```
/*read file
std::ifstream file(filename);
if (!file){
   std::cerr << "error";
   exit(AppErrors::CannotOpenFile);
}
std::string strToy;
do{
   std::getline(file, strToy);
   if (file){
      if (object[0] != '#') {
         ppToy[count] = new sdds::Toy(strToy);
         ++count;
      }
   }
} while (file);
file.close();*/
```

```
/* Resize */
ConfirmOrder& operator+=(const Toy& toy) {
   const Toy** temp = nullptr;
   temp = new const Toy * *[m_count + 1];
   for (size_t i = 0u; i < m_count; i++){
      temp[i] = m_toys[i];
      temp[m_count] = &toy;
   }
   delete[] m_toys;
   m_toys = nullptr;
   m_toys = temp;
   m_count++;
}
```

```
/* Composition - Toy object is created as the Child object is created
   and destroyed as the Child object is destroyed.
Aggragation - don't need to create a Toy object in constructor*/
class Child {
   //other member variables
   const sdds::Toy** m_toyArrPtr = nullptr;
   size_t m_count = 0u;
public:
   Child(const Toy** toys, size_t count) {
      m_count = count;
      m_toyArrPtr = new const Toy * [m_count];
      for (size_t i = 0u; i < count; i++){
         m_toyArrPtr[i] = new Toy(*toys[i]);
      }
   }
   /* ConfirmOrder::ConfirmOrder():m_count(0){m_toyArrPtr=nullptr;}*/
   Child(const Child& c) { *this = c; }
   Child& operator=(const Child& c) {
      if (this!=&c){
         for (size_t i = 0u; i < m_count; i++) {delete m_toyArrPtr[i];}
         delete[] m_toyArrPtr;
         m_toyArrPtr = nullptr;
         m_count = c.m_count;
         m_toyArrPtr = new const Toy * [m_count];
         for (size_t i = 0u; i < m_count; i++) {
            m_toyArrPtr[i] = new Toy(*c.m_toyArrPtr[i]);
         }
      }
```

```
'*ConfirmOrder& operator=(const ConfirmOrder& co){
   if (this!=&co){
      delete[] m_toys;
      m_toys = nullptr;
      m_count = co.m_count;
      m_toys = new const Toy * [m_count];
      for (size_t i = 0; i < m_count; i++) {
         m_toys[i] =co.m_toys[i];
      }
   }
   return *this;
*/
hild(Child&& c) {// Move constructor
   *this = std::move(c);

hild& operator=(Child&& c) { //Move assignment
   if (this!=&c){
      m_count = c.m_count;
      delete[] m_toyArrPtr;
      m_toyArrPtr = nullptr;
      m_toyArrPtr = c.m_toyArrPtr;
      c.m_count = 0;
      c.m_toyArrPtr = nullptr;
   }
   return *this;
```

```
/*lambda
   auto add4 = [](int i) {return i + 4;};//example 1
   cout << add4(10);
   [](char ch) { //example 2
      for (size_t i = 0; i < 10; i++)cout << ch;
      cout << endl;
   }('=');
   int a = 4; // //example 3 capture by reference
   auto addI2A = [&](int i) {return a += i; };
   addI2A(10); // 10+4
   template <typename T> // lambda passing to Template
   int add(int i, T func) {return func(i);}
   int main() {
   int k=4;
   auto lambda = [&](int i) {return i + k; };
   cout << add(10, lambda); // or
   cout << add(10, [&](int i) {return  i + k; });
}*/
```

```
virtual ~Child() {
   for (size_t i = 0u; i < m_count; i++) { delete m_toyArrPtr[i]; }
   delete[] m_toyArrPtr;
}
/*ConfirmOrder::~ConfirmOrder(){delete[] m_toys;}*/
friend std::ostream& operator<<(std::ostream& ostr, const Child& c);
};
/ friend insertion operator -or
td::ostream& operator<<(std::ostream& ostr, const Child& c) {
   ostr << std::setprecision(2); // <iomanip>
   ostr << std::setiosflags(std::ios::right);
   ostr << std::resetiosflags(std::ios::right);
```

```
/*read text to attribute one by one*/
Toy(const std::string& toy) {
   size_t posB = 0u;
   size_t posE = toy.find(':'); // <string>
   std::string temp = toy.substr(posB, posE - posB);
   temp.erase(0, toy.find_first_not_of(" ")); // remove heading spaces
   temp.erase(toy.find_last_not_of(" ") + 1); // remove tailing spaces
}
```

2. If an expression is a/an xvalue decltype(expression) evaluates to an rvalue reference
3. The operands associated with the following operators must be lvalues: & ++
4. Select the constrained cast that converts a pointer of integral type to a pointer of another integral type
   reinterpret_cast
5. Expressions based on the following operators evaluate to prvalues: & - address of postfix !
6. Copying a polymorphic object requires a different function for each dynamic type
7. An rvalue reference to an object or a function can be a non-type template parameter false
8. Just like an object of class X cannot be an instance variable of class X, a template cannot be a template parameter False
9. We should model our classes in an inheritance hierarchy on their behaviors_

The C++ keyword that distinguishes a plain enumeration from a scoped enumeration is **class**.

classes are **strong** encapsulated.

The template argument for an integral non-type parameter can be any variable or expression of that type. (**False**)

The Liskov Substitution Principle provides guidance on the design of the instance variables in a concrete classes. (**False**)

A function that is noexcept and calls another function that can throws an exception will always terminate immediately. (**False**)

The code in a try block always executes completely if the application returns through a normal exit. (**False**)

an std::exception should be caught before any other exception. (**False**)

C++ templates implement **parametric polymorphism**.

If an expression is a/an **xvalue** decltype(expression) evaluates to an rvalue reference.

The function that destroys objects with static storage duration and flushes and closes all open streams is **void exit(int)**.

```
/*funcPtr
void add(int a, int b) {implementation }
void (*funcPtr)(int, int) = add;
int main(){funcPtr(10, 2);}
*/

/*funcPtrArray
void add(int a, int b) {implementation }
void sub(int a, int b) {implementation }
void mul(int a, int b) {implementation }
int main(){void (*funcPtr[3])(int, int) = { add,sub,mul,divid };
  for (auto i = 0u; i < 3; i++){func[i] = (10, 2);}}
*/

/*funcPtrTemplate
template <typename T>
bool ascending(T a, T b) {return a > b;}
template <typename T>
bool decending(T a, T b) { return a < b; }
template <typename T>
void sort(T* arr, int arrSize, bool(*funcPtr)(T, T)) {
  if (funcPtr(arr[i],arr[j])) { implementation };}
int main() {
  int a[] = {1,5,3,6,7,2};
  sort(a, 6, ascending<int>);
  sort(a, 6, decending<int>);
}*/
```

```
/*exception
double d = 4;
char str[] = "hoohoo";
for (i = 0; i < 3; i++) {
  try {
     if (i == 0) {throw d;}
     if (i == 1) {throw str;}}
  catch (double de) {cout << "Double: " << de << endl;}
  catch (const char* se) {cout << "Stirng: " << se << endl;}
*/
```

```
/*Templated functions*/
template<typename Atype, typename Btype>
auto add(const Atype& a, const Btype& b)->decltype(a + b) { return a + b; }
// the compiler doesn't know the return type when it starts processing the definition
```

```
/*interface--only head file*/
class Interface {
public://no constructor, an abstract class can not be instantiated.
  virtual void function(int a) = 0;
  virtual ~Interface();
};
```

```
/*functor
enum class Order { // To define enum class type
  ascending, descending};
class Compare { // functor
  Order m_order;
public:
  Compare(Order o) :m_order(o) {};
  bool operator()(int& a, int& b)const {
     return m_order == Order::ascending ? a > b:a < b;// using enum class type}};
void sort(int* arr, int arrSize, const Compare& comp) {
  if (comp(arr[i],arr[j])) { implementation };}
int main() {
  int a[] = { 1, 5, 2, 3, 6, 7, 2 };
  int arrSize = sizeof a / sizeof(int);
  sort(a, arrSize, Compare(Order::ascending));
  sort(a, arrSize, Compare(Order::descending));}}*/
```

```
/*template class
template <typename T,unsigned int capacity>
class Collection {
  T m_arr[capacity];
  unsigned int m_noOfElements{};
  static T m_dummy;
public:
  //member functions;
};
// initialization of m_dummy
template <typename T, unsigned int capacity>
T Collection<T, capacity>::m_dummy{};
template<> // specilization
Pair Collection<Pair, 100>::m_dummy("No Key", "No Value");
*/
```