This chapter introduces the notion of dynamic memory allocation of variables and objects in a C++ program. A second section describes how C++ handle errors, called exception, such as failing to allocate memory dynamically.

1. Dynamic memory allocation in C++

The C++ programming language, like C, allows to store data dynamically. Unlike C that uses the function malloc, calloc, realloc and free to store data dynamically on the Heap, in C++ dynamical memory is allocated on a region called Free Store (may be referred as Heap) with the operators new and delete. We note that the C operators (malloc, calloc, realloc and free) are still available in C++ via the library cstdlib. Although the operator new does not allow to reallocate memory, like realloc does with malloc/calloc, it is advised to use the operators new and delete for the following reasons:

- on failure, new throws an exception that by default stops the code. We remind that malloc and calloc return a NULL pointer on failure. We refer to the next section for more information on exception.
- new allows to initialize a pointer to a variable (like int, double).
- new calls the default constructor of the class when allocating memory for an object of this class. Possibility to call a user defined constructor.
- delete calls the destructor of the class when deallocating the dynamic memory associated to an object of this class.

1.1. The operators new and delete. The operator new can be used to allocate dynamic memory to store a pointer of a variable of type T as follows.

```cpp
T* pt=new T;
```

The memory can then be deallocated with the operator delete.

```cpp
delete T;
```

It is possible to initialize the value pointed by a pointer defined with new as follows.

```cpp
int* pt_i=new int(4); // pt_i is 4
double* pt_d=new double(-3.2); // pt_d is -3.2
delete pt_i; delete pt_d;
```

The operator new can also be used to allocate dynamic memory for a pointer to an object of a class A. In that case, the default constructor of A is called during the creation of the pointer.
1.2. The operators new[] and delete[]. The allocation of an array of variables (int, double, pointer) or objects of a class is done using the operator new[] as follows:

```cpp
int* pt = new int[4]; // array of 4 integers
```

The deallocation of the dynamic memory is done using the operator delete[] as follows:

```cpp
delete[] pt;
```

It is important to note that mixing the operator new with delete[] or new[] with delete generate undefined behavior. So you should always use delete with variable/object that are allocated with new. Reciprocally always use delete[] for array of variables/ojbjects that are allocated with new[].

Unlike pointer of a single variable or object, it is not possible to call a user constructor when allocating dynamic memory for an array of variables or objects. Thus, the following instructions would generate a compilation error.

```cpp
int* pt = new int[5](4); // bad try to set 5 integers to 4
```

However it is possible to set the element of an array of integer, float or double to zero by adding () after the call to new[] as follows:

```cpp
int* pt = new int[5](); //5 integers set to zero.
```

One could also call a user constructor when defining an array of 4 objects of the class A as follows:

```cpp
A** pt = new A*[4];
for (int i=0; i<4;i++){
    pt[i]=new A(arg_user_constructor);
}
```

We refer to the example ex39_dynamic_memory_C++.cc for more information.

1.3. Multidimensional array. The allocation of a multidimensional array with dynamic memory is done similarly as in C but with the operator new and delete. Let A be a pointer of pointer of double that is representing a two dimensional matrix of dimension (M,N). The memory associated to A can be allocated either contiguously as follows:

```cpp
double** A=new double*[M];
A[0]=new double[M*N];
for (int i=1; i<M; i++){
}
```

or discontiguously
double ** A= new double *[M];
for ( int i=0; i<M; i++){
    A[i]= new double [N];
}

where the matrix is store using a row major index such that the element (i,j) of the matrix is represented by A[i][j]. Note that each call to the operator new[] has to be followed by a call to delete[] later in the program. As a consequence, the first method (contiguous storage) only requires two call of delete[]:

delete [] A[0];
delete [] A;

while the second one requires a loop:

for ( int i=0; i<M; i++){
    delete [] A[i];
}
delete [] A;

2. Exception: error handling in C++

A C++ exception is a way to answer to a problem that occurs during the execution of a program. They introduce the notion of exception handlers that allow to execute a specific set of instructions when encountering a problem. These handlers consist of three keyword throw, try and catch that respectively:

- throw: throw an exception when a problem occurs.
- try: encapsulate a block of code called try block. During execution, the program checks if an exception is thrown in the try block with the keyword throw.
- catch: create a catch block that must be placed just after a try block. Allow the programmer to implement instructions that are executed when an exception is thrown in the try block. These instructions usually consists of printing a description of the error encountered and stopping the program.

The following describes the process of throwing and catching an exception. We introduce some standard exception provided by C++ and show how to define user exception that return a description of the exception and stop the program.

2.1. Throwing/catching exception. A throw expression is associated to one parameter. The type of the parameter defines the type of the exception that is thrown. For instance, the following instructions throw exceptions of type "int" and "const char*".

throw 12; //exception of type int
throw "exception"; //exception of type const char*

The type of the exception is then used in the catch block to execute instructions associated to the type of exception thrown as follows:
#include <iostream>
int main(void){
  try{
    // code to try
    throw 13; // throw exception
  }
  catch(int e){ // catch throw of integer
    std::cout << "Exception encountered: " << e << std::endl;
    return 1; // stop program
  }
  return 0; // no exception encountered
}

where the variable e in the catch block represent the parameters that is associated to throw. Meaning the above code print "Exception encountered: 13" when executed.

It is possible to add multiple catch block after a try block as thrown exceptions may have different types. A program often contains one try block that encapsulate the instructions of the main function. It allows to catch a call to throw anywhere in the program. Here is a generic way to catch all exceptions and stop the program if an exception happens.

int main(void){
  try{
    // instructions
  }
  catch(...){ // catch all exceptions
    std::cout << "Exception encountered." << std::endl;
    return 1; // stop program
  }
  return 0; // no exception encountered
}

Remark: Throwing an exception does not force the program to stop. If the associated catch block does not stop the program, the program then read the instructions that follow the try-catch blocks. Note that when a throw is called, the instructions that follows in the try block are never executed. If an exception is thrown but its type is not caught by a catch block, the execution of the program is aborted.

2.2. Standard C++ exceptions. The C++ programming language provides a set of standard C++ exceptions that are always thrown if encountered. These standard exceptions are implemented as derived classes of the class std::exception that is defined in the library exception. Here is a few examples of standard exception:

  - std::bad_alloc. Thrown by operator new when memory allocation fails.
  - std::range_error. Can be thrown when a variable should be used to represent another one but there is a conflict in their declarations.
  - std::invalid_argument. Thrown when an argument is not accepted.
Each standard exception has a member function "what" that contains information on the exception they represent. A reference to a standard exception is compatible with the reference to the exception of type std::exception. As a consequence, they can all be caught with the following instructions.

```cpp
#include <iostream>
#include <exception>
int main (void){
    try{
        // Uncomment to test
        // throw std::bad_alloc();
    }
    catch (std::exception& e){ // catch standard exceptions
        std::cout << "Exception: " << e.what() << std::endl;
        return 1; // stop program
    }
    return 0; // no exception encountered
}
```

where we use the function what to display information on the caught exception. If a standard exception is not caught by a try-catch block, the program is stopped by default. All these standard exceptions can be caught with the catch block of the above example as they are derived class of the class std::exception. We refer to the literature for more information on the different standard exception provided by C++.

2.3. **Throwing a simple user exception.** A program may present exceptions that are not part of the standard C++ exceptions such as a division by zero or inverse a non invertible matrix. A way to handle these user exceptions is to call throw with a set of characters that describes the problem. The type of the thrown exception is then "const char". It can be caught with a try-catch block as follows:

```cpp
#include <iostream>
int main (void){
    try{
        // instructions
        throw "Something bad happened";
    }
    catch (const char* e){
        std::cout << e << std::endl;
        return 1;
    }
    return 0;
}
```

where the catch block displays the set of characters that is given to throw and stops the code. Note that a throw is usually called in a if statement to set a condition when the program should throw an exception. We refer to the class and the example ex40_exception.cc for more information.
2.4. **Create a user exception derived from the base class std::exception.** The class std::exception of the library exception can be used as a base class to create new exception. This class contains a virtual function "what" that returns a sequence of character (const char*). This virtual function can be redefined in a derived class to provide a description of the new exception. The following block of code shows how to create a new type of exception in a header file called my_exception.hh.

```cpp
#ifndef MY_EXCEPTION_HH
#define MY_EXCEPTION_HH
#include <iostream>
#include <exception>

class my_exception : public std::exception{
    const char* what() const throw() {
        return "my_exception";
    }
};
#endif
```

The function main can then be written as follows to catch an exception of the type my_exception.

```cpp
#include "my_exception.hh"
int main(void){
    try{
        throw my_exception;
    }
    catch(my_exception& e){
        std::cout << "Exception: " << e.what() << std::endl;
        return 1;
    }
    return 0;
}
```

2.5. **Note on dynamic memory and standard exception.** By default the operator new throw an exception bad_alloc if it fails to allocate the memory requested. However it is possible to tell the operator new to not throw an exception and return a NULL pointer as follows:

```cpp
int* a = new(nothrow) int[100];
```

This feature requires to include the standard library new even though the operator new, new[], delete and delete[] are accessible by default. A programmer should be careful when using this nothrow version of new as dereferencing a NULL pointers is undefined behavior.