* Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. If an exception is thrown and not caught anywhere, the program terminates abnormally.
* There are two types of exceptions:

1. Synchronous: Classical C++ exceptions, thrown whenever the user calls a function with wrong arguments. These can be caught with a try..catch block.
2. Asynchronous: Asynchronous errors, on the other hand, are those that describe faults in asynchronously executed code, for example inside a command group or a kernel. By default, they are considered "lost". The way in which we can retrieve them is by providing an error handler function.

* Different components of Exception Handling:

1. try: define a block of code to be tested for errors while it is being executed.
2. catch: allows you to define a block of code to be executed, if an error occurs in the try block.
3. throw: Used to throw an exception when a problem is detected, which lets us create a custom error. Also used to list the exceptions that a function throws, but doesn’t handle itself.
4. catch all: special catch block used to catch all types of exceptions. catch(…)

* void fun(int \*ptr, int x) signature is fine by the compiler, but not recommended. Ideally, the function should specify all uncaught exceptions and function signature should be "void fun(int \*ptr, int x) throw (int \*, int)".
* Code within a try/ctch block is known as protected code.
* C++ library has a standard exception class which is base class for all standard exceptions. All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type. Hence A derived class exception should be caught before a base class exception.
* In C++, all exceptions are unchecked. Compiler doesn’t check whether an exception is caught or not.
* Pseudo Code:

try {

if (condition)

{

throw exception/variable;

}

}

catch (data\_type variable) { \\normal catch

}

catch (...) { \\catch all exceptions

}

}

Example:

Normal Catch:

try {

int age = 15;

if (age >= 18) {

cout << "Access granted - you are old enough.";

} else {

throw (age);

}

}

catch (int myNum) {

cout << "Access denied - You must be at least 18 years old.\n";

cout << "Age is: " << myNum;

}

Catch All:

try {

int age = 15;

if (age >= 18) {

cout << "Access granted - you are old enough.";

} else {

throw 505;

}

}

catch (...) {

cout << "Access denied - You must be at least 18 years old.\n";

}

Explained:

1. We use the try block to test if the age variable is less than 18 or not.
2. If age is less than 18 we will throw an exception, and handle it in our catch block.
3. In the catch block, we catch the error and do something about it.
4. The catch statement takes a parameter an int variable (myNum) to output the value of age in code written in catch.
5. If no error occurs, e.g. if age is 20 instead of 15, it is greater than 18, the catch block is skipped.

C++ Standard Exceptions

1. std::exception: Parent class of all the standard C++ exceptions.
2. std::bad\_alloc: Type of the object thrown as exceptions by the allocation functions to report failure to allocate storage. This can be thrown by new.
3. std::bad\_cast: This can be thrown by dynamic\_cast to a reference type fails the run-time check (e.g. because the types are not related by inheritance), and also from std::use\_facet if the requested facet does not exist in the locale.
4. std::bad\_exception: This is useful device to handle unexpected exceptions thrown by the C++ runtime in the following situations:
5. If std::exception\_ptr stores a copy of the caught exception and if the copy constructor of the exception object caught by std::current\_exception throws an exception, the captured exception is an instance of std::bad\_exception.
6. If a dynamic exception specification is violated and std::unexpected throws or rethrows an exception that still violates the exception specification, but the exception specification allows std::bad\_exception, std::bad\_exception is thrown.
7. std::bad\_typeid: This can be thrown by typeid, when a typeid operator is applied to a dereferenced null pointer value of a polymorphic type.
8. std::logic\_error: An exception that theoretically can be detected by reading the code. Defines a type of object to be thrown as exception. It reports errors that are a consequence of faulty logic within the program such as violating logical preconditions or class invariants and may be preventable. No standard library components throw this exception directly, but the exception types are derived from std::logic\_error:
9. std::domain\_error: This is an exception thrown when a mathematically invalid domain is used, that is, situations where the inputs are outside of the domain on which an operation is defined.
10. std::invalid\_argument: It reports errors that arise because an argument value has not been accepted.
11. std::length\_error: This is thrown when a too big std::string is created. It reports errors that result from attempts to exceed implementation defined length limits for some object.
12. std::out\_of\_range: This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().
13. std::future\_error
14. std::experimental::bad\_optional\_access
15. std::runtime\_error: An exception that theoretically cannot be detected by reading the code. Defines a type of object to be thrown as exception. It reports errors that are due to events beyond the scope of the program and can not be easily predicted. Exceptions of type std::runtime\_error are thrown by the following standard library components: std::locale::locale and std::locale::combine.
16. std::overflow\_error: This is thrown if a mathematical overflow occurs. It can be used to report arithmetic overflow errors (that is, situations where a result of a computation is too large for the destination type)
17. std::range\_error: This is occurred when you try to store a value which is out of range. It can be used to report range errors (that is, situations where a result of a computation cannot be represented by the destination type).
18. std::underflow\_error: This is thrown if a mathematical underflow occurs. It may be used to report arithmetic underflow errors (that is, situations where the result of a computation is a subnormal floating-point value)