# CS-202

## Recapitulation (Pt.2)

**C. Papachristos**
Autonomous Robots Lab
University of Nevada, Reno

# Course Week

## Course , Projects , Labs:

| Monday | Tuesday | Wednesday | Thursday | Friday | | Sunday |
|---|---|---|---|---|---|---|
| | | | | | | |
| | **RECAP** CLASS | | **FINAL** | | | |
| PASS Session | PASS Session | | | | | |

Your **Final** is on Thursday 5/9 @ 4:30pm.

# Today's Topics

Recapitulation:

- Dynamic Data Structures
- Templates
- Exceptions

Prerequisites (not covered in Recap):

- Pass-by-Value

```
void func( DataType obj );
```

- Pass-by-Reference – Pass-by-const-Reference

```
void func( DataType & obj ); / void func( const DataType & obj );
```

- Pass-by-Address(Pointer) – Pass-by-const-Address(Pointer)

```
void func( DataType * obj ); / void func( const DataType * obj );
```

- Return-a-Value

```
DataType func( );
```

- Return-a-Reference – Return-a-const-Reference

```
DataType & func( ); / const DataType& func( );
```

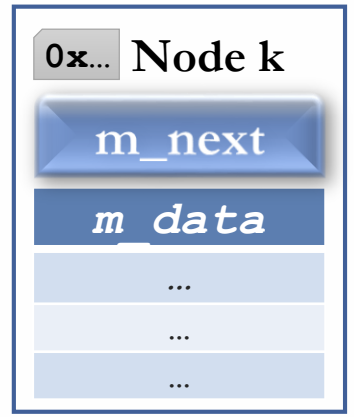
- Return-an-Address(Pointer) – Return-a-const-Address(Pointer)

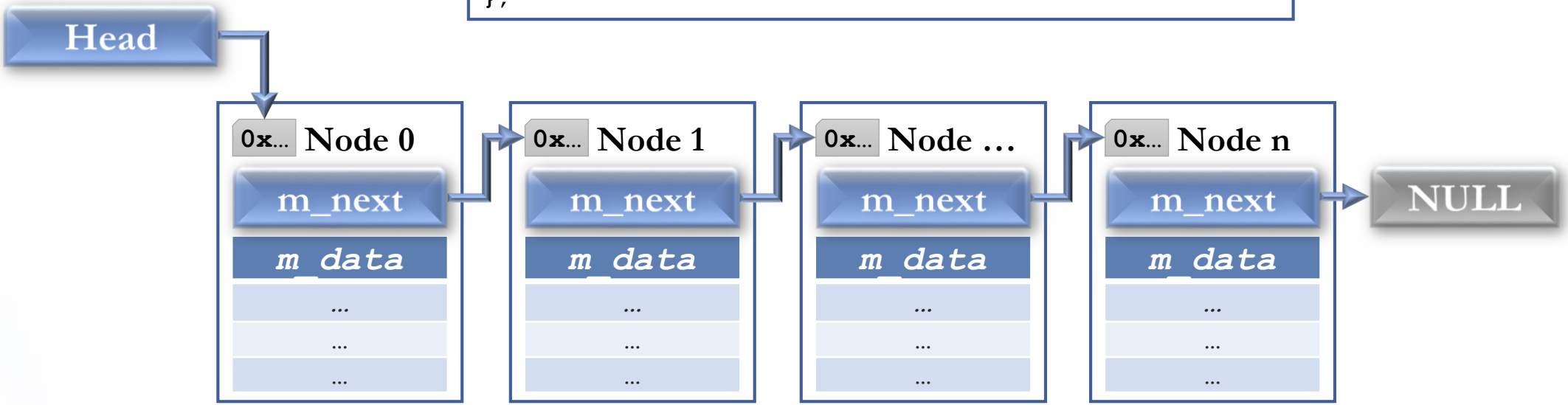
```
DataType * func( ); / const DataType * func( );
```

# Dynamic Data Structures

## Linked-List(s)

```
class Node{
public:
  Node() : m_next(NULL){ }
  Node(const DataClass & data, Node * next = NULL)
     : m_data(data), m_next(next){ }
  const DataClass & data() const{ return m_data; }
  DataClass & data(){ return m_data; }
//declaration of friend classes - Queue,Stack,List,etc.
friend class List;
private:
  Node * m_next;
  DataClass m_data;
};
```

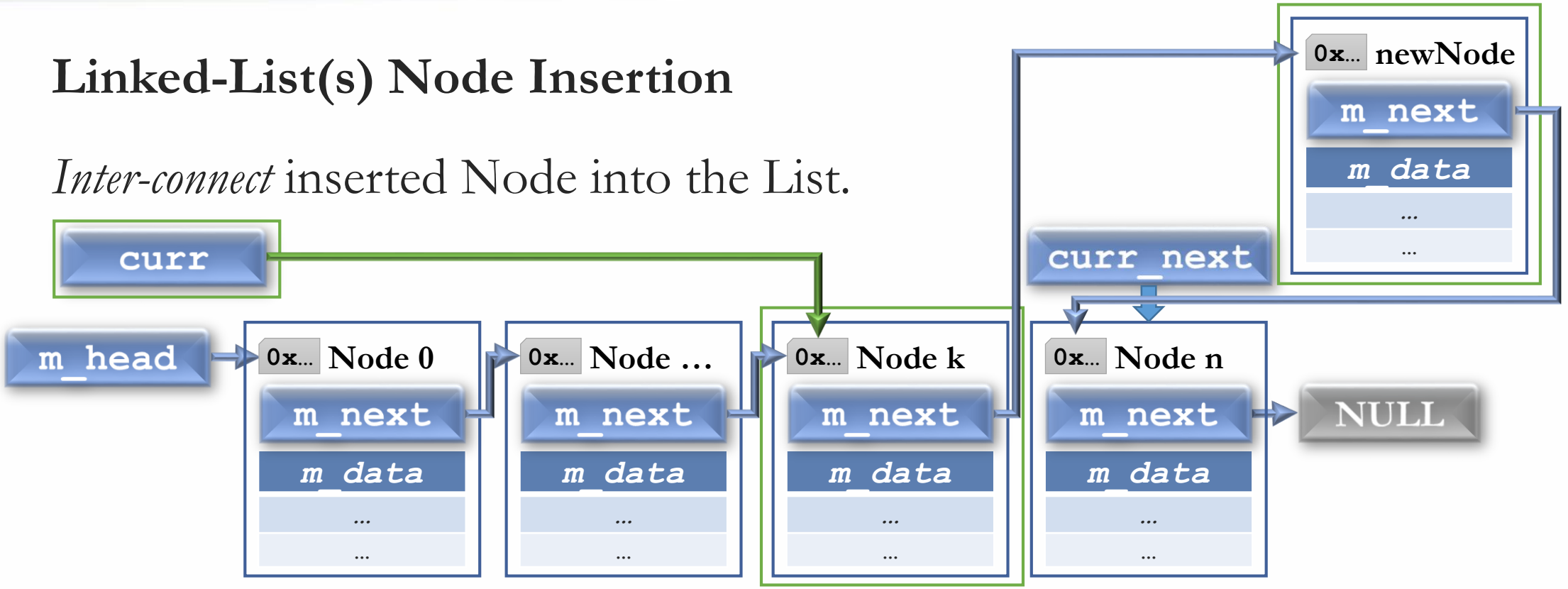


# Dynamic Data Structures

## Linked-List(s) Node Insertion

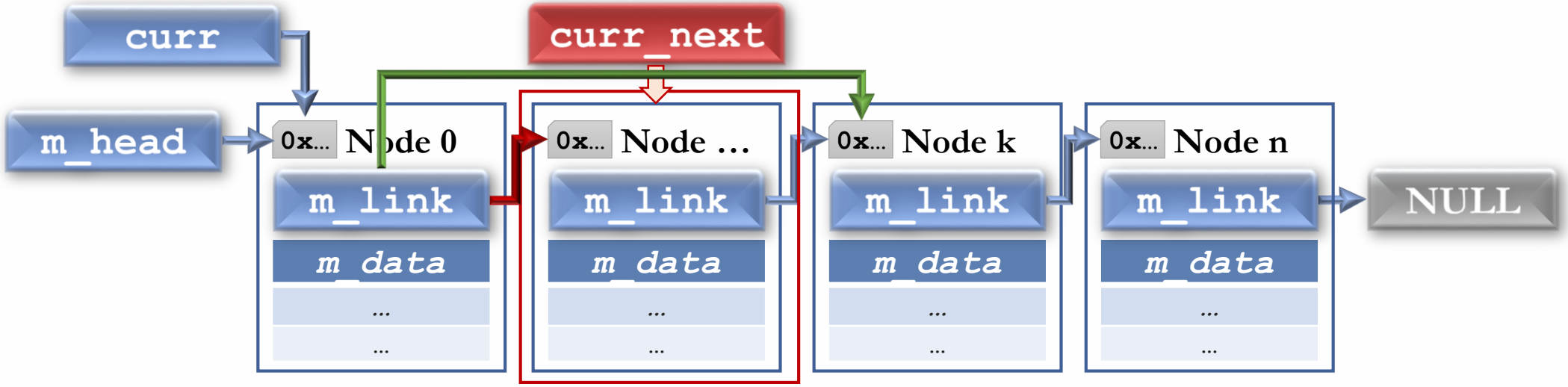
*Inter-connect* inserted Node into the List.




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_next) //traversal
  if ( curr->m_data == targetData ){ //search
    Node * curr_next = curr->m_next;
    curr->m_next = new Node(data, NULL);
    curr->m_next->m_next = curr_next;
  }
```

## Linked-List(s) Node Deletion

Change the Link of *predecessor* Node.

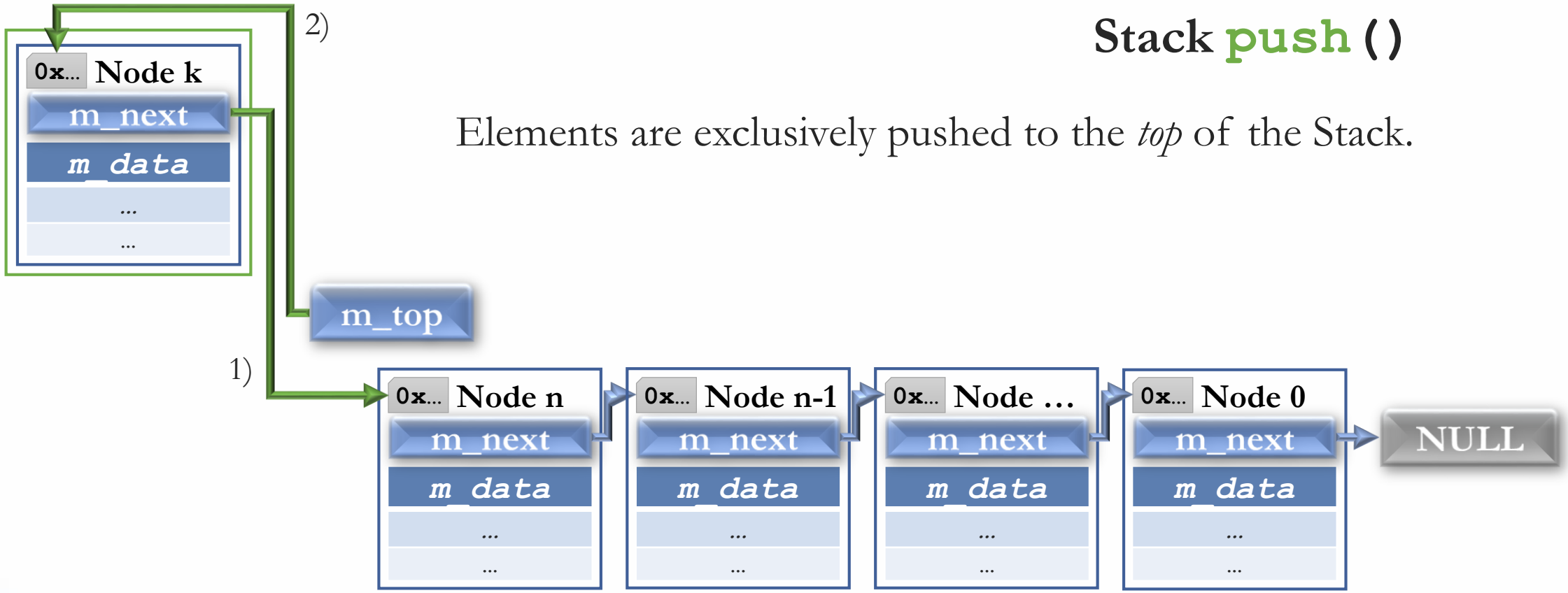



```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_next) //traversal
  if ( curr->m_next && curr->m_next->m_data == targetData ){ //search
    Node * curr_next = curr->m_next;
    curr->m_next = curr->m_next->m_next;
    delete curr_next;
  }
```

# Dynamic Data Structures

## Stack `push()`

Elements are exclusively pushed to the *top* of the Stack.

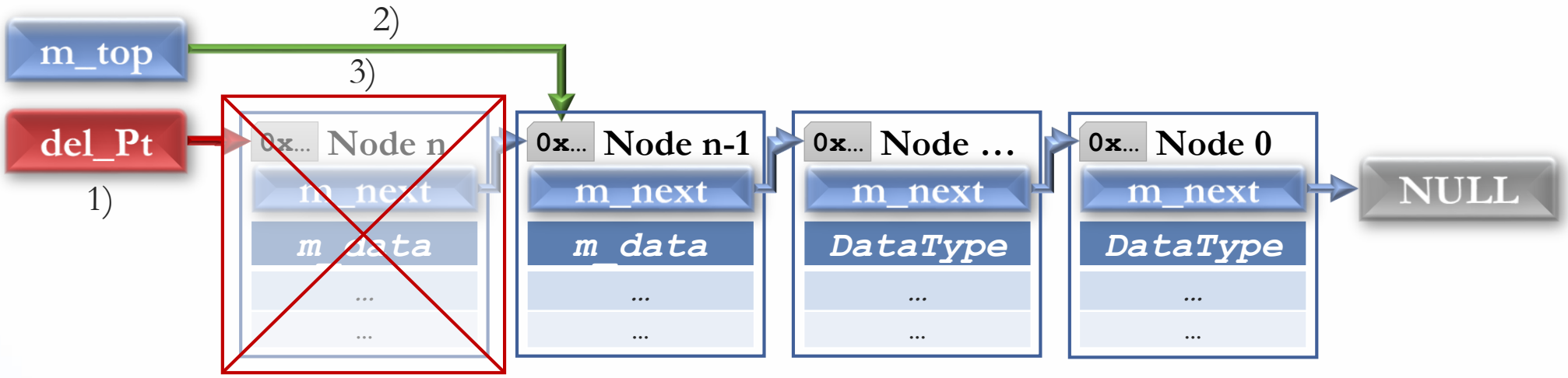



1) `newNode_Pt->m_next = m_top;`
2) `m_top = newNode_Pt;`

# Dynamic Data Structures

## Stack pop ()

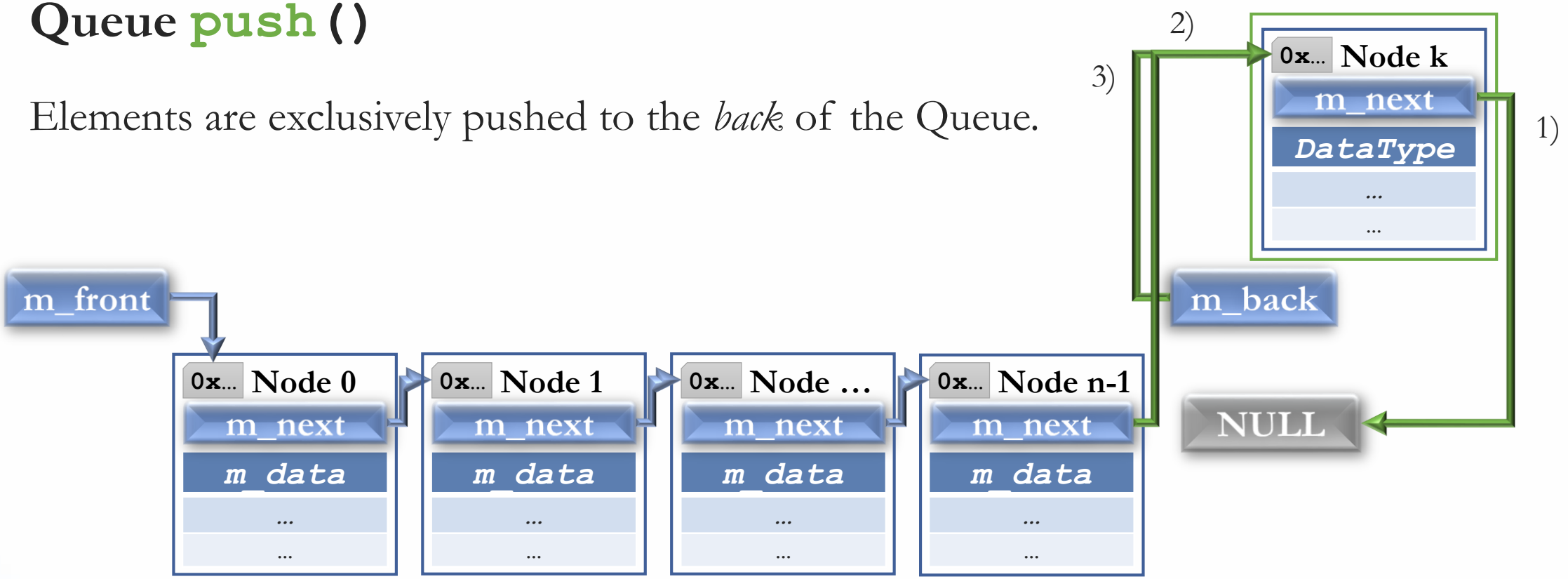
Elements are exclusively popped from the *top* of the Stack.




```
1) Node * del_Pt = m_top;
2) m_top = m_top->m_next;
3) delete del_Pt;
```

## Queue `push()`

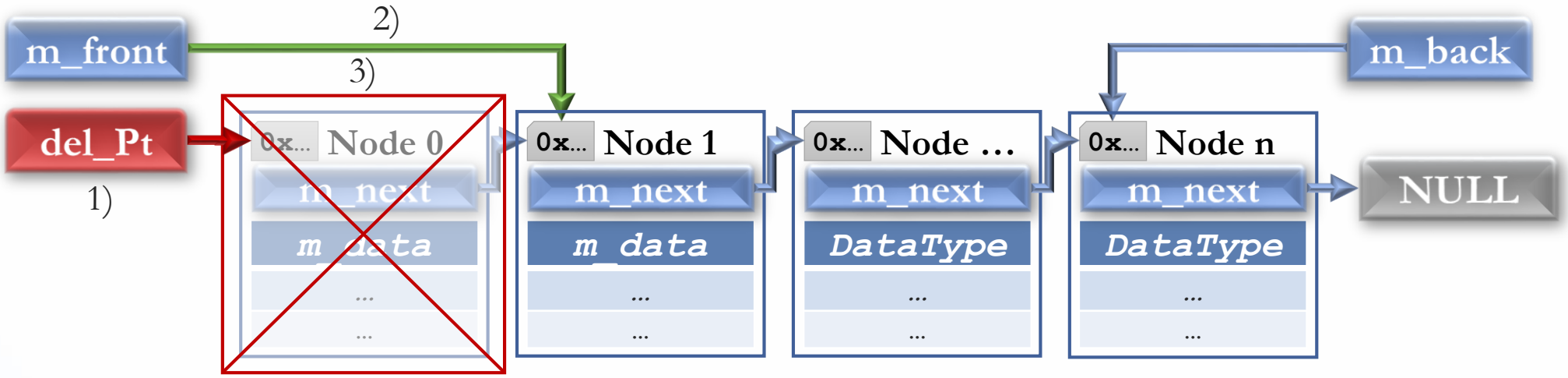
Elements are exclusively pushed to the *back* of the Queue.




```
1) newNode_Pt->m_next = NULL;
2) m_back->m_next = newNode_Pt;
3) m_back = newNode_Pt;
```

## Queue pop()

Elements are exclusively popped from the *back* of the Queue.



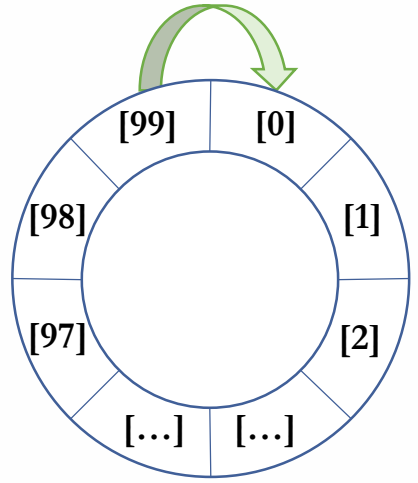

```
1) Node * del_Pt = m_front;
2) m_front = m_front->m_next;
3) delete del_Pt;
```

# Dynamic Data Structures

## Array-based Queue(s)

**push()-ing**: Advance **m_back** to next circular array position.

```
m_back = (m_back + 1) % m_maxsize;
++m_size; //remember the size
```




| | |
|---|---|
| m_size := 3 | m_front := 97 |
| m_maxsize := 99 | m_back := 99 |

| m_arr [0] | m_arr [1] | m_arr […] | m_arr [96] | m_arr [97] | m_arr [98] | m_arr [99] |
|---|---|---|---|---|---|---|
| char | char | char | char | char | char | char |
| … | … | … | … | A | B | C |

```
charQueue.push('D');
```

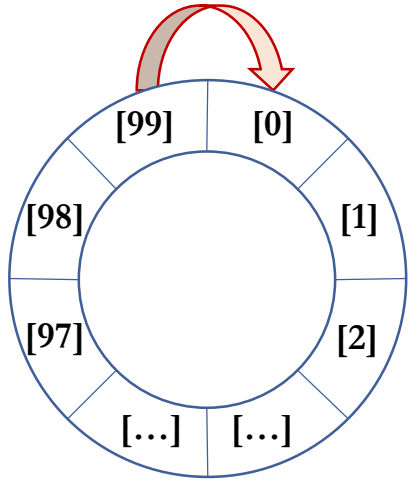
| | |
|---|---|
| m_size := 4 | m_front := 97 |
| m_maxsize := 99 | m_back := 0 |

| m_arr [0] | m_arr [1] | m_arr […] | m_arr [96] | m_arr [97] | m_arr [98] | m_arr [99] |
|---|---|---|---|---|---|---|
| char | char | char | char | char | char | char |
| D | … | … | … | A | B | C |

## Array-based Queue(s)

**pop()-ping**: Advance **m_front** to next circular array position.

```
m_front = (m_front + 1) % m_maxsize;
--m_size; //remember the size
```




| m_size := 4 | m_front := 99 |
|---|---|
| m_maxsize := 99 | m_back := 2 |

| m_arr [0] | m_arr [1] | m_arr [2] | m_arr […] | m_arr [97] | m_arr [98] | m_arr [99] |
|---|---|---|---|---|---|---|
| char | char | char | char | char | char | char |
| B | C | D | … | … | … | A |

```
charQueue.pop();
```

| m_size := 3 | m_front := 0 |
|---|---|
| m_maxsize := 99 | m_back := 2 |

| m_arr [0] | m_arr [1] | m_arr [2] | m_arr […] | m_arr [97] | m_arr [98] | m_arr [99] |
|---|---|---|---|---|---|---|
| char | char | char | char | char | char | char |
| B | C | D | … | … | … | … |

## Templated Function(s)

```
// forward declaration
template < class T >
void Swap(T & v1, T & v2);

// templated implementation
template < class T >
void Swap(T & v1, T & v2){ T temp = v1; v1 = v2; v2 = temp; };
```

Call with implicit / explicit template parameter statement:

```
int    i1=0,        i2=1;          Swap(i1, i2);
float  f1=0.1,      f2 = 99.9;     Swap< float >(f1, f2);
Car    c1("GRAY"), c2("WHITE");   Swap(c1, c2);
Date   d1(4,20),   d2(4,21);      Swap< Date >(1, d2);
```

| Inferred / Declared Type |
| --- |
| T : int |
| T : float |
| T : Car |
| T : Date |

## Templated Class(es)

```
// forward declarations -in order- (successful compilation requires these)
template <class T, size_t N_CART> class Train;
template <class T, size_t N_CART> std::ostream & operator<<(std::ostream & os,
                                                            const Train<T,N_CART> & car);

template <class T, size_t N_CART = 1>
class Train {
  public:
     Train();
     Train(size_t capacity, const T & item_value = T());
     Train(const Train<T,N_CART> & other);
     ~Train();
     Train<T,N_CART> & operator=(const Train<T,N_CART> & other);
     const T * getCart(size_t i) const;
     T * getCart(size_t i);
  // friend function declared as specialization of templated operator
  friend std::ostream & operator<< <> (std::ostream & os, const Train<T,N_CART> & car);
  private:
     T * m_carts[N_CART];  // an array of N_CART subarrays containing T objects
     size_t m_capacities[N_CART];  // an array of number of elements per cart
};
```

## Templated Class(es)

```
template <class T, size_t N_CART>
Train<T,N_CART>::Train(){
    for (size_t i = 0; i < N_CART; ++i){
        m_carts[i] = NULL; // initialize pointers
        m_capacities[i] = 0; // defensive
    }
}

template <class T, size_t N_CART>
Train<T,N_CART>::Train(size_t n_per_cart, const T & item_value){
    for (size_t i = 0; i < N_CART; ++i){ // iterative new needs exception handling
        // for each pointer allocate a new subarray
        m_carts[i] = new T [ n_per_cart ];
        m_capacities[i] = n_per_cart;
        for (size_t j = 0; j < n_per_cart; ++j){
            m_carts[i][j] = item_value;
        }
    }
}
```

## Templated Class(es)

```
template <class T, size_t N_CART>
Train<T,N_CART>::Train(const Train<T,N_CART> & other){
    for (size_t i = 0; i < N_CART; ++i){ // iterative new needs exception handling
        m_capacities[i] = other.m_capacities[i];
        if (!m_capacities[i])
          continue;
        m_carts[i] = new T [ other.m_capacities[i] ]; // allocate subarray
        for (size_t j = 0; j < m_capacities[i]; ++j){
            m_carts[i][j] = other.m_carts[i][j];
        }
    }
}

template <class T, size_t N_CART>
Train<T,N_CART>::~Train(){
    for (size_t i = 0; i < N_CART; ++i){
       //deleting a pointer to an allocated array, needs delete [] variant
        delete [] m_carts[i];
    }
}
```

## Templated Class(es)

```
template <class T, size_t N_CART>
Train<T,N_CART> & Train<T,N_CART>::operator=(const Train<T,N_CART> & other){
  if (this != &other){ // check for self-assignment
    for (size_t i = 0; i < N_CART; ++i){ // iterative new needs exception handling
        delete m_carts[i]; // deallocate previous memory (if necessary)
        m_carts[i] = NULL; // set pointers to NULL, exception might be thrown later
        m_capacities[i] = other.m_capacities[i];
        if (!m_capacities[i])
           continue;
        // for each pointer allocate a new subarray, sizes are stores in m_capacities
        m_carts[i] = new T [ other.m_capacities[i] ];
        for (size_t j = 0; j < m_capacities[i]; ++j){
            m_carts[i][j] = other.m_carts[i][j];
        }
    }
  }
  // return calling object
  return *this;
}
```

## Templated Class(es)

```
template <class T, size_t N_CART>
const T * Train<T,N_CART>::getCart(size_t i) const{  return m_carts[i];  }

template <class T, size_t N_CART>
T * Train<T,N_CART>::getCart(size_t i){  return m_carts[i];  }

// implementation of templated friend (non-member) function
template <class T, size_t N_CART>
std::ostream& operator<<(std::ostream & os, const Train<T,N_CART> & train){
    for (size_t i = 0; i < N_CART; ++i){
        if (train.m_carts[i]){
            for (size_t j = 0; j < train.m_capacities[i]; ++j){
                os << train.m_carts[i][j] <<" ";
            }
            os << endl;
        }
    }
    return os;
}
```

## Templated Dynamic Data Structures

```
// forward declaration of (any) class or function that will be a friend of Node
// and is necessary for any other component to compile
template <class T> class Queue;
// forward declaration of (any) class or function that will be a friend of DDS (Queue)
template <class T> std::ostream & operator<<(std::ostream & os, const Queue<T> & queue);

// templated Node
template <class T>
class Node{
    friend class Queue<T>;  //declaration of templated friend class
    public:
        Node() : m_next( NULL ){ }
        Node(const T & data, Node<T> * next = NULL) : m_data( data ), m_next( next ){ }
        const T & getData() const{ return m_data; }
        T & getData(){ return m_data; }
    private:
        Node<T> * m_next;
        T m_data;
};
```

# Templated Dynamic Data Structures

```
template <class T>     // templated DDS (Queue)
class Queue{
  friend std::ostream & operator<< <> (std::ostream & os, const Queue<T> & queue);
  public:
    Queue();
    Queue(size_t size, const T & value = T());
    Queue(const Queue<T> & other);
    ~Queue();
    Queue<T> & operator=(const Queue<T> & rhs);
    T & front();       const T& front() const;
    T & back();        const T& back() const;
    void push(const T & value);
    void pop();
    size_t size() const;
    void clear();
    void serialize(std::ostream & os) const;
  private:
    Node<T> * m_front;
    Node<T> * m_back;
};
```

## Templated Dynamic Data Structures

```
template <class T>
Queue<T>::Queue()
 : m_front( NULL ),
   m_back = NULL )
{
}

template <class T>
Queue<T> & Queue<T>::Queue(size_t size, const T & value)
  : m_front( NULL ),  //new in body might fail
    m_back( NULL )    //new in body might fail
{
  if (count){
     Node<T> * currNode = m_front = new Node<T>(value);
    while (--count){
       currNode = currNode->m_next = new Node<T>(value);
    }
    //currNode->m_next = NULL; //unnecessary, NULL-initialized by Node ctor
  }
}
```

## Templated Dynamic Data Structures

```
template <class T>
Queue<T>::Queue(const Queue<T> & other)
 : m_front( NULL ),  //new in body might fail
   m_back( NULL )    //new in body might fail
{
  Node<T> * otherNode = other.m_front;
  if (otherNode){
    Node<T> * myNode = m_front = new Node<T>(otherNode->m_data);
    while (otherNode->m_next){
      otherNode = otherNode->m_next;
      myNode = myNode->m_next = new Node<T>(otherNode->m_data);
    }
    m_back = myNode;
  }
}
```

# Templated Dynamic Data Structures

```
template <class T>
Queue<T>::~Queue(){
  //traverse to deallocate
  while (m_front){
    Node<T> * del_Pt = m_front;
    m_front = m_front->m_next;
    delete del_Pt;
  }
}

template <class T>
void Queue<T>::clear(){
  //traverse to deallocate
  while (m_front){
    Node<T> * del_Pt = m_front;
    m_front = m_front->m_next;
    delete del_Pt;
  }
  m_front = NULL; //reset pointers to NULL
  m_back = NULL; //reset pointers to NULL
}
```

## Templated Dynamic Data Structures

```
template <class T>
Queue<T> & Queue<T>::operator=(const Queue<T> & rhs){
  //check for self-assignment
  if (this != &rhs){
     clear(); //clear previous content first
     Node<T> * otherNode = rhs.m_front;
     if (otherNode){
          Node<T> * myNode = m_front = new Node<T>(otherNode->m_data);
          while (otherNode->m_next){
               otherNode = otherNode->m_next;
               myNode = myNode->m_next = new Node<T>(otherNode->m_data);
          }
          m_back = myNode;
     }
  }
  //return calling object by-reference
  return *this;
}
```

## Templated Dynamic Data Structures

```
template <class T>
const T & Queue<T>::front() const{  return m_front->m_data;  }

template <class T>
T & Queue<T>::front(){  return m_front->m_data;  }

template <class T>
const T & Queue<T>::back() const{  return m_back->m_data;  }

template <class T>
T & Queue<T>::back(){  return m_back->m_data;  }

template <class T>
size_t Queue<T>::size() const{
    size_t size = 0;
    Node<T> * trav_Pt = m_front;
    while (trav_Pt){
        ++size;
        trav_Pt = trav_Pt->m_next;
    }
    return size;
}
```

# Templated Dynamic Data Structures

```
template <class T>
void Queue<T>::push(const T & value){
  if (!m_back){  //empty back and front initialized
    m_back = m_front = new Node<T>(value);
  }
  else{  //append to back then update back
    m_back = m_back->m_next = new Node<T>(value);
  }
}

template <class T>
void Queue<T>::pop(){
  if (m_front){
    Node<T> * del_Pt = m_front;
    m_front = m_front->m_next;
    delete del_Pt;
    if (!m_front){  //no more elements after popping last one
      m_back = NULL;
    }
  }
}
```

## Templated Dynamic Data Structures

```
template <class T>
void Queue<T>::serialize(std::ostream & os) const{
  Node<T> * out_Pt = m_front;
  //traverse to output
  while (out_Pt){
    os << out_Pt->getData() << " ";
    out_Pt = out_Pt->m_next;
  }
}

template <class T>
std::ostream & operator<<(std::ostream & os, const Queue<T> & queue){
  queue.serialize(os);
  //return std::ostream object
  return os;
}
```

## The try – throw – catch Flow

```
Car::Car(const char * lPlates){
  setLicensePlates( lPlates );   2)
}
void Car::setLicensePlates(const char* lPlates){
    std::string lPlates_str( lPlates );
    if (lPlates_str.find_first_not_of("ABCDEF … 0123456789")
      throw (lPlates_str) ;
    m_licensePlates = lPlates_str;   3)
}
…
  Car * myCar_pt = NULL;
  try{
    myCar_pt = new Car("@#!~+^");   1)
  }
  catch(const std::string & ex_lp){
   cerr << "Plates " << ex_lp << " contain invalid characters...";
  }   4)
…
```

## The try – throw – catch Flow

```
Car::Car(const char * lPlates){
  setLicensePlates( lPlates );   2)
}
void Car::setLicensePlates(const char* lPlates){
    std::string lPlates_str( lPlates );
    if (lPlates_str.find_first_not_of("ABCDEF … 0123456789")
      throw (lPlates_str) ;  3)
    m_licensePlates = lPlates_str;
}
…
  Car * myCar_pt = NULL;
  try{
    myCar_pt = new Car("@#!~+^");  1)
  }
  catch(const std::string & ex_lp){
   cerr << "Plates " << ex_lp << " contain invalid characters...";
  }
…
```

## The try – throw – catch Flow

```
Car::Car(const char * lPlates){          5-a)
  setLicensePlates( lPlates );
}
void Car::setLicensePlates(const char* lPlates){
    std::string lPlates_str( lPlates );   4)
    if (lPlates_str.find_first_not_of("ABCDEF … 0123456789")
      throw (lPlates_str) ;               3)
    m_licensePlates = lPlates_str;
}
…
  Car * myCar_pt = NULL;
6) try{
    myCar_pt = new Car("@#!~+^");         5-b)
  }
  catch(const std::string & ex_lp){
7)  cerr << "Plates " << ex_lp << " contain invalid characters...";
  }
…    8)
```

## Semantics of `throw` and `catch`

Evaluate the value of **_expression_** and use it to copy-initialize an Exception Object of the same type (Copy-`ctor` of the type must be available).

```
/*a block scope somewhere*/
{
  throw _expression_ ;
}
```

Abandon current `catch` Block and re-`throw` the currently handled Exception object (the exact same – not a copy).

```
/*a block scope somewhere*/
{
  throw ;
}
```

Catch possible Exception type(s) in order (and potentially manipulate the Exception Object)

```
try{
  /* something */
catch (const ExceptionClass & ex){
  /*handling & manipulating
    ExceptionClass type ex Exceptions*/
}
catch (const int &){
  /*handling int type Exceptions*/
}
catch (...)
{
  /*handling any type of Exception*/
}
```

# Finals Sample – Program 1

```
#include <iostream>
#include <cstring>  // allowed to use built-in c-string functions
using namespace std;
//////////////////////////HELPERS//////////////////////////////  (should be considered as pre-implemented & working)
class Cover{
 public:
  Cover() : m_hard(false){}
  Cover(bool hard) : m_hard(hard){}
 friend std::ostream & operator<<(std::ostream & os, const Cover & cover){ os << (cover.m_hard?"hardcover":"paperback");
                                                                           return os; }
 friend std::istream & operator>>(std::istream & is, Cover & cover){ is >> cover.m_hard; return is; }
  bool getValue() const{ return m_hard; }
 private:
  bool m_hard;
};
class Client{
 public:
  Client(){ m_name = NULL; }
  Client(const char * name){ m_name = new char[ strlen(name)+1 ]; strcpy(m_name,name); }
  Client(const Client & other){ m_name = new char[ strlen(other.m_name)+1 ]; strcpy(m_name,other.m_name); }
  ~Client(){ delete [] m_name; }
  Client & operator=(const Client & other){ delete [] m_name; m_name = new char[ strlen(other.m_name)+1 ];
                                            strcpy(m_name,other.m_name); }
  const char * getName() const{ return m_name; }
 friend std::istream & operator>>(std::istream & is, Client & client){ if (client.m_name){ is >> client.m_name; } return is; }
 friend std::ostream & operator<<(std::ostream & os, const Client & client){ os << client.m_name; return os; }
 private:
   char * m_name;
};
```

# Finals Sample – Program 1

```
////////////////////////////BOOK//////////////////////////////
class Book {
  friend std::ostream & operator<<(std::ostream & os, const Book & book);
  public:
    Book();
    Book(const char * title, const Cover & cover=Cover(),
         const Client * client=NULL, size_t serial=count);
    Book(const Book & other);
    ~Book();
    Book & operator=(const Book & other);
    const Cover & getCover() const;
    void setCover(const Cover & cover);
    const Client * getClient() const;
    void setClient(const Client * client);
    void serialize(std::ostream & os) const;

  private:
    char * m_title;  // raw pointer
    Cover m_cover;   // composition
    const Client * m_client;  // aggregation
    const size_t m_serial;  // const
    static size_t count;    //static
};
```

# Finals Sample – Program 1

```
size_t Book::count = 0;  // instantiation of static variables

Book::Book() : m_serial( count++ ){
  m_title = NULL;  // initialization of pointers
  m_client = NULL;
}

Book::Book(const char * title, const Cover & cover, const Client * client, size_t serial)
      : // set static to the greater value, and initialize const member at the same time
         m_serial( count = serial>count?serial:count ),
         m_cover(cover),
         m_client(client) {
  m_title = new char [ strlen(title)+1 ];
  strcpy(m_title, title);
  ++count;  // increment at the end, constructor done & no exceptions occurred
}
```

# Finals Sample – Program 1

```
Book::Book(const Book & other)
      :  m_serial( count ),
         m_cover(other.m_cover),
         m_client(other.m_client) {
  m_title = new char [ strlen(other.m_title)+1 ];
  strcpy(m_title, other.m_title);
  ++count;  // increment at the end, constructor done & no exceptions occurred
}

Book & Book::operator=(const Book & other){
  if (this != &other){  // check for self-assignment
    delete [] m_title;  // first deallocate dynamic memory in assignment
    m_title = NULL;  // and re-set pointers to NULL
    m_title = new char [ strlen(other.m_title)+1 ];
    strcpy(m_title, other.m_title);
    m_client = other.m_client;
  }
  //return calling object by-reference
  return *this;
}
```

# Finals Sample – Program 1

```
Book::~Book(){
  // cover is class member object (composition) - will be automatically destroyed
  // m_client is pointer to external object (aggregation) - no deleting here
  delete [] m_title; // m_title is object-bound dynamic memory - delete
  //--count;  // no decrement, count specified to generate unique increasing serial(s)
}

void Book::serialize(std::ostream & os) const{
  os << m_serial<<":"<<m_title<<"("<<m_cover<<")";
  if (m_client)
    os <<" client:"<< *m_client;  // m_client is a pointer! cout has to dereference it
  return os;
}

std::ostream & operator<<(std::ostream & os, const Book & book){
  book.serialize(os);
  return os;
}

const Cover & Book::getCover() const{       return m_cover;   }

void Book::setCover(const Cover & cover){   m_cover = cover;   }

const Client * Book::getClient() const{   return m_client;   }

void Book::setClient(const Client * client){   m_client = client;   }
```

# Finals Sample – Program 1

```
//////////////////BOOK-INHERITANCE-POLYMORPHISM/////////////////
class Book {
  public:
     …
    virtual void serialize(std::ostream& os) const;  // making this virtual causes
                                                     // dynamic binding to work
  protected:
    char * m_title;  //moved to protected access
    Cover m_cover;  //moved to protected access
    const size_t m_serial;  //moved to protected access
    static size_t count;  //moved to protected access
  private:
    const Client * m_client;
};

// the virtual method implementation remains as is
void Book::serialize(std::ostream & os){
  os << m_serial<< ":" <<m_title<< "(" << m_cover << ")";
  if (m_client)
    os << " client:" << *m_client;  // m_client is a pointer! cout has to dereference it
  return os;
}
```

# Finals Sample – Program 1

```
//////////////////////////CHILDRENBOOK//////////////////////////////
class ChildrenBook : public Book{  //inheritance
  public:
    ChildrenBook();
    ChildrenBook(const char * title, bool graphic, const Cover & cover=Cover(),
                 const Client * client=NULL, size_t serial=count);
    ChildrenBook(const ChildrenBook & other);
    ~ChildrenBook();
    ChildrenBook & operator=(const ChildrenBook & other);
    bool getGraphic() const;
    void setGraphic(const bool& graphic);

    virtual void serialize(std::ostream& os) const;  // overridden method is virtual in Base
                                                     // class therefore dynamic binding enabled

  /*  Unnecessary if serialize is virtual, dynamic binding on Base class object will work !  */
  /* friend std::ostream & operator<<(std::ostream & os, const ChildrenBook & childrenbook); */

  private:
    bool m_graphic;
};
```

# Finals Sample – Program 1

```
ChildrenBook::ChildrenBook()
      : Book(){ //call default base ctor at instantiation
    // count increases when base class constructor gets called
}

ChildrenBook::ChildrenBook(const char * title, bool graphic, const Cover & cover,
                           const Client * client, size_t serial)
      : //use base class parametrized constructor with arguments (passing them along)
        Book(title, cover, client, serial),
        m_graphic(graphic){
  // count increases when base class constructor gets called
}

ChildrenBook::ChildrenBook(const ChildrenBook & other)
      :  //have to use GetClient() because m_client is private, not protected
        Book(other.m_title, other.m_cover, other.getClient(), other.m_serial),
        m_graphic(other.m_graphic){
  // count increases when base class constructor gets called
}

ChildrenBook::~ChildrenBook(){
  // derived class has no dynamic memory to manage
  // base class destructor will get automatically called right after
}
```

# Finals Sample – Program 1

```
ChildrenBook & ChildrenBook::operator=(const ChildrenBook & other){
  if (this != &other){  // check for self-assignment
    delete [] m_title;  // handle base class members
    m_title = NULL;
    m_title = new char [ strlen(other.m_title)+1 ];
    strcpy(m_title, other.m_title);
    m_client = other.m_client;
    m_graphic = other.m_graphic;  // handle derived class members
  }
  //return calling object by-reference
  return *this;
}
```

# Finals Sample – Program 1

```
// overriding function of the base class serialize(), virtual as well to enable Dynamic Binding
void ChildrenBook::serialize(std::ostream & os){
  os << m_serial<<":"<<m_title<<"("<<m_cover<<","<<(m_graphic?"graphic":"novel")<<")";
  if (getClient()){
    //m_client is a pointer, and it is also private (not protected)
    os << " client:" << *getClient();
  }
}

/* Unnecessary if serialize is virtual, dynamic binding on Base class object will work! */
std::ostream & operator<<(std::ostream & os, const ChildrenBook & childrenbook){
  childrenbook.serialize(os);
  return os;
}
```

# Finals Sample – Program 1

```
////////////////////////////MAIN////////////////////////////////
int main()
{
   Client jDoe("John Doe");
   Book myBook("LOTR ROTC", Cover(true), &jDoe, 999);

   Client jDoeJr("John Doe Jr");
   ChildrenBook myChildBook("LOTR comic", true, Cover(false), &jDoeJr);

   Book * book_Pt;

   book_Pt = &myBook;
   cout << *book_Pt << endl;

   book_Pt = &myChildBook;
   cout << *book_Pt << endl; /* this uses the friend operator<< function which is not a
                                member function (and hence cannot be a virtual one) */
   /*however if the Base class method is virtual (dynamic binding) then the Derived
     class method override will be called */

   return 0;
}
```

# Finals Sample – Program 2

```
class DynamicMatrix {
 public:
  // 1) instatiates a [0]x[0] NULL matrix
  DynamicMatrix();
  // 2) instatiates a [rows]x[cols] matrix with all elements set to [value]
  DynamicMatrix(size_t rows, size_t cols, int value=0);
  // 3) instantiates via matrix copy
  DynamicMatrix(const DynamicMatrix & otherDynamicMatrix);
  // 4) destroys matrix and deallocates dynamic memory
  ~DynamicMatrix();

  // 5) assignment operator
  DynamicMatrix & operator=(const DynamicMatrix & other);
  // 6) parenthesis operator, to be used for [row],[col] indexing
  int & operator()(size_t row_pos, size_t col_pos);
  // 7) checks if two matrices are by-size-and-by-value equal
  bool operator==(const DynamicMatrix & other);

 private:
  size_t m_rows;
  size_t m_cols;
  int ** m_matrix;
};
```

# Finals Sample – Program 2

```
DynamicMatrix::DynamicMatrix(){
    m_matrix = NULL;  //initialize pointer(s) to NULL
    m_rows = 0;  //defensive strategy sometimes desired
    m_cols = 0;  //defensive strategy sometimes desired
}

DynamicMatrix::~DynamicMatrix(){
    // check that top-level pointer is not NULL
    // (otherwise cannot index by it m_matrix[i] to call delete on row(s) subarrays)
    if (m_matrix){
      for (size_t i=0; i<m_rows; ++i){
        delete [] m_matrix[i];  // delete subarrays via pointers of top-level array
      }
      delete [] m_matrix;  // delete top level array of pointers
    }
}
```

# Finals Sample – Program 2

```
DynamicMatrix::DynamicMatrix(size_t rows, size_t cols, int value){
     //get new m_rows, m_cols values
     m_rows = rows;
     m_cols = cols;
     //allocate memory
     try{
       m_matrix = new int * [m_rows];  //allocate memory for rows (array of pointers to row subarrays)
       for (size_t i=0; i<m_rows; ++i)  //initialize all these pointers to NULL
          m_matrix[i] = NULL;
       for (size_t i=0; i<m_rows; ++i){
          try{
            m_matrix[i] = new int [m_cols];  //allocate memory for row i (subarray of int(s))
          }
          catch(const std::bad_alloc & ex){  //delete all row(s) i that were allocated before
            for (; i>=0; --i)
              delete [] m_matrix[i];
            throw ;  //re-throw the original exception
          }
       }
       for (size_t i=0; i<m_rows; ++i)  //reached this far, now initialize matrix with values
         for (size_t j=0; j<m_cols; ++j)
            m_matrix[i][j] = value;
     }
     catch(const std::bad_alloc & ex)
     {
       delete [] m_matrix;
     }
}
```

# Finals Sample – Program 2

```
DynamicMatrix::DynamicMatrix(const DynamicMatrix & otherDynamicMatrix){
      //free current memory first
      if (m_matrix){  //check that top-level pointer is not NULL
        for (size_t i=0; i<m_rows; ++i){  delete [] m_matrix[i];  }
        delete [] m_matrix;
      }
      m_rows = otherDynamicMatrix.m_rows;  // get new m_rows, m_cols values
      m_cols = otherDynamicMatrix.m_cols; // get new m_rows, m_cols values
      try{
        m_matrix = new int * [m_rows];
        for (size_t i=0; i<m_rows; ++i)
            m_matrix[i] = NULL;
        for (size_t i=0; i<m_rows; ++i){
            try{
              m_matrix[i] = new int [m_cols];
            }
            catch(const std::bad_alloc & ex){
              for (; i>=0; --i)
                delete [] m_matrix[i];
              throw ;
            }
        }
        for (size_t i=0; i<m_rows; ++i)  //reached this far, initialize matrix with otherDynamicMatrix
          for (size_t j=0; j<m_cols; ++j)
            m_matrix[i][j] = otherDynamicMatrix.m_matrix[i][j];
      }
      catch(const std::bad_alloc & ex)
      {
        delete [] m_matrix;
      }
}
```

# Finals Sample – Program 2

```
DynamicMatrix & DynamicMatrix::operator=(const DynamicMatrix & otherDynamicMatrix){
  if (this != &otherDynamicMatrix){  //check for self-assignment first
    //free current memory first
    if (m_matrix){  //check that top-level pointer is not NULL
      for (size_t i=0; i<m_rows; ++i){  delete [] m_matrix[i];  }
      delete [] m_matrix;
    }
    // get new m_rows, m_cols values
    m_rows = otherDynamicMatrix.m_rows;
    m_cols = otherDynamicMatrix.m_cols; // get new m_rows, m_cols values
    try{
     …
     … // 2d array allocation handling here
     …
    }
    catch(const std::bad_alloc & ex)
    {
      delete [] m_matrix;
    }
  }
  //return calling object
  return *this;
}
```

# Finals Sample – Program 2

```
bool DynamicMatrix::operator==(const DynamicMatrix & otherDynamicMatrix){
    //checking for equality pre-requires equal rows, cols
    if (m_rows!=otherDynamicMatrix.m_rows || m_cols!=otherDynamicMatrix.m_cols){
      return false;
    }
    for (size_t i=0; i<m_rows; ++i){
      for (size_t j=0; i<m_cols; ++i){
        if (m_matrix[i][j] != otherDynamicMatrix.m_matrix[i][j]){
          return false;
        }
      }
    }
    return true;
}

int & DynamicMatrix::operator()(size_t row_pos, size_t col_pos){
    return m_matrix[row_pos][col_pos];
}
```

# Finals Sample – Program 3

## Templated Dynamic Data Structures

```
template <class T>     // templated DDS (Queue)
class Queue{
  friend std::ostream & operator<< <> (std::ostream & os, const Queue<T> & queue);
  public:
    Queue();
    Queue(size_t size, const T & value = T());
    Queue(const Queue<T> & other);
    ~Queue();
    Queue<T> & operator=(const Queue<T> & rhs);
    T & front();       const T & front() const;
    T & back();        const T & back() const;
    void push(const T & value);
    void pop();
    size_t size() const;
    void clear();
    void serialize(std::ostream & os) const;
  private:
    Node<T> * m_front;
    Node<T> * m_back;
};
```

# Finals Sample – Question 1

```
#include <iostream>
#include <cstring>

using namespace std;

class Base{
  public:
    Base(){ cout << "B" << ++count << endl; }
    ~Base(){ cout << "~B" << --count << endl; }
  protected:
    static size_t count;
};
size_t Base::count = 0;

class Derived : public Base{
  public:
    Derived(){ cout << "D" << ++d_count << "," << count << endl; }
    ~Derived(){ cout << "~D" << --d_count << "," << count << endl; }
  private:
    static size_t d_count;
};
size_t Derived::d_count = 0;

void fB(){
     Base b;
}
void fD(){
     Derived d;
}
```

```
int main()
{
  fB();
  fD();
  return 0;
}
```

Output:

```
B1
~B0
B1
D1,1
~D0,1
~B0
```

# Finals Sample – Question 2

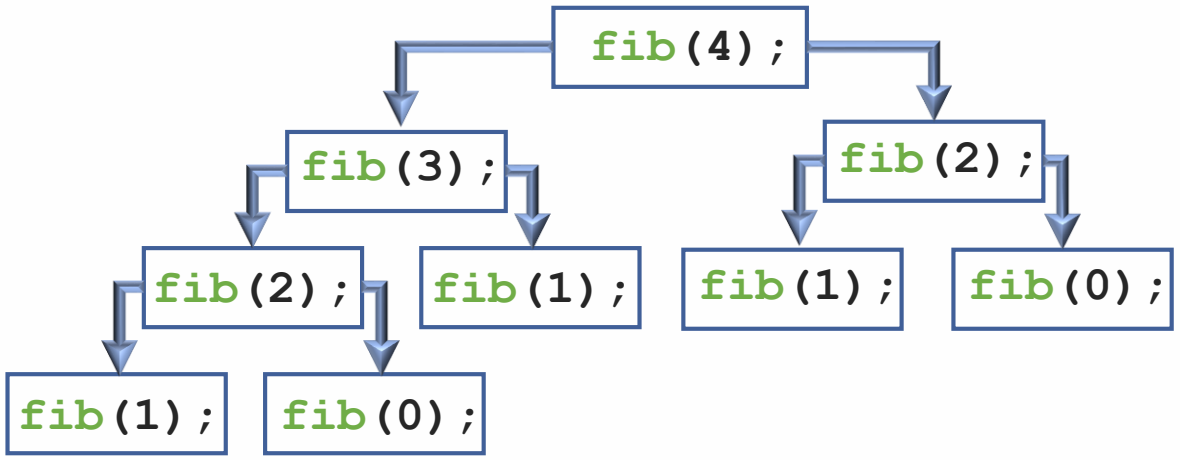
```
#include <iostream>
#include <cstring>

using namespace std;

void rec (int n){
  if (n < 0){              // base case here is never reached, modulo is positive or zero
    cout << n << endl;
  }
  else {
    rec( n / 10 );
    cout << ( n % 10 ) << endl;
  }
}


int main()
{
  rec(123);

  return 0;
}
```

# Finals Sample – Question 2

```
#include <iostream>
#include <cstring>

using namespace std;

int rec (int n) {
  cout << n << " ";
  if (n > 1)
    return rec (n-1) + rec (n-2);
  else
    return n;
}
```

```
int main()
{
  int r = rec(4);

  cout << endl << r;

  return 0;
}
```

Output:
432101210
3



# Finals Sample – Question 3

```
#include <iostream>
#include <cstring>

using namespace std;
```

```
int main(){
  Matrix<float, 10, 5> mat;
  cout << mat;
  return 0;
}
```

```
// without even one of the forward declarations below in this exact order, the compiler does not understand any
// Matrix<T,NROWS,NCOLS> statements when it gets to the templated overloaded operator<< function declaration
template<class T, size_t NROWS, size_t NCOLS> class Matrix;
template<class T, size_t NROWS, size_t NCOLS> std::ostream& operator<<(std::ostream& os, const
Matrix<T,NROWS,NCOLS>& matrix);

template<class T, size_t NROWS=1, size_t NCOLS=1>
class Matrix{
 public:
  Matrix(){}
 friend std::ostream& operator<< <> (std::ostream& os, const Matrix<T,NROWS,NCOLS>& matrix);
 private:
  T container[NROWS][NCOLS];
};

template<class T, size_t NROWS, size_t NCOLS>
std::ostream& operator<<(std::ostream& os, const Matrix<T,NROWS,NCOLS>& matrix){
  for (size_t i=0; i<NROWS; ++i){
    for (size_t j=0; j<NCOLS; ++j)
      os  <<  matrix.container[i][j] << " ";
    os << std::endl;
  }
  os << std::endl;
}
```

# Finals Sample – Question 4

```
#include <iostream>
#include <string.h>

using namespace std;

class MyException{
 public:
    // instantiates and initializes info string
    MyException(const char * s) : m_info(s){ }

    // sets info string to desired value
    void setInfo(const char * s){ m_info = s; }

  // handles output of exception object data (info string)
  friend std::ostream& operator<<(std::ostream & os,
                                  const MyException & e){
     os << e.m_info;
     return os;
 }
 private:
    std::string m_info;
};

class A{
  public:
    A(){ cout << "A" << endl; }
    ~A(){ cout << "~A" << endl; }
};
```

# Finals Sample – Question 4

```
int main(){

  try{
    A anA;

    try{
      A anotherA;

       //error detected
       throw MyException("Something awful happened here...");
    }
    catch(MyException & e){
       cerr << e << endl;
       e.setInfo( "It's been taken care of!" );
       throw;
    }
  }
  catch(const MyException & e){
    cerr << e << endl;
  }

  return 0;
}
```

Output:

```
A
A
~A
Something awful happened here...
~A
It's been taken care of!
```

# CS-202

Time for Questions !