Class I

- generalization of a C structure
- recap: in a structure we can have variables and function pointers as members
- in a class we can have variables and functions as members
- a class usually (protected members introduced later) has two kinds of members:
  - public members
  - private members
Class II

- general structure of a class:

```cpp
class class-name {
private:
    /* variable declarations, if any */
    /* function-declarations, if any */
public:
    /* variable declarations, if any */
    /* function-declarations, if any */
};
```

- some people prefer to declare the `public` members first, its a matter of taste
Class III

- in general think out about a class as a package of “stuff”:
  - it contains some private data members
  - these private data members could be accessed by public “getter” functions
  - these private data members may be modified by public “setter” functions
  - some computation could be done by calling the public functions which (possibly) use the private data members
  - how this computation is done internally is completely hidden from the user
Class IV

- an often used analogy of a class: the class of cars
  - the private data members are e.g. pistons, radiator, temperature controller, gear box etc.
  - the public “getter” functions are e.g. speedometer etc.
  - the public “setter” functions are e.g. the sterring wheel, the gear-stick etc.
  - how moving the gear-stick in a certain direction causes the actual gear shift is not revealed to the users of the cars
Class V

- constructors:
  - we create an instance of a class, i.e., an object by calling a constructor (of the relevant class)
  - constructors are functions having the name: class-name

- destructors:
  - we destroy an instance of a class, i.e., an object by calling a destructor (of the relevant class)
  - constructors are functions having the name: ~class-name
Class Declaration I

- this goes into a .H or .hpp file, called the header file:

```cpp
class MonteCarloSpecs {
private:
  int n_iters;
  float time_in_secs;
  float prop_burn_in;
  double *log_density;

public:
  MonteCarloSpecs (int n_iters,
                   float time_in_secs,
                   float prop_burn_in = 0.5);
  ~MonteCarloSpecs (void);
  int get_n_iters (void) const;
  void set_n_iters (int n_iters);
  float get_time_in_secs (void) const;
  void set_time_in_secs (float time_in_secs);
  float get_prop_burn_in (void) const;
  void set_prop_burn_in (float prop_burn_in);
  void print (void) const;
};
```
Class Declaration II

- note the `const` keywords: they essentially make sure that the corresponding functions do not get to modify the "this" object and its members

- the `float prop_burn_in = 0.5` in the constructor function `MonteCarloSpecs()` is a way of setting default values to function arguments, this strategy holds for any functions, not only for constructors

- note the default values need only be specified in the declaration and not in the definition of the function
Class Declaration III

- while writing the header file, i.e., the .h or the .hpp files remind yourself of the following recommendations and requirements:
  - say the file is called foo.h
  - the first non-commented line in the file should be:
    ```
    #ifndef FOO_H
    #define FOO_H
    ```
  - the last non-commented line in the file should be
    ```
    #endif // end of FOO_H
    ```
  - every header file should be self-contained, i.e., it should include all the header files it needs, so assuming our foo.h only requires some standard library stuff and nothing really fancy, we should have the following before any code:
    ```
    #include <iostream>
    ```
  - this will include the standard header
Class Implementation I

• this goes into a .c or .cpp file, called the implementation file:

• constructor:

```cpp
MonteCarloSpecs::MonteCarloSpecs (int n_iters, 
    float time_in_secs, 
    float prop_burn_in)
    : n_iters(n_iters),
      time_in_secs(time_in_secs),
      prop_burn_in(prop_burn_in)
{
    cout << "Creating a MonteCarloSpecs object with n_iters = " 
    << n_iters << endl;
    log_density = new double[n_iters];
    for (int ii = 0; ii < n_iters; ++ii)
        log_density[ii] = 0.0;
}
```
Class Implementation II

- things to note:
  - "::" is the scope operator
  - ":" is the marker for the start of the *initializer list*
  - the names for the arguments of the constructor
    
    MonteCarloSpecs::MonteCarloSpecs( ) function are same as that of the class private data members, remember function arguments names don’t matter, I chose the same names for lack of imagination and because I re-cycle names as much as possible
  
    note the compact *initializer list* in the constructor, a very efficient way to implement constructors (although, this should be used for simple assignments)
  
    there are other ways of doing the same thing, e.g. using the “this->foo = foo;” statements but its not advised and hence not explained here, coming shortly in a different context
Class Implementation III

• things to note:
  – difference with C: the `new` operator does the job of the `malloc( )` function (will see more of `new` later), it returns a pointer
  – difference with C: a variable (int ii;) could be declared anywhere, it may not be the very first line of the scope

• things to remember while writing the .C or the .cpp file:
  – say the file is called `foo.C`
  – it should have `#include "foo.H", if it exists, among other `#include` statements
  – it should have the following as the first few non-commented lines:
    ```
    using std::cout;
    using std::endl;
    ```
    will see why we need the above very shortly
Class Implementation IV

- destructor:

  ```cpp
  MonteCarloSpecs::~MonteCarloSpecs (void)
  {
      cout << "Destroying a MonteCarloSpecs object with n_iters = "
      << n_iters << endl;
      delete [] log_density;
  }
  ```

- destructor doesn’t take any arguments and doesn’t return anything

- the destructor essentially does the clean up i.e. frees up any memory consumed by the object whose destructor was called

- difference with C: the `delete []` operator does the job of the `free( )` function

- there is also an operator `delete` which works on individual variables as opposed to arrays

- it should only be used for variables created with the `new` operator
Class Implementation V

• the print function:

```cpp
void MonteCarloSpecs::print (void) const
{
    cout << "This Monte Carlo object:" << endl
         << "n_iters: " << get_n_iters() << endl
         << "time_in_secs: " << get_time_in_secs() << endl
         << "prop_burn_in: " << get_prop_burn_in() << endl
         << "log_density:" << endl;

    int ii;
    for (ii = 0; ii < n_iters - 1; ++ii)
        cout << log_density[ii] << "", ";
    cout << log_density[ii] << endl;
}
```

• difference with C: the `cout <<` is like the `printf()` function but here we don’t need “%d”, “%f” etc.

• difference with C: the `endl` symbol is like \n but `endl` flushes the output after printing \n
Class Implementation VI

- setter functions: they could be used to do error-check on the arguments

```c
void MonteCarloSpecs::set_n_iters (int n_iters)
{
    assert(n_iters > 0);
    this->n_iters = n_iters;
    delete [] log_density;
    log_density = new double[n_iters];
    for (int ii = 0; ii < n_iters; ++ii)
        log_density[ii] = 0.0;
}

void MonteCarloSpecs::set_time_in_secs (float time_in_secs)
{
    assert(time_in_secs > 0.0);
    this->time_in_secs = time_in_secs;
}

void MonteCarloSpecs::set_prop_burn_in (float prop_burn_in)
{
    assert(0.0 < prop_burn_in && prop_burn_in < 1.0);
    this->prop_burn_in = prop_burn_in;
}
```
Class Implementation VII

- things to note:
  - the `this` is a keyword, it is a `const` pointer to the object which is an instance of the class under consideration
  - note here `this` is needed to differentiate between e.g. `time_in_secs` as the argument of `MonteCarloSpecs::set_time_in_secs()` and the private member the class `MonteCarloSpecs` of the same name
  - in the following implementation of the `MonteCarloSpecs::set_time_in_secs()` function, this won’t be needed because the above differentiation is not needed anymore:

```cpp
void MonteCarloSpecs::set_time_in_secs (float timeInSecs)
{
    assert(timeInSecs > 0.0);
    time_in_secs = timeInSecs;
}
```
Class Implementation VIII

- now we could use the setters to make a smarter constructor which does error-check on the arguments:

```cpp
MonteCarloSpecs::MonteCarloSpecs (int n_iters,
    float time_in_secs,
    float prop_burn_in)
{
    cout << "Creating a MonteCarloSpecs object with n_iters = "
    << n_iters << endl;
    log_density = NULL;
    set_n_iters (n_iters);
    set_time_in_secs (time_in_secs);
    set_prop_burn_in (prop_burn_in);
}
```
Class Implementation IX

- the getter functions are simple e.g.:
  ```
  int MonteCarloSpecs::get_n_iters (void) const
  {
      return n_iters;
  }
  ```

- similarly, one can write `get_time_in_secs` and `get_prop_burn_in`

- note the difference between a const member function such as all the above getter functions and non-const member functions such as all the setter functions above is the following:
  - within const member functions the this pointer is represented as: type MonteCarloSpecs const *const this
  
  - within non-const member functions the this pointer is represented as: MonteCarloSpecs *const this
Class Use I

- note the usage of default values in the constructor call:

```c
int main (int argc, char **argv)
{
    MonteCarloSpecs mcs1(10, 10, 0.1);
    MonteCarloSpecs mcs2(5, 5);
    MonteCarloSpecs *mcs3 = new MonteCarloSpecs(1, 1, 0.01);

    mcs1.print( );
    mcs1.set_n_iters(20);
    mcs1.set_time_in_secs(20);
    mcs1.set_prop_burn_in(0.2);
    mcs1.print( );
    mcs2.print( );
    mcs3->print( );
    test( );
    delete mcs3;
    return 0;
}
```
Class Use II

- note the . and the \( \rightarrow \) operators are used much like in C while dealing with structure and pointer to structures

- note the destructor function \(~\text{MonteCarloSpecs}( )\) is not called manually, its done at run time automatically when the object is no longer needed (e.g. it goes out of scope, look at function \text{test}(\ )\) in file \text{prog1.C}\

- note the order of the constructor and desctructor calls when the program is run

- desctructor is called in the reverse order of creation

- note the \text{new-delete} duo for manual memory management

- note the \text{delete mcs3} in turn calls the destructor function \(~\text{MonteCarloSpecs}( )\)
Class Implementation X

- constructors for `const` and reference class members need special care
- they must be initialized through initializer list
- you could also do error checking in the initializer list through a function call
Class Implementation XI

- suppose we change our class declaration a little bit like so:

```cpp
class MonteCarloSpecs {
private:
    int n_iters;
    float time_in_secs;
    float prop_burn_in;
    double *log_density;
    string const name_of_algo;
    int &debug_level;
    int &check_debug_level (int &debug_level);

public:
    MonteCarloSpecs (int n_iters, float time_in_secs, float prop_burn_in, 
                     string const name_of_algo, 
                     int &debug_level);
    ~MonteCarloSpecs (void);
    int get_n_iters (void) const;
    void set_n_iters (int n_iters);
    float get_time_in_secs (void) const;
    void set_time_in_secs (float time_in_secs);
    float get_prop_burn_in (void) const;
    void set_prop_burn_in (float prop_burn_in);
    void print (void) const;
};
```
Class Implementation XII

- the implementation of the constructor should change to:

```cpp
MonteCarloSpecs::MonteCarloSpecs (int n_iters,
    float time_in_secs,
    float prop_burn_in,
    string const name_of_algo,
    int &debug_level)

    : name_of_algo(name_of_algo),
    debug_level(check_debug_level(debug_level))
{
    cout << "Creating a MonteCarloSpecs object with n_iters = "
    << n_iters << endl;
    log_density = NULL;
    set_n_iters (n_iters);
    set_time_in_secs (time_in_secs);
    set_prop_burn_in (prop_burn_in);
}
```
Class Implementation XIII

- the error-checking function `check_debug_level()` is declared private because the user of the class do not have any use for this

- the implementation of `check_debug_level()` might look like:

```cpp
int & MonteCarloSpecs::check_debug_level (int &debug_level)
{
    assert (debug_level >= 0);
    assert (debug_level <= 2);
    return debug_level;
}
```
Code Files

prog1.H
prog1.C
prog1Makefile
prog5.H
prog5.C
prog5Makefile