Prototypal_C Readme and Mini-tutorial

//Prototypal_C is a header implementing a class that allows users to write dynamic, type-safe, prototypal inheritance based c++ code. The Object class contained within this header can be instantiated to create generic containers capable of augmenting themselves with members and functions of various types. Members are accessed indirectly by passing a string to a "get" function.

```
//Example:
  Object object;
  int x = 5;
  object.set("x",x);             // add a new member x to object and set its name to "x".
  int i = object.get<int>("x");   // get member of type int whose name is "x".
  std::cout << i << std::endl;    // print 5.

//==================================================================
```

//One member function can be called directly using the "call" method, which avoids the overhead of object storage and retrieval. This feature allows members of type Object to have their own persistent variables.

```
//Example:
  struct ss{static void print() {std::cout << "hello world" << std::endl;} };
  object.setFunc(ss::print);  // sets object's function pointer to the print function.
  object.call();                // directly calls the print function.


//==================================================================
```

//Note that only static global functions, non-static global functions, and non-static class member functions can be passed using setFunc() and call(). Member functions setFunc() and call() can also be used to pass parameters of primitive types or pointers to class types.

```
//Example:
  struct vv {static void func(Object * o, int x) {} };
  object.setFunc(vv::func );    // sets object's function pointer to the func function.
  Object * ob = &object;
  object.call(ob, x);       // directly calls the func function.
//==================================================================
```

//Also note that for functions returning non-void, the return type must be a pointer whose contents will be allocated on heap.

```
//Example:
  struct ww {static int * add(int x, int y) {return new int(x+y);} };
  object.setFunc(ww::add);      // sets object's function pointer to the add function.
  x = object.call<int>(5, 6);     // returns 11. The dereferenced return type is specified in angle brackets.
  std::cout << x << std::endl;
```

//==

//Members of type Object can designate a parent and pass searches for variables and function calls to their parent.

//Example:

```
  Object child;
  child.my_parent = &object;
  object.my_parent = nullptr;
  std::cout << object.get<int>("x") << std::endl;           // gets the member of type int whose name is "x"
from object. prints 5.
```

//==

//Direct function calls can also be designated object parent if they are equal to the default value, nullptr, in the child.

//Example:

```
  std::cout << child.call<int>(5, 6) << std::endl;         // returns 11. The dereferenced return type is
specified in angle brackets.
```

//==

//Indirect function calls can also be performed. Objects which can directly call a function can be placed inside of other objects using the "object.set(std::string name, Type T)" method. An outer object can call an inner object by with the method "object.Do<Return_Type T>(std::string name, Parameter_Pack P)".

//Example:

```
  object.set("child", child);     // add a new member child to object and set its name to "child".
  std::cout << object.Do<int>("child", 5, 6) << std::endl;  // tells member whose name is "child" to
perform the call function.
```

==

//Note that "Object child" is inside of "Object object" and that "Object child" has access to "Object object". This pattern can also be applied to subclasses of Object.

//For Example:

```
class Computer : public Object {};
class Printer : public Object {};
Computer comp;
Printer p;
p.setFunc(ss::print);
comp.set("print",p);
comp.Do("print");                // Computer calls Printer's print function.
```

==

In conclusion, by using the Prototypal_C header with the above functions and design patterns, c++ programmers can implement various design patterns and programming techniques that are not readily availible in the language.