```
MyClass myObj;
BigObject a = myObj.data(); // Invokes a copy
myObj.data().bigObjFunc(); // Does not invoke a copy.
```

- Note that we have now exposed the contents of `myData` to outside its class - possibly bad!

# Return by reference ctd

- In order to prevent alteration, we should of course return a constant reference:

```
const BigObject& MyClass::data(){
  return myData;
}
```

or even

```
const BigObject& MyClass::data()const{
  return myData;
}
```

which would prevent the call to `bigObjFunc()` above if it were not a `const` member function.

# Outline

# Copy-constructor

- The copy-constructor is used in the case:

```
MyClass a;
MyClass b = a;
```

  i.e. `b` is constructed by copying `a`.
  (It is also used when passing an object of type `MyClass` to a function.)

- It is declared as:

```
class MyClass{
  MyClass(const MyClass& c);
};
MyClass::MyClass(const MyClass& c){
  // Initialize all members as necessary from c
  myInt = c.myInt;
}
```

- Note that a class has access to all private members of any object of the same type.

# Default copy constructor

- If a copy-constructor is not defined, then a default version is created that copies the object member-by-member, i.e. all members are copied using their own copy constructors.
- This is often the required behaviour, unless the object contains heap-allocated pointers that would be freed on destruction of an object.
- In this case, the copy-constructor needs to allocate more memory and copy the data pointed to.

# Copy-assignment

A slightly different version of the above occurs when objects are copied by assignment:

```
MyClass a(x, y, z);
MyClass b(u, v, w);

b = a;
```

Here, the appropriate member function definition is given as:

```
class MyClass{
  MyClass& operator=(const MyClass& c){
    // Copy data from c as appropriate
    return *this;
  }
};
```

- Once again, the default is to copy-assign each member by itself, but this may need to be altered in the case of heap-allocation.
- The `this` pointer is a pointer to the current object. It is not usually necessary to use it.

# Class summary

- Member functions are typically
  - actions - do something to this object
  - read/set functions - get/set information in this object
- Member data should usually be private, to avoid unregulated access
- Member functions are typically public - for access from other objects/functions
- although some may be private - for use internally
- If you just want a collection of data, then use a `struct`
- A `struct` is identical to a `class` with all its members `public` by default