Solutions to Problems

10.1 This implementation of a `Point` class uses the common device of ending the name of each data member with an underscore (_). This has the advantage of making it easy to match up the names of constructor parameters (`x`, `y`, and `z`) with their corresponding data members (`x_`, `y_`, and `z_`) without conflict.

```
#include <cmath>
#include <iostream>
using namespace std;
class Point
{ public:
    Point(float x=0, float y=0, float z=0): x_(x), y_(y), z_(z) {}
    Point(const Point& p) : x_(p.x_), y_(p.y_), z_(p.z_)  { }
    void negate() { x_ *= -1; y_ *= -1; z_ *= -1; }
    double norm() { return sqrt(x_*x_ + y_*y_ + z_*z_); }
    void print()
      { cout << '(' << x_ << "," << y_ << "," << z_ << ")"; }
  private:
    float x_, y_, z_;
};
```

10.2 In this implementation of a `Stack` class, `top` is always the index of the top element on the stack. The data member `size` is the size of the array that holds the stack items. So the stack is full when it contains that number of items. The constructor sets `size` to 10 as the default.

```
class Stack
{ public:
    Stack(int s=10) : size(s), top(-1) { a = new int[size]; }
    ~Stack() { delete [] a; }
    void push(const int& item) { a[++top] = item; }
    int pop() { return a[top--]; }
    bool isEmpty() const { return top == -1; }
    bool isFull() const { return top == (size-1); }
  private:
    int size;  // size of array
    int top;   // top of stack
    int* a;    // array to hold stack items
};
```

10.3

```
class Time
{ public:
    Time(int h=0, int m=0, int s=0)
      : hr(h), min(m), sec(s) { normalize(); }
    int hours() { return hr; }
    int minutes() { return min; }
    int seconds() { return sec; }
    void advance(int =0, int =0, int =1);
    void reset(int =0, int =0, int =0);
    void print() { cout << hr << ":" << min << ":" << sec; }
  private:
    int hr, min, sec;
    void normalize();
};
void Time::normalize()
```

```
{ min += sec/60;
  hr += min/60;
  hr = hr % 24;
  min = min % 60;
  sec = sec % 60;
}
void Time::advance(int h, int m, int s)
{ hr += h;
  min += m;
  sec += s;
  normalize();
}
void Time::reset(int h, int m, int s)
{ hr = h;
  min = m;
  sec = s;
  normalize();
}
```

10.4 This implementation of a Random class uses a utility function `normalize()`, which normalizes the `Time` object so that its three data members are in the correct range: $0 \leq$ `sec` < 60, $0 \leq$ `min` < 60, and $0 \leq$ `hr` < 24. It also uses the utility function `randomize()`, which implements the *Linear Congruential Algorithm* introduced by D. H. Lehmer in 1949. The utility function `_next()` updates the `_seed` by calling the `_randomize()` function a random number of times.

```
#include <iomanip>
#include <iostream>
#include <limits>
#include <ctime>
using namespace std;
class Random
{ public:
    Random(long seed=0) { _seed = ( seed?seed:time(NULL) ); }
    void seed(long seed=0) { _seed = ( seed?seed:time(NULL) ); }
    int integer() { return _next(); }
    int integer(int min, int max)
      { return min +_next()%(max-min+1);}
    double real()
      { return double(_next())/double(INT_MAX); }
  private:
    unsigned long _seed;
    void _randomize()
      { _seed = (314159265*_seed + 13579)%ULONG_MAX;}
    int _next()
    { int iterations = _seed % 3;
      for (int i=0; i <= iterations; i++) _randomize();
      return int(_seed/2);
    }
};
int main()
{ Random random;
  for (int i = 1; i <= 10; i++)
    cout << setw(16) << setiosflags(ios::right)
         << random.integer()
         << setw(6) << random.integer(1,6)
```

```
          << setw(12) << setiosflags(ios::fixed | ios::left)
          << random.real() << endl;
}
```

The test driver makes 10 calls to each of the three random number functions, generating 10 pseudo-random integers in the range 0 to 2,147,483,647, 10 pseudo-random integers in the range 1 to 6, and 10 pseudo-random real numbers in the range 0.0 to 1.0.

10.5

```
class Person
{ public:
    Person(const char* =0, int =0, int =0);
    ~Person() { delete [] name_; }
    char* name() { return name_; }
    int born() { return yob_; }
    int died() { return yod_; }
    void print();
  private:
    int len_;
    char* name_;
    int yob_, yod_;
};
Person::Person(const char* name, int yob, int yod)
  : len_(strlen(name)),
    name_(new char[len_+1]),
    yob_(yob),
    yod_(yod)
{ memcpy(name_, name, len_+1);
}
void Person::print()
{ cout << "\tName: " << name_ << endl;
  if (yob_) cout << "\tBorn: " << yob_ << endl;
  if (yod_) cout << "\tDied: " << yod_ << endl;
}
```

To keep the object self-contained, `name_` is stored as a separate string. To facilitate this separate storage, we save its length in the data member `len_` and use the `memcpy()` function (defined in `string.h`) to copy the string `name` into the string `name_`. Then the destructor uses the delete operator to de-allocate this storage.

10.6 This implementation of a String class includes three constructors: the default constructor with optional parameter `size`, a constructor that allows an object to be initialized with an ordinary C string, and the copy constructor. The second access function is named `convert()` because it actually converts from type `String` to `char*` type. The "subscript" function is named `character()` because it returns one character in the string—the one indexed by the parameter `i`.

```
class String
{ public:
    String(short =0);                  // default constructor
    String(const char*);               // constructor
    String(const String&);             // copy constructor
    ~String() { delete [] data; }      // destructor
    int length() const { return len; } // access function
    char* convert() { return data; }   // access function
    char character(short i) { char c = data[i]; return c; }
    void print() { cout << data; }
  private:
    short len;     // number of (non-null) characters in string
    char* data;    // the string
```