**Classes and Scoped Enumerations – continued**

Class Members

A class can include data members that hold the same information for all instances of the class and access that information regardless of the number of instances that currently exist.  These are called class variables.  Class functions can access these variables at any time.

Class Variables

A *class variable* lasts the lifetime of the program and holds a value that all instances of the class share.  Application examples include the interest on a savings account for customers of a non-discriminating bank or counters that keep track of the number of objects instantiated but not yet destroyed.

The keyword static declares a variable in a class definition to be a class variable.  We define and initialize the class variable in the implementation file for the module.  For example,

|  |  |
| --- | --- |
| // Class Variables - Header  // classVariable.h  class Horse {  unsigned age;  unsigned no;  public:  static unsigned noHorses;  Horse(unsigned a);  void display() const;  ~Horse();  }; | 0 horses  Horse 1 is 3 years old  Horse 2 is 4 years old  2 horses  2 horses  2 horses  0 horses |
| // Class Variables - Implementation  // classVariable.cpp  #include <iostream>  #include "classVariable.h"  unsigned Horse::noHorses = 0; // initializer  Horse::Horse(unsigned a) :  age{a}, no{++noHorses} {}  void Horse::display() const {  std::cout << "Horse " << no <<  " is " << age << " years old\n";  }  Horse::~Horse() { noHorses--; } |
| // Class Variables - Application  // classVariableMain.cpp  #include <iostream>  #include "classVariable.h"  int main() {  std::cout << Horse::noHorses << " horses\n";  {  Horse silver(3), northernDancer(4);  silver.display();  northernDancer.display();  std::cout << Horse::noHorses  << " horses" << std::endl;  std::cout << silver.noHorses  << " horses" << std::endl;  std::cout << northernDancer.noHorses  << " horses" << std::endl;  }  std::cout << Horse::noHorses << " horses\n";  } |

We can refer to a class variable through its class name or through anyone of its objects' names.  We can access a class variable even if there are no instances of the class at the time.

Class Functions

A class function provides access to private class variables.  The keyword static declares a function in a class definition to be a class function.  We define the class function in the implementation file.  For example,

|  |  |
| --- | --- |
| // Class Functions - Header  // classFunction.h  class Horse {  unsigned age;  unsigned no;  static unsigned noHorses;  public:  Horse(unsigned a);  void display() const;  ~Horse();  static unsigned howMany();  }; | 0 horses  Horse 1 is 3 years old  Horse 2 is 4 years old  2 horses  2 horses  2 horses  0 horses |
| // Class Functions - Implementation  // classFunction.cpp  #include <iostream>  #include "classFunction.h"  unsigned Horse::noHorses = 0;  Horse::Horse(unsigned a) : age{a}, no{++noHorses} {}  void Horse::display() const {  std::cout << "Horse " << no <<  " is " << age << " years old\n";  }  Horse::~Horse() { noHorses--; }  unsigned Horse::howMany() { return noHorses; } |
| // Class Functions - Application  // classFunctionMain.cpp  #include <iostream>  #include "classFunction.h"  int main() {  std::cout << Horse::howMany() << " horses\n";  {  Horse silver(3), northernDancer(4);  silver.display();  northernDancer.display();  std::cout << Horse::howMany() << " horses"  << std::endl;  std::cout << silver.howMany() << " horses"  << std::endl;  std::cout << northernDancer.howMany()  << " horses" << std::endl;  }  std::cout << Horse::howMany() << " horses\n";  } |

All data members of class scope to which a class function refers must be class variables.  A class function may not refer to any instance variable.

Structs and Unions

A struct or union is a class that is weakly encapsulated.  Its members are public by default.  Its members, like those of a class may be of different types.

Structs

The members of a struct are arranged sequentially but not necessarily contiguously in memory.

Unions

The members of a union are arranged in parallel in memory.  Unlike the members in a class or a struct, the members in a union share the same address in memory

An object of union type can only hold the value of one of its members at any particular time.  For example,

|  |  |
| --- | --- |
| // Unions  // union.cpp  #include <iostream>  #include <cstring>  union Product { // some have skus, some have upcs  int sku;  char upc[13];  };  int main() {  Product cereal, tissue;  cereal.sku = 4789;  std::strcpy(tissue.upc, "0360002607555");  std::cout << cereal.sku << std::endl;  std::cout << tissue.upc << std::endl;  } | 4789  0360002607555 |

Note that only the value of the member that was most recently assigned is stored in memory.

Enumerations

An enumeration is a type that holds a discrete set of symbolic constants.  These constants can simplify the readability and the upgradability of an application significantly.

Each enumeration definition declares a type that is different from all other types and has an underlying type.  The default underlying type of an enumeration in an int.

Plain Enumerations

The definition of a plain enumeration type lists the symbolic constants in the following form

|  |
| --- |
| enum *Type* { *symbolic\_constant\_1*, *symbolic\_constant\_2*, *symbolic\_constant\_3*, ... }; |

The enum distinguishes the enumerated type from any other type. Type is the name of the enumerated type.  Each symbolic constant in the list is an *enumeration constant*.

We define a variable of enumerator type using a declaration of the form

|  |
| --- |
| *Type identifier*; |

*identifier* is the name of the variable that holds one of the symbolic constants of type *Type*.

For example,

|  |  |
| --- | --- |
| // Plain Enumerations  // plain\_enum.cpp  #include <iostream>  // define the Colour enumeration type  enum Colour {white, red, green, blue};  std::ostream& operator<<(std::ostream& os,  const Colour& colour) {  std::string str;  switch(colour) {  case white: // symbolic constant  str = "white";  break;  case red: // symbolic constant  str = "red";  break;  case green: // symbolic constant  str = "green";  break;  case blue: // symbolic constant  str = "blue";  break;  default:  str = "none";  }  os << str.c\_str();  return os;  }  int main() {  Colour wall, ceiling, door; // define Colour variables  wall = red;  ceiling = white;  door = green;  std::cout << wall << ' '  << ceiling << ' ' << door << std::endl;  } | red white green |

Tracking Access in Unions usings Structs and enum

There is no independent way of identifying a union's most recently accessed member.  To address this ambiguity, we wrap a union type within a struct type and declare an enumerator as a member of the struct type.  The enumerator identifies the most recently accessed member within the union type.  For example,

|  |  |
| --- | --- |
| // Structs with Unions  // struct\_union.cpp  #include <iostream>  #include <cstring>  enum ProductId {sku, upc};  typedef struct {  ProductId id; // enumeration type holds a symbol  union { // some have skus, some upcs  int sku;  char upc[13];  } code;  } Product;  int main() {  Product p[2];  p[0].id = sku;  p[0].code.sku = 4789;  p[1].id = upc;  std::strcpy(p[1].code.upc, "0360002607555");  for (int i = 0; i < 2; i++)  switch(p[i].id) {  case sku:  std::cout << p[i].code.sku << std::endl;  break;  case upc:  std::cout << p[i].code.upc << std::endl;  break;  }  } | 4789  0360002607555 |

Note the type definitions and the two anonymous types in this example.

Scoped Enumerations

A scoped enumeration is an enumeration that restricts access to its symbolic constant based on its scope.  Symbolic constants with the same name but different scopes are distinct from one another.

The definition of a scoped enumeration type lists the symbolic constants and takes the following form.  The keyword class identifies a scoped enumeration

|  |
| --- |
| enum class *Type* {  *symbolic\_constant\_1*,  *symbolic\_constant\_2*,  *symbolic\_constant\_3*, ...  }; |

Type is the name of the enumerated type.  Each symbolic constant in the list is an *enumeration constant*.

We define a variable of enumerator type using a declaration of the form

|  |
| --- |
| *Type identifier*; |

*identifier* is the name of the variable that will hold one of the symbolic constants of type *Type*.

For example,

|  |  |
| --- | --- |
| // Scoped Enumerations  // scoped\_enum.cpp  #include <iostream>  // define the Colour type  enum class Colour {white, red, green, blue};  std::ostream& operator<<(std::ostream& os,  const Colour& colour) {  std::string str;  switch(colour) {  case Colour::white:  str = "white";  break;  case Colour::red:  str = "red";  break;  case Colour::green:  str = "green";  break;  case Colour::blue:  str = "blue";  break;  default:  str = "none";  }  os << str.c\_str();  return os;  }  int main() {  Colour wall, ceiling, door; // define Colour variables  wall = Colour::red;  ceiling = Colour::white;  door = Colour::green;  std::cout << wall << ' '  << ceiling << ' ' << door << std::endl;  } | red white green |

Underlying Types

The compiler associates each symbolic constant of an enumeration with a unique value of the enumeration's underlying type.  By default, the compiler assigns the value 0 to the first constant in the list and a value 1 greater than the preceding value to each successive constant.

We may assign our own value to an enumeration constant provided that each value is unique within the list for that type.  For example,

|  |
| --- |
| enum class Colour {  white = 0x01,  red = 0x02,  green = 0x04,  blue = 0x08  }; |

Note that the value of red is 2, which is 1 greater than white, but the value of blue is 8, which is 4 greater than green.

Modifying an Enumeration

Enumeration syntax simplifies upgradability.  If we insert a new symbolic constant into the enumeration, the compiler renumbers the underlying types for the subsequent constants in the list without requiring any further modification:

|  |  |
| --- | --- |
| // Inserting Enumeration Constants  // insert\_enum.cpp  #include <iostream>  // define the Colour type  enum class Colour {white, yellow, red, green, blue};  std::ostream& operator<<(std::ostream& os,  const Colour& colour) {  std::string str;  switch(colour) {  case Colour::white:  str = "white";  break;  case Colour::yellow:  str = "yellow";  break;  case Colour::red:  str = "red";  break;  case Colour::green:  str = "green";  break;  case Colour::blue:  str = "blue";  break;  default:  str = "none";  }  os << str.c\_str();  return os;  }  int main() {  Colour wall, ceiling, door, window;  wall = Colour::red;  ceiling = Colour::white;  door = Colour::green;  window = Colour::yellow;  std::cout << wall << ' ' window << ' '  << ceiling << ' ' << door << std::endl;  } | red yellow white green |

Note that the only changes required for inserting the new symbolic constant yellow were adding it to the enumeration definition and adding a corresponding case to the switch-case construct.