```
class X {
    int i;
public:
    X();
    void f1(int a);
};

X::X() {
    :
}

void X::f1(int a) {
    :
}
```

```
template <class T>
class X {
    T int i;
public:
    X();
    void f1(int a);
};

template <class T>
X<T>::X() {
    :
}

template <class T>
void X<T>::f1(int a) {
    :
}
```

Template class specialization

```
template <>
  class X <char> {
  char int i;
  public:
      X();
      void f1( int a);
  };

X <char> :: X() {
      :
      }
void X<char>:: f1 (int a){
    :
    }
```

Exception Handling ⟹ try
catch
throw

```
int *p;
  :
p = (int *) malloc ( .... );
if (p == NULL) {
    : // Error Handling.
}
```

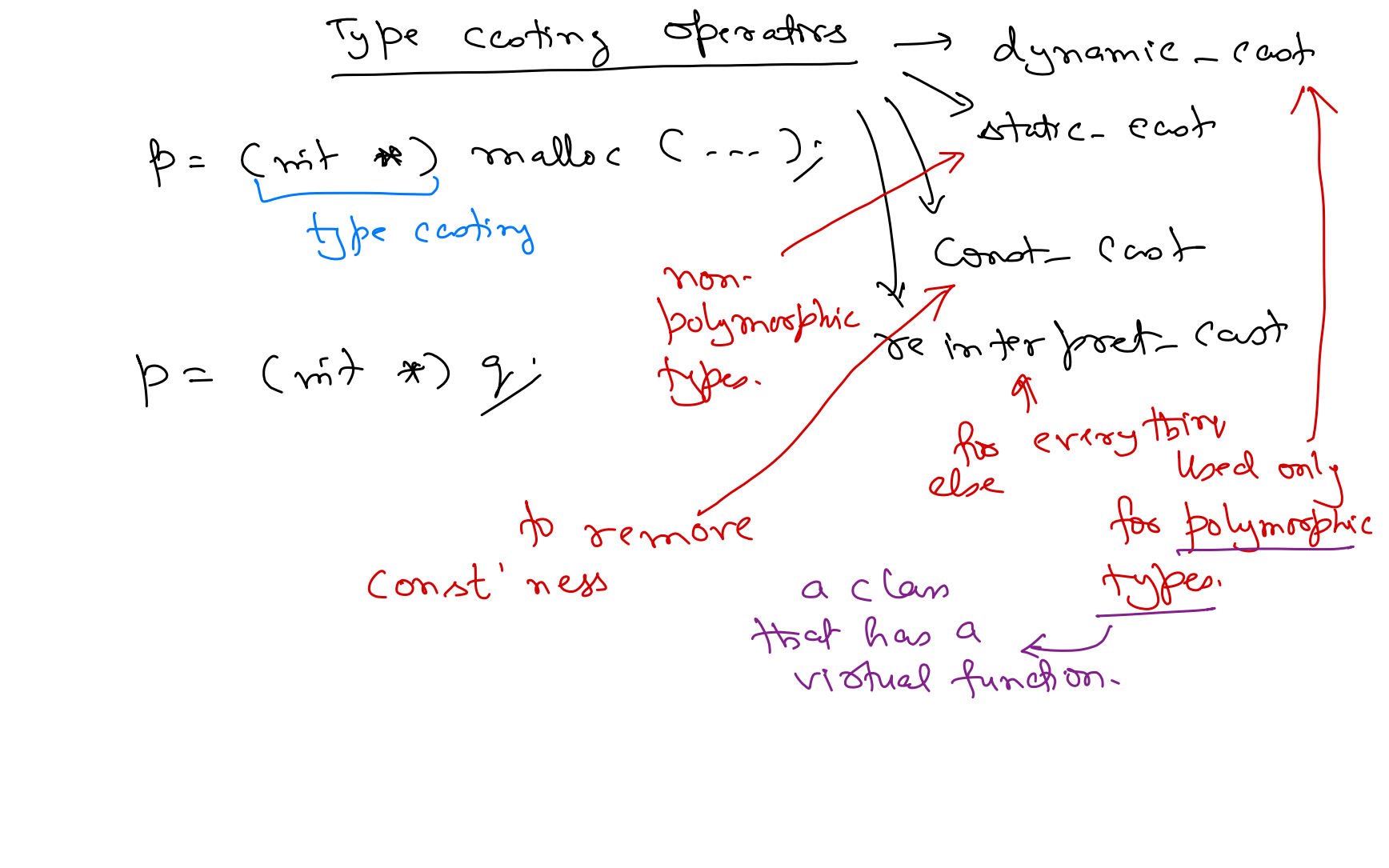
```
void f1( ) {
  :
  :
if (...) // error situation
    throw (1);
  :
}
```

(1) → value of exception

```
try {
    f1();
}
catch (int e) {
    :
}
catch (float f) {
    :
}
catch (some class obj) {
    :
}
```

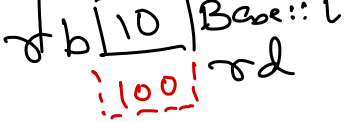
← statements in try block that might throw exception. Some value.

← Implement exception handlers.


Type casting operators
dynamic_cast
static_cast
Const_cast
reinterpret_cast
p = (int *) malloc (...);
type casting
p = (int *) q;
non-polymorphic types.
for everything else
Used only for polymorphic types.
to remove Const'ness
a class that has a virtual function.

```
class Base {
public:
        int i;
        :
};
class Derived: public Base
   public:
          int j;
          :
};
```

```
Base objB;
:
Base &rb = objB;
rb.i = 10;
:
Derived &rd
= rb;
```

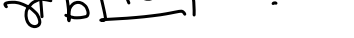



```
Derived &rd =
    static_cast<Derived&>(rb);
rd.j = 100;
```

```
class Base {
public:
        int i;
virtual void f1(){..}
        :
};
class Derived: public Base
  public:
        int j;
        :
};
```

polymorphic type.

```
Base objB;
:
Base &rb = objB;
rb.i = 10;
:
Derived &rd
  = rb;
```




```
Derived &rd =
  dynamic_cast<Derived&>(rb);
rd.j = 100;
```

it will check if casting is safe or not.

If not allocated it throws exception
of type bad_cast

RTTI ⟶ typeid
type_info

File I/O

What is a file? ⇒ Sequence of bytes.

Text / Binary file processing.

↑ Printable characters

↑ Memory dump.

istream cin

ostream cout

ifstream ofstream

↘ ↙

fstream

Command line arguments

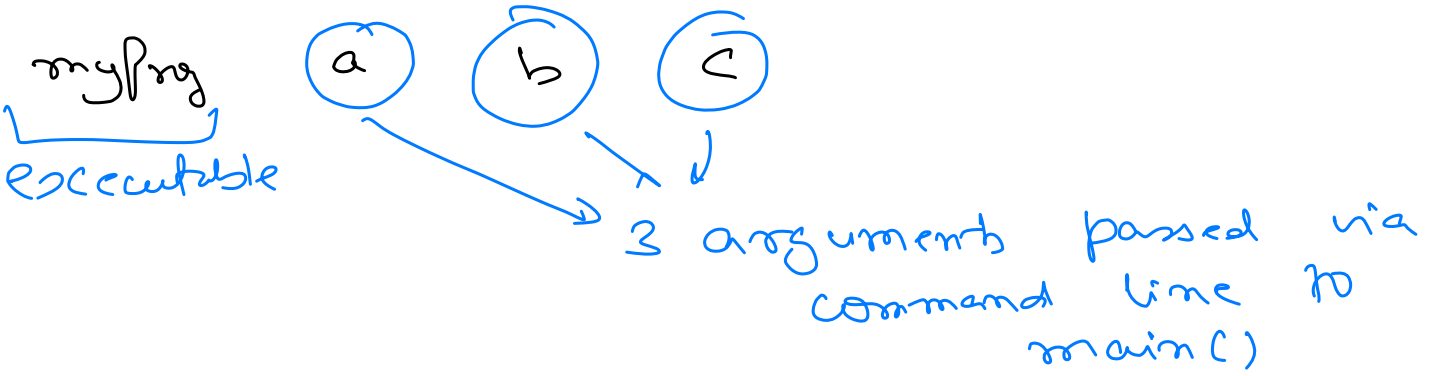
⇓

Passed to main()

```
main ( int , char * [])
```

int → number of arguments

char * [] → array of strings, each string is one argument.

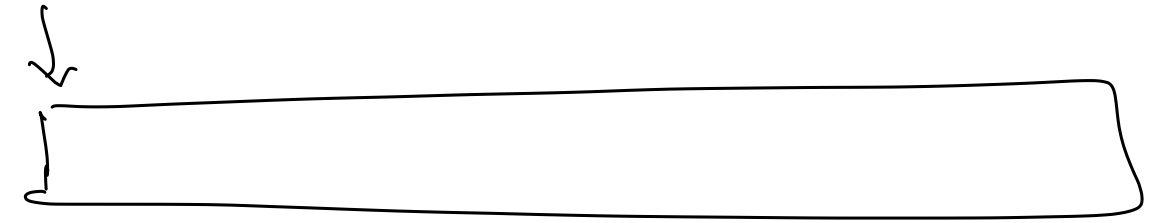


main (int , char * [] , char * [])

array of strings for environment variables. with last element being NULL.

position in file where next I/O operation will happen.

file pointer



file

fseek ⟹ move file pointer to a specific

place in file.

tell() ⇒ current file pointer pos.

seek (value, reference)

10 — begin ⇒ set file pointer to 10th byte from start of file

-10 — end ⇒ set file pointer to 10th byte from end towards start of file.

5 — current ⇒ set file pointer ahead by 5 byte from current pos.

STL

⇓

Standard Template Library

Consists of a collection of template classes known as containers.

⇓

Stores data.

Some of the template classes / containers are vector, list, stack, map, etc.

vector → resizable array

To access element stored in containers in a generic manner STL has iterators.

think of it like a pointer to access array elements.

STL also contains template function, known as algorithms to perform common operations like sort, reverse, search, etc.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
  std::vector<int> nums;

  // Add elements to vector.
  nums.push_back(10);
  nums.push_back(1);
  nums.push_back(5);
  nums.push_back(100);
  nums.push_back(50);
  nums.push_back(25);
  nums.push_back(75);

  // Use sort algorithm for sorting data in range
  // defined by iterators
  std::sort(nums.begin(), nums.end());

  // Using iterator access and print all elements.
  for (auto it = nums.begin(); it != nums.end(); ++it) {
    std::cout << *it << "\n";
  }

  return 0;
}
```

container specific header file

declares algorithms ⇒ template functions

create object of vector to store int

each container class provide a member function to add element to it

using sort algorithm (template function) to sort data.

indicates end of container

returns a way to access first element in container

goto next element

defines

type of it based on value with which its initialized.

access value stored in container, as indicated by iterator.

start with first element of container

until we have reached end of container

One way to add "iterator" to Array class we have implemented.

```
class Array {
    int* pData;
    const int size;

    void copyArray(int* dest, int* src, int size) {
        for (int i = 0; i < size; ++i) {
            dest[i] = src[i];
        }
    }

public:
    Array(int n) : size(n) {
        std::cout << "Allocate memory for array of size " << size << "\n";
        pData = new int[size];
    }
```

```
~Array() {
    std::cout << "Free array memory of size " << size << "\n";
    delete[] pData;
}
int& operator[](int i) {
    static int temp;

    if ((i < 0) || (i >= size)) {
        return temp;
    }

    return pData[i];
}
Array(Array& obj) : size(obj.size) {
    std::cout << "Array copy constructor of size " << size << "\n";
    pData = new int[size];
    copyArray(pData, obj.pData, size);
}

// One way to implement "iterator for this class".
int* begin() const {
    return pData;
}
int* end() const {
    return pData + size;
}
```

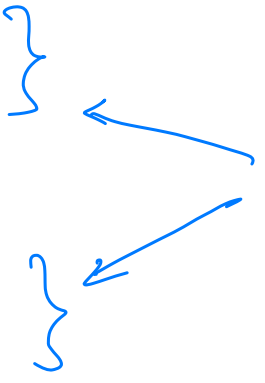
```
};

int main() {
    Array nums(5);

    std::cout << "Enter 5 numbers: ";
    // Using iterator access and all elements.
    for (auto it = nums.begin(); it != nums.end(); ++it) {
        std::cin >> *it;
    }

    std::cout << "You entered ...\n";
    // Using iterator access and print all elements.
    for (auto it = nums.begin(); it != nums.end(); ++it) {
        std::cout << *it << "\n";
    }

    return 0;
}
```



Using "iterator" to access all elements of Array class.