

```

1  //=====
2  // Section 13: OOP - Classes and Objects
3  //=====
4
5  // What is OOP
6  Procedural programming
7  - Focus on processes or actions the program takes
8  - Collection of functions
9  - Data is declared separately
10 - Passed into functions as arguments
11 - Fairly easy to learn
12
13 Limitations:
14 functions need to know about the structure of the data
15 If it changes the functions probably need to change too
16
17 As programs get bigger they get:
18 - Difficult to understand, maintain, extend, debug, etc
19 - Hard to reuse code
20 - Fragile and easier to break
21
22 // Classes and objects
23 - Classes model real world domain entities
24 - higher levels of abstraction
25 - best for large programs
26
27 // Encapsulation
28 - Objects contain data and operations that work on the data
29 - ADT - Abstract data type
30
31 // Information hiding
32 - Implementation-specific logic can be hidden
33 - Users of the class code to the interface since they know nothing else
34 - More abstraction
35 - Easier to test maintain and debug.
36
37 // Reusability
38 - Easier to reuse classes in other applciation
39 - Faster development
40 - Higher quality
41
42 // Inheritance
43 - Create a new class based on an existing class
44 - Helps us reuse classes
45 - Polymorphic classes
46
47 // Limitations
48 Not a cure all
49 - Won't make bad code better, it probably makes it worse
50 - Not suitable for all problems
51 - Not everything decomposes to a class
52
53 Learning curve
54 - Steeper learning curve, especially for C++
55 - Many OO languages and variations of OO concepts
56
57 Design
58 - Usually more up front design is needed for good models and hierarchies
59
60 Programs can be:
61 - Large in size
62 - slower
63 - more complex
64
65
66
67
68
69

```

```

70 // What are classes and objects?
71 Classes
72 - Blueprints from which objects are created
73 - User defined
74 - has data and functions (methods and attributes)
75 - Can hide data and methods
76 - provide a public interface
77
78
79 Objects
80 - Created from classes
81 - Represents an instance of the class
82 - We can have lots of objects
83 - Each has its own identity
84 - Each can use the defined class methods
85
86 // Declaring a class and creating objects
87 class ClassName{ // capitalize class names
88     // declarations
89 }; // semicolon is required
90
91 class Player{
92     // attributes
93     std::string name;
94     int health;
95     int xp;
96
97     // methods
98     void talk(std::string textToSay);
99     bool isDead();
100 }; // Remember the semicolon after the class
101
102 // After a class is declared we can create objects from it
103 Player Yaya;
104 Player hero;
105 Player * enemy = new Player(); // can create pointers on the heap
106 delete enemy; // delete it then
107
108 // Accessing class members
109 We can access:
110 - attributes
111 - methods
112
113 some class members will not be accessible
114 We need an object to access instance variables
115
116 If we have an object (dot operator)
117 Account myAccount;
118 myAccount.balance;
119 myAccount.deposit(1000.00);
120
121 If we have a pointer (two ways)
122 - Dereference then use the dot operator
123
124 Account * myAccount = new Account();
125
126 (*myAccount).balance; // dot operator has higher precedence
127 (*myAccount).deposit(1000.00);
128
129 Alternatively, use the member of pointer operator (->)
130 myAccount->balance;
131 myAccount->deposit(1000.00);
132
133
134
135
136
137
138

```

```

139 //=====
140 // Example
141 //=====
142 #include <iostream>
143 #include <vector>
144 #include <string>
145 #include <cmath>
146 #include <ctime>
147
148 class Player {
149 public: // by default all members are private
150     // Attributes
151     std::string name;
152     int health;
153     int xp;
154
155     // methods
156     void talk(std::string incomingString) {
157         std::cout << name << " says: " << incomingString << std::endl;
158     };
159     bool isDead() {};
160 };
161
162
163 int main(){
164
165     Player yaya;
166     yaya.name = "Yaya";
167     yaya.health = 200;
168     yaya.xp = 512;
169     std::cout << yaya.name << " " << yaya.health << " " << yaya.xp << std::endl;
170     yaya.talk("I am the mighty yaya");
171     Player* babs = new Player();
172     babs->name = "Bablet";
173     (*babs).health = 200; // Dereference syntax - use the arrow instead
174     babs->xp = 34;
175     std::cout << babs->name << " " << babs->health << " " << babs->xp << std::endl;
176     babs->talk("Babs is the best");
177
178     return 0;
179 }
180 //=====
181 // Example
182 //=====
183
184 // Public and private
185 public
186 - accessible everywhere
187
188 private
189 - accessible only by members or friends of the class
190
191 protected
192 - used with inheritance
193
194 class ClassName{
195     private:
196     // Stuff in here is private
197
198     public:
199     // Stuff in here is public
200
201     protected:
202     // Stuff in here is protected
203 }
204
205 Compiler will not let you access private class members
206
207

```

```

208 // Implementing member methods
209 Similar to how we implemented functions
210 Member methods have access to member attributes
211 -No need to pass arguments
212
213 Can be implemented inside the class declaration
214 - Implicitly inline
215
216 Can be implemented outside
217 - Use scope resolution to access: ClassName::methodName
218
219 Separate specification from the implementation
220 .h file for the class declaration
221 .cpp for the implementation
222
223 //Separating classes out
224
225 Put in Account.h file
226 #ifndef _ACCOUNT_H_
227 #define _ACCOUNT_H_
228 class Account{
229     private:
230         double balance;
231     public:
232         void setBalance(double bal);
233         double getBalance();
234 }
235 #endif
236
237 //Put in Account.cpp file
238 #include "Account.h"
239 void Account::setBalance(double bal){
240     balance = bal;
241 }
242
243 double Account::getBalance(){
244     return balance;
245 }
246
247 // Include guards to make sure the compiler processes this only once
248 #ifndef _ACCOUNT_H_
249 #define _ACCOUNT_H_
250
251 #endif
252
253 or use #pragma once
254
255 // Example Account.h file
256 #pragma once
257 #include <string>
258 class Account {
259 private:
260     // attributes
261     std::string name;
262     double balance;
263 public:
264     // Methods
265     void setBalance(double bal);
266     double getBalance();
267     void setName(std::string n);
268     std::string getName();
269     bool deposit(double amount);
270     bool withdraw(double amount);
271 };
272
273
274
275
276

```

```

277 // Example Account.cpp file
278 #include "Account.h"
279 void Account::setBalance(double bal) {
280     balance = bal;
281 }
282 double Account::getBalance() {
283     return balance;
284 }
285
286 void Account::setName(std::string n) {
287     name = n;
288 }
289
290 std::string Account::getName() {
291     return name;
292 }
293 bool Account::deposit(double amount) {
294     balance += amount;
295     return true;
296 }
297 bool Account::withdraw(double amount) {
298     if (balance - amount > 0) {
299         balance -= amount;
300         return true;
301     } else {
302         return false;
303     }
304 }
305
306 // Example main.cpp file
307 #include <iostream>
308 #include <vector>
309 #include <string>
310 #include <cmath>
311 #include <ctime>
312 #include "Account.h"
313
314 int main(){
315
316     Account josieAccount;
317     josieAccount.setName("Josie Revisited");
318     josieAccount.setBalance(1000.00);
319
320     if (josieAccount.deposit(200)) {
321         std::cout << "Deposit ok" << std::endl;
322     } else {
323         std::cout << "Deposit not allowed" << std::endl;
324     }
325
326     if (josieAccount.withdraw(500.00)) {
327         std::cout << "Withdrawal okay" << std::endl;
328     } else {
329         std::cout << "Not sufficient funds" << std::endl;
330     }
331
332     if (josieAccount.withdraw(1500)) {
333         std::cout << "Withdrawal okay" << std::endl;
334     } else {
335         std::cout << "Not sufficient funds" << std::endl;
336     }
337     return 0;
338 }
339
340
341
342
343
344
345

```

```

346 // Constructors and destructors
347 Constructors
348 - Special member method
349 - Invoked during object creation
350 - Used for initialization
351 - Same name as the class
352 - No return type
353 - Can be overloaded - multiple constructors
354
355 class Player{
356     private:
357         // stuff
358     public:
359         // Overloaded constructors
360         Player();
361         Player(std::string name);
362         Player(std::string name, int health, int xp);
363 }
364
365 Destructors
366 - special member method
367 - Same name as class preceded with a ~
368 - Invoked automatically when an object is destroyed
369 - No return type and no parameters
370 - Only one destructor is allowed per class - we can't overload these
371 - Useful to release memory and other resources
372
373 class Player{
374     private:
375         // stuff
376     public:
377         // Overloaded constructors
378         Player();
379         Player(std::string name);
380         Player(std::string name, int health, int xp);
381
382         // Destructor
383         ~Player();
384 }
385
386 // Creating objects
387 {
388     Player slayer; // no args constructor called
389     Player Yaya{"Yaya", 100, 10}; // Three args constructor called
390     Player Babs{"Babs"}; // One arg constructor called
391     Player * enemy = new Enemy("Fiteme", 1000, 0); //Works the same with pointers
392     delete enemy; // destructor called
393 } // Destructor called when the other objects go out of scope
394
395 NOTE: If we don't provide one, C++ will give us a constructor and destructor for free!
396 But they are empty.
397
398 // Default constructors
399 Default constructor
400 - Does not expect any arguments (aka the no-args constructor)
401 - If you write no constructors C++ will generate one
402 - Called when you create an object with no arguments
403 - Best practice is to define our own no-args constructor
404 - If you define a constructor that has argument, you no longer get a free default
  constructor from C++
405 - If we need it, we need to define it ourselves
406 - We would no longer be able to create objects with no arguments
407
408
409
410
411
412
413

```

```

414 class Player{
415     private:
416         std::string name;
417         int health;
418         int xp;
419     public:
420         // Overloaded constructors
421         Player(){ // default no-args constructor
422             name = "none";
423             health = 100;
424             xp = 0;
425         }
426         Player(std::string name){
427             name = name;
428             health = 100;
429             xp = 0;
430         }
431         Player(std::string name, int health, int xp){
432             name = name;
433             health = health;
434             xp = xp;
435         }
436
437         // Destructor
438         ~Player(){};
439     }
440
441     // Overloading constructors
442     Classes can have as many constructors as necessary
443     Each must have a unique signature
444     Default constructor is no longer compiler-generated once another constructor is declared
445     Make sure that you initialize all member variables so that nothing contains garbage data
446
447     // Constructor initialization lists
448     More efficient
449     Initialization list immediately follows the parameter list
450     Initializes the data members as the object is created
451     Order of initialization is the order of declaration in the class
452     So far what we have been doing isn't initialization. It is assignment.
453
454     // Using an initializer list
455     Player::Player()
456         : name{"None"}, health{0}, xp{0}
457     { // whatever other statements necessary in here }
458
459     class Player{
460     private:
461         std::string name;
462         int health;
463         int xp;
464     public:
465         // Overloaded constructors
466         Player()
467             : name{"None"}, health{0}, xp{0}{ // default no-args constructor
468         }
469         Player(std::string name)
470             : name{name}{
471         }
472         Player(std::string name, int health, int xp)
473             : name{name}, health{health}, xp{xp}{
474         }
475
476         // Destructor
477         ~Player(){};
478     }
479
480
481
482

```

```

483 // Delegating constructors
484 Often the code for constructors is very similar
485 Duplicated code can lead to errors
486 C++ allows delegating constructors
487 - Code for one constructor can call another in the initialization list
488 - Avoids duplicating code
489
490 class Player{
491     private:
492         std::string name;
493         int health;
494         int xp;
495     public:
496         // Overloaded constructors
497         Player(std::string name, int health, int xp)
498             : name{name}, health{health}, xp{xp}{
499         }
500
501         Player()
502             : Player{"None", 0,0}{ // This calls the three args constructor with these values
503         }
504         Player(std::string name)
505             : Player{name, 0, 0}{ // Again, calls the three args constructor and passes in
506                                     these values
507         }
508
509         // Destructor
510         ~Player(){};
511     }
512
513     // Constructor parameters and default
514     This can simplify the code and reduce the number of overloaded constructors
515     Same rules apply as with non-member functions
516     You can't always have default values, but many times you can
517
518
519 class Player{
520     private:
521         std::string name;
522         int health;
523         int xp;
524     public:
525         // Constructor prototype with default parameters. We can now create objects with
526         // any number of args
527         Player(std::string name = "None", int health = 100, int xp = 0)
528         {}
529
530         // Destructor
531         ~Player();
532     }
533
534     // Constructor implementation outside the class
535     Player::Player(std::String name, int health, int xp)
536         :name{name}, health{health}, xp{xp}{
537     }
538
539     // Copy constructor
540     When objects are copied, C++ must create a new object from an existing one
541     When do we do this?
542     - Passing an object by value as a parameter
543     - Returning an object from a function by value
544     - Constructing one object based on another of the same class
545     - C++ MUST have a way to do this so if we don't specify, the compiler will make one for
546     us
547
548

```



```

549 Use cases
550 When we pass an object to a function by value (copy is made)
551 // Create an object
552 Player hero{"Hero", 100, 20};
553
554 // Some random display function
555 void displayPlayer(Player playerObjectByValue){
556     // playerObjectByValue is a COPY of hero in this example
557     // the function will do whatever it needs to with the copy
558     // The destructor for playerObjectByValue will then be called
559 }
560
561 // Call our function - it needs to make a copy of hero to do its job
562 displayPlayer(hero);
563
564 // What do they do?
565 If you don't provide one, the compiler will
566 Copies the values of each data member to the new object
567 - Defaults to memberwise copy - which is each member is just directly copied over
568
569 Perfectly fine in many cases
570 Beware if you have a pointer as a data member
571 - The pointer will be copied, but not what it is pointing to
572 - This is a shallow copy and you will run into issues with two pointers pointing to the
    same data
573
574 // Best practices
575 If you are using raw pointers, provide your own copy constructor
576 Create it with a const reference parameter
577 Use STL classes when you can - they already have their own copy constructors
578 Avoid using raw pointers if you can
579
580 // Declaring the copy constructor
581 Type::Type(const Type &source); // it passes in a single object as a constant
    reference so we don't damage the source
582 Player::Player(const Player&source);
583
584 // Now we can do whatever initialization we need
585 Player::Player(const Player&source){
586     // We can put in whatever code or "assignment" style initialization we need
587 }
588
589 // Or we can use the initializer list (preferred)
590 Player::Player(const Player&source)
591 : name{source.name}, health{source.health}, xp{source.xp}{
592     // Any other stuff here
593 }
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615

```

```

616 //=====
617 // Copy constructor example program
618 //=====
619 #include <iostream>
620 #include <vector>
621 #include <string>
622 #include <cmath>
623 #include <ctime>
624
625 class Player {
626 private:
627     std::string name;
628     int health;
629     int xp;
630
631 public:
632     std::string getName() { return name; }
633     int getHealth() { return health; }
634     int getXp() { return xp; }
635     // Constructor
636     Player(std::string name = "None", int health = 100, int xp = 0);
637
638     // Copy Constructor
639     Player(const Player& source);
640
641     // Destructor
642     ~Player() {
643         std::cout << "Destructor called for: " << name << std::endl;
644     };
645 };
646
647 // Create a function that expects a Player object - this will need to use the copy
648 // constructor
649 void displayPlayer(Player p) {
650     std::cout << "Name: " << p.getName() << std::endl;
651     std::cout << "Health: " << p.getHealth() << std::endl;
652     std::cout << "XP: " << p.getXp() << std::endl;
653 }
654
655 Player::Player(std::string name, int health, int xp)
656 : name{ name }, health{ health }, xp{ xp } {
657     std::cout << "Three args constructor with initiliazer list to take care of
658     everything" << std::endl;
659 }
660
661 Player::Player(const Player& source)
662 : name(source.name), health(source.health), xp(source.xp) {
663     std::cout << "Copy constructor - made a copy of: " << source.name << std::endl;
664 }
665
666 int main(){
667     Player empty;
668     displayPlayer(empty);
669     Player Josie{ "Josie" };
670     Player hero{ "Hero", 100 };
671     Player villain{ "Villain", 1000, 1000 };
672
673     return 0;
674 }
675
676
677
678
679
680
681
682

```

```

683 //=====
684 // Copy constructor example program
685 //=====
686
687 // Note that we could also use a delegating constructor
688 Player::Player(const Player& source)
689    //: name(source.name), health(source.health), xp(source.xp) {
690     :Player{source.name, source.health, source.xp}{
691     std::cout << "Copy constructor - made a copy of: " << source.name << std::endl;
692 }
693
694 // Shallow copying
695 If a class has a raw pointer, the constructor allocates storage and initializes the
696 pointer
697 The destructor then releases that memory
698
699 Default copy constructor:
700 memberwise copy - each object attribute copied as-is
701 The pointer is copied, but NOT what it's pointing to.
702 The problem is when one of the objects goes out of scope or the destructor is called, the
703 allocated memory is released but there is still a pointer out there that thinks it's
704 valid
705
706 // Shallow example
707 class Shallow{
708     private:
709         int * data;
710     public:
711         // Constructor
712         Shallow(int d);
713
714         // Copy constructor
715         Shallow(const Shallow &source);
716
717         // Destructor
718         ~Shallow(){}
719 }
720
721 // Constructor implementation
722 Shallow::Shallow(int d){
723     data = new int; // Allocate space on the heap for it
724     *data = d; // Assign the value we passed in
725 }
726
727 // Destructor implementation
728 Shallow::~Shallow(){
729     delete data;
730 }
731
732 // Copy constructor implementation
733 Shallow::Shallow(const Shallow & source)
734     : data(source.data){
735     // Both the copy and the source will point to the same object!
736 }
737
738
739
740
741
742
743
744
745
746
747
748
749

```

```

750 //=====
751 // Shallow copy constructor example program
752 //=====
753 #include <iostream>
754 #include <vector>
755 #include <string>
756 #include <cmath>
757 #include <ctime>
758 #include ".././std_lib_facilities.h"
759
760
761 class Shallow {
762 private:
763     int* data;
764
765 public:
766     void setData(int d) { *data = d; }
767     int getDataValue() {
768         return *data;
769     }
770
771     //Constructor
772     Shallow(int d);
773
774     // Copy Constructor
775     Shallow(const Shallow& source);
776
777     // Destructor
778     ~Shallow();
779 };
780
781 Shallow::Shallow(int d) {
782     data = new int;
783     *data = d;
784 }
785
786 Shallow::Shallow(const Shallow& source)
787     :data(source.data) {
788     std::cout << "Copy Constructor - shallow copy" << std::endl;
789 }
790
791 Shallow::~Shallow() {
792     delete data;
793     std::cout << "Destructor freeing data" << std::endl;
794 }
795
796 void displayShallow(Shallow shallowObjectToCopy) {
797     std::cout << shallowObjectToCopy.getDataValue() << std::endl;
798 }
799
800
801 int main(){
802
803     Shallow object1{ 100 };
804     displayShallow(object1);
805
806     Shallow object2{ object1 };
807     object2.setData(1000);
808
809     return 0;
810 }
811
812
813
814
815
816
817
818

```

```

819 // Deep copying with the copy constructor
820
821 This copies the pointer and the data it points to
822 Each copy has a pointer to unique storage in the heap
823 Always use when you have a raw C++ pointer
824
825 class Deep{
826     private:
827         int * data;
828     public:
829         Deep (int d);
830         Deep (const Deep &source);
831         ~Deep();
832 }
833
834 Deep::Deep(int d){
835     data = new int;
836     *data = d;
837 }
838
839 Deep::Deep(const Deep &source){
840     data = new int;
841     *data = *source.data;
842     std::cout << "Deep copy of the data" << std::endl;
843 }
844
845 Deep::~Deep(){
846     delete data; // Free storage
847 }
848
849 // We can also use delegation
850 Deep::Deep(const Deep &source)
851 :Deep{*source.data}{
852     std::cout << "Copy constructor - deep" << std::endl;
853 }
854
855 //=====
856 // Deep copy constructor example program
857 //=====
858 #include <iostream>
859 #include <vector>
860 #include <string>
861 #include <cmath>
862 #include <ctime>
863 #include "../std_lib_facilities.h"
864
865 class Deep {
866     private:
867         int* data;
868
869     public:
870         void setData(int d) { *data = d; }
871         int getDataValue() {
872             return *data;
873         }
874
875         //Constructor
876         Deep(int d);
877
878         // Copy Constructor
879         Deep(const Deep& source);
880
881         // Destructor
882         ~Deep();
883 };
884
885
886
887

```

```

888 Deep::Deep(int d) {
889     data = new int;
890     *data = d;
891 }
892
893 Deep::Deep(const Deep& source)
894     :Deep{ *source.data } { // In this case we do a deep copy by delegating to the main
    constructor
895     std::cout << "Copy Constructor - shallow copy" << std::endl;
896 }
897
898 Deep::~Deep() {
899     delete data;
900     std::cout << "Destructor freeing data" << std::endl;
901 }
902
903 void displayDeep(Deep shallowObjectToCopy) {
904     std::cout << shallowObjectToCopy.getDataValue() << std::endl;
905 }
906
907 int main() {
908
909     Deep object1{ 100 };
910     displayDeep(object1);
911
912     Deep object2{ object1 };
913     object2.setData(1000);
914
915     return 0;
916 }
917
918 // Move Constructor
919 Sometimes when we execute code the compiler will create unnamed temp values
920 int total{0};
921 total = 100 + 200;
922
923 100 + 200 is evaluated and the result 300 is stored in an unnamed temp value
924 Then the 300 is stored in the variable total
925 Then the temp value is discarded
926 The same happens with objects
927 It's important to tell whether an expression is an L-Value or an R-Value
928
929 // When are move semantics useful?
930 Sometimes copy constructors are called a LOT because of how we copy things
931 The ones doing deep copy can take a LOT of time
932 Sometimes it makes more sense to move the object rather than copy it
933 It's optional, but recommended when you use a raw pointer
934 Copy elision - C++ may automatically eliminate the unnecessary copying
935 RVO - Return Value Optimization - the compiler generates code that doesn't copy a return
936 value from a function. Makes it more efficient
937
938 //R-Value references
939 Used in move semantics and perfect forwarding
940 Move semantics are all about r-value references - values that can't be addressed
941 Used by the move constructor and move assignment operator to efficiently move an object
    rather than copy it
942 R-Value reference operator is the double ampersand (&&)
943
944 // Examples
945 L-Value references:
946 int x{100}; // In this instance, x is an l-value. It's addressable and it has a name
947 int &lRef = x; // l-value reference
948 lRef = 10;
949
950 R-Value references
951 int &&rReference = 200; // R-value reference
952 rReference = 300; // Change rReference to 300;
953 int &&xReferernce = x; // We can't assign an L-Value to it - it gives a compiler error
954

```

```

955 // Dealing with this in functions
956 void func(int &&num);
957 func(200); // This is okay - it's an r-value
958 func(x); // this won't be okay, because x is an l-value
959
960 // Overloading them
961 void func(int &num); // Pass my integer by reference
962 void func(int &&num); // Pass my integer as a r-value
963
964 Now we can either use x or 300
965
966 // Move constructor
967 class Move{
968     private:
969         int *data;
970
971     public:
972         void setData(int d){*data = d;} // Setter
973         int getData(){return *data;} // Getter
974         Move(int d); // Constructor
975         Move(const Move &source); // Copy Constructor
976         ~Move(); // Destructor
977 }
978
979 // Copy constructor
980 Move::Move(const Move &source){
981     data = new int;
982     *data = *source.data;
983 }
984
985 // The copy constructors would be called to copy these temps - this can be inefficient
986 // since it will do a LOT of copies
987 Vector<Move> vec;
988 vec.push_back(Move{10});
989 vec.push_back(Move{20});
990
991 // What does the move constructor do?
992 Instead of making deep copies, it moves resources
993 Copies the address of the source to the destination and nulls out the source
994 Very efficient
995
996 // Syntax for r-value references - note cannot be const
997 Type::Type (Type &&source);
998 Player::Player(Player &&source);
999
1000 class Move{
1001     private:
1002         int *data;
1003
1004     public:
1005         void setData(int d){*data = d;} // Setter
1006         int getData(){return *data;} // Getter
1007         Move(int d); // Constructor
1008         Move(const Move &source); // Copy Constructor
1009         Move(Move &&source); // Move constructor
1010         ~Move(); // Destructor
1011 }
1012
1013 Move::Move(Move &&source)
1014 : data{source.data}{ // Steal the data
1015     source.data = nullptr; // null the original data - if we don't do this we end
1016     // up being a shallow copy instead
1017     // and we will have pointers all over the place
1018 }
1019
1020 // This time these will call the Move constructor since they are r-values being passed in
1021 Vector<Move> vec;
1022 vec.push_back(Move{10});
1023 vec.push_back(Move{20});

```

```

1022 //=====
1023 // Move copy constructor example program
1024 //=====
1025
1026 #include <iostream>
1027 #include <vector>
1028 #include <string>
1029 #include <cmath>
1030 #include <ctime>
1031 // #include "../..//std_lib_facilities.h"
1032
1033 class Move {
1034 private:
1035     int* data;
1036 public:
1037     void setDataValue(int d) { *data = d; }
1038     int getDataValue() { return *data; }
1039     Move(int d); // constructor
1040     Move(const Move& source); // Copy Constructor
1041     Move(Move&& source) noexcept; // Move constructor
1042     ~Move(); // Destructor
1043 };
1044
1045 // Constructor to create the object
1046 Move::Move(int d) {
1047     data = new int;
1048     *data = d;
1049     std::cout << "Constructor for: " << d << std::endl;
1050 }
1051
1052 // Copy constructor - this is doing a deep copy
1053 Move::Move(const Move& source)
1054     : Move{ *source.data } {
1055     std::cout << "Copy Cosntructor - deep copy for: " << *data << std::endl;
1056 }
1057
1058 // Move constructor
1059 Move::Move(Move&& source) noexcept
1060     : data{ source.data } {
1061     source.data = nullptr;
1062     std::cout << "Move constructor - moving the resource: " << *data << std::endl;
1063 }
1064
1065 Move::~~Move() {
1066     if (data != nullptr) {
1067         std::cout << "Destructor freeing data for: " << *data << std::endl;
1068     } else {
1069         std::cout << "Destructor freeing data for nullptr." << std::endl;
1070     }
1071     delete data;
1072 }
1073
1074 int main(){
1075     std::vector<Move> moveObjects;
1076     moveObjects.push_back(Move{ 20 });
1077     moveObjects.push_back(Move{ 30 });
1078     moveObjects.push_back(Move{ 40 });
1079     moveObjects.push_back(Move{ 50 });
1080     moveObjects.push_back(Move{ 60 });
1081     return 0;
1082 }
1083
1084
1085
1086
1087
1088
1089
1090

```



```

1091 // The 'this' pointer
1092 This
1093 - reserved keyword
1094 - Contains the address of the object - so it's technically a pointer to the object
1095 - Can only be used in the class scope
1096 - All member access is done via the this pointer
1097 - Can be used by the programmer to:
1098 -- Access data members and methods
1099 -- Determine if two objects are the same
1100 -- Deferenced to give us the current object
1101
1102 // ways to write classes
1103 void Account::setBalance(double balance){
1104     this->balance = balance;
1105 }
1106
1107 // Check if two objects are the same
1108 int Account::compareBalance(const Account &other){
1109     if (this == &other){
1110         std::cout << "They are the same" << std::endl;
1111     }
1112 }
1113
1114 yayaAccount.compare(yayaAccout);
1115
1116 //=====
1117 // Using const with classes
1118 //=====
1119
1120 We can pass arguments to class member methods as 'const'
1121 We can also create const objects
1122 What happens if we call member funtions on const objects? Maybe bad stuff. We need to be
1123     const correct
1124
1125 // Creating a const object
1126 const Player villian{"Villain", 100, 55}; // this creates a const object and the
1127     attributes cannot be changed
1128
1129 // Calling member methods
1130 Even when you call getter methods, the compiler sees that it could potentially change
1131 data and it won't compile
1132 void displayPlayerName(const Player &p){
1133     std::cout << p.getName() << std::endl;
1134 }
1135
1136 displayPlayerName(villain); // This will fail because the compiler sees that it
1137     could potentially change something
1138
1139 // How do we solve this?
1140 Mark the methods we want to use as const
1141 class Player{
1142     private:
1143         // stuff
1144     public:
1145         std::string getName() const;
1146 }
1147
1148 Now the calls to the getters will work
1149
1150
1151
1152
1153
1154
1155

```

```

1156 //=====
1157 // Const-correctness example program
1158 //=====
1159
1160 #include <iostream>
1161 #include <vector>
1162 #include <string>
1163 #include <cmath>
1164 #include <ctime>
1165 #include "../std_lib_facilities.h"
1166
1167 class Player {
1168 private:
1169     std::string name;
1170     int health;
1171     int xp;
1172
1173 public:
1174     // This could change the object - we have to qualify it as const. We promise the
1175     // compiler we won't change it
1176     std::string getName() const { return name; }
1177     void setName(std::string name) { this->name = name; }
1178
1179     // Constructors
1180     Player();
1181     Player(std::string name);
1182     Player(std::string name, int health, int xp);
1183     ~Player() { std::cout << "Destructor called for: " << name << std::endl; };
1184 };
1185
1186 Player::Player()
1187 : Player{"None", 0, 0} {
1188     std::cout << "No args constructor called" << std::endl;
1189 }
1190
1191 Player::Player(std::string name)
1192 : Player{ name, 0, 0 } {
1193     std::cout << "Two args constructor for: " << name << " called" << std::endl;
1194 }
1195
1196 // remember that the constructor with the most arguments is the one everyone else
1197 // delegates to
1198 Player::Player(std::string name, int health, int xp)
1199 : name{ name }, health{ health }, xp{ xp } {
1200     std::cout << "Three args constructor for: " << name << " called" << std::endl;
1201 }
1202
1203 // This function with the const qualifier - will fail
1204 void displayPlayerName(const Player &source) {
1205     std::cout << source.getName() << std::endl;
1206 }
1207
1208 int main(){
1209     // villian's attributes cannot be changed
1210     const Player villain{ "Villain", 100, 55 };
1211     Player hero{ "Hero", 100, 15 };
1212
1213     std::cout << villain.getName() << std::endl;
1214     std::cout << "The name is: " << hero.getName() << std::endl;
1215     return 0;
1216 }
1217
1218
1219
1220
1221
1222

```

```

1223 // Static class members
1224 Class members can be declared as static
1225 - The data member belongs to the class, not any specific object
1226 - Useful to store class-wide information
1227
1228 Class functions can also be static
1229 - Independent of any objects
1230 - Can be called using the class name
1231
1232 // Set up the class in the header file
1233 class Player{
1234     private:
1235         static int numberOfPlayers;
1236
1237     public:
1238         static int getNumberOfPlayers();
1239
1240         Player()
1241     }
1242
1243 // Initialize the static member in the .cpp file
1244 int Player::numberOfPlayers = 0;
1245
1246 // Implement the member function as well in the .cpp file
1247 int Player::getNumberOfPlayers(){
1248     return numberOfPlayers;
1249 }
1250
1251 // Update the constructor to add 1 to the static member
1252 Player::Player(std::string name, int health, int xp)
1253     :name{name}, health{health}, xp{xp}{
1254     ++numberOfPlayers;
1255 }
1256 NOTE: Make sure that you only increment in one place - make the other constructors
      delegate!
1257
1258 // Now we need to make our own destructor
1259 Player::~~Player(){
1260     --numberOfPlayers;
1261 }
1262
1263 Now we can call these static member methods
1264 //=====
1265 // Static class members Example program
1266 //=====
1267
1268 // Player.h file
1269 #pragma once
1270 #include <string>
1271
1272 class Player {
1273 private:
1274     std::string name;
1275     int health;
1276     int xp;
1277     static int numberOfPlayers; // Note that this cannot be initialized in here
1278 public:
1279     // Methods
1280     static int getNumberOfPlayers(); // This only has access to other static things
1281
1282     // three arg constructor- delegate to this one, also using defaults
1283     Player(std::string name = "None", int health = 0, int xp = 0);
1284
1285     // Copy constructor
1286     Player(const Player& source);
1287
1288     // Destructor
1289     ~Player();
1290 };

```

```

1291 // Player.cpp file
1292 #include "Player.h"
1293 #include <iostream>
1294
1295 // Initialize the static variable
1296 int Player::numberOfPlayers{ 0 };
1297
1298 // Add to the static member in the constructor
1299 Player::Player(std::string name, int health, int xp)
1300     : name{ name }, health{ health }, xp{ xp } {
1301     ++numberOfPlayers; // Add one to the number of players
1302 }
1303
1304 // Delegate from the copy constructor to the three args constructor
1305 Player::Player(const Player& source)
1306     : Player{source.name, source.health, source.xp} {
1307 }
1308
1309 // Destructor - delete one player when this is called
1310 Player::~~Player() {
1311     std::cout << "Calling destructor for: " << name << std::endl;
1312     --numberOfPlayers; // Make sure that we clean up the players
1313 }
1314
1315 int Player::getNumberOfPlayers() {
1316     return numberOfPlayers;
1317 }
1318
1319 // Main.cpp file
1320 #include <iostream>
1321 #include <vector>
1322 #include <string>
1323 #include <cmath>
1324 #include <ctime>
1325 #include "../std_lib_facilities.h"
1326 #include "Player.h"
1327
1328 void displayActivePlayers() {
1329     std::cout << "Active players: " << Player::getNumberOfPlayers() << std::endl;
1330 }
1331
1332 int main(){
1333     displayActivePlayers();
1334     Player josie{ "Josie", 100, 100 };
1335     displayActivePlayers();
1336     {
1337         Player yaya{ "Yaya", 10,10 };
1338         displayActivePlayers();
1339     }
1340     displayActivePlayers();
1341     Player Bab{}; // Now with the defaults a no args object will work
1342
1343     Player* enemy = new Player{ "Enemy", 1000, 1000 };
1344
1345     displayActivePlayers();
1346     delete enemy;
1347
1348     displayActivePlayers();
1349     return 0;
1350 }
1351
1352 // Structs versus classes
1353 Structs are from C
1354 Essentially the same as classes, but public by default
1355
1356 struct Person{
1357     std::string name;
1358     std::string getName(); // this is public by default
1359 }

```

```

1360
1361 // When to use which?
1362 struct:
1363 - Use for passive objects with public access
1364 - don't declare methods in a struct
1365
1366 class:
1367 - Use for active objects with private access
1368 - Implement getters and setters as needed
1369 - Implement member methods as needed
1370
1371 // Friends of a class
1372 Friend:
1373 - A function or class that has access to private class members
1374 - that function or class is not a member of the class it is accessing
1375
1376 Function:
1377 - Can be regular non-member functions
1378 - Can be member methods of another class
1379
1380 Class:
1381 - Another class can have access to private class members
1382
1383 // Controversy
1384 Do friends enhance or detract from encapsulation
1385
1386 // The big picture:
1387 Friendship must be granted. It cannot be taken
1388 - Declared explicitly in the class that is granting it
1389 - Declared in the function prototype with the keyword friend
1390
1391 Friendship is not symmetric
1392 - Must be explicitly granted
1393 - A can be a friend of B, but that doesn't necessarily mean B is a friend of A
1394
1395 Friendship is also not transitive
1396 - A is a friend of B
1397 - B is a friend of C
1398 - A is not automatically a friend of C
1399
1400 Friendship is also not inherited
1401
1402 // Examples
1403 class Player{
1404     // This stuff defaults to private
1405     friend void displayPlayer(Player &source);
1406
1407     public:
1408     // stuff here
1409 }
1410
1411 That function can now directly access the private attributes
1412 void displayPlayer(){
1413     std::cout << p.name << std::endl;
1414 } //It can change private data members as well.
1415
1416 // Declaring the method of another class as a friend:
1417 Player{
1418     friend void OtherClass::displayPlayer(Player &source);
1419 }
1420
1421 Now the displayPlayer method in the other class can access things in this class
1422
1423 // Declare an entire class as a friend
1424 Class Player{
1425     friend class OtherClass;
1426 }

```