0) From C to C++

From struct to classes.

Free -> distructor

The generation : Include <iostream>
Using namespace std:

Cout – output , Cin-input , Endle

1) Introduction to OOP in C++

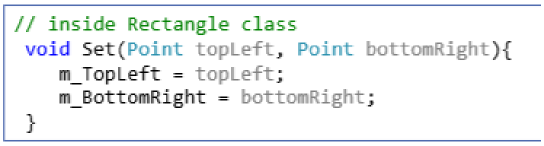
Permissions -Public- everyone can use,

Private- can use only from the class,

Protected – can use from the father only.

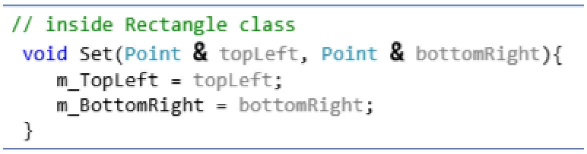
The default is private unless we define another.

Passing parameters by value:



```
// inside Rectangle class
 void Set(Point topLeft, Point bottomRight){
    m_TopLeft = topLeft;
    m_BottomRight = bottomRight;
 }
```

Passing parameters by reference:

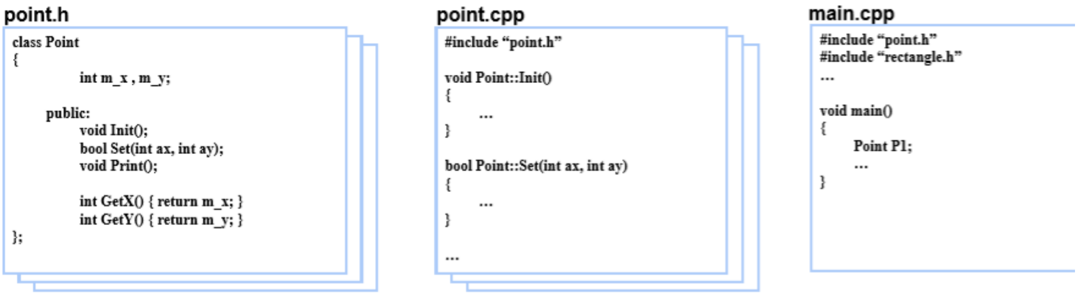


```
// inside Rectangle class
 void Set(Point & topLeft, Point & bottomRight){
    m_TopLeft = topLeft;
    m_BottomRight = bottomRight;
 }
```

Global function – can't be assigned to any specific class.

This* pointer – Return the object as return value.

Working with multi files we need to use with :: For example :



point.h

```
class Point
{
        int m_x , m_y;

    public:
        void Init();
        bool Set(int ax, int ay);
        void Print();

        int GetX() { return m_x; }
        int GetY() { return m_y; }
};
```

point.cpp

```
#include "point.h"

void Point::Init()
{
        ...
}

bool Point::Set(int ax, int ay)
{
        ...
}

...
```

main.cpp

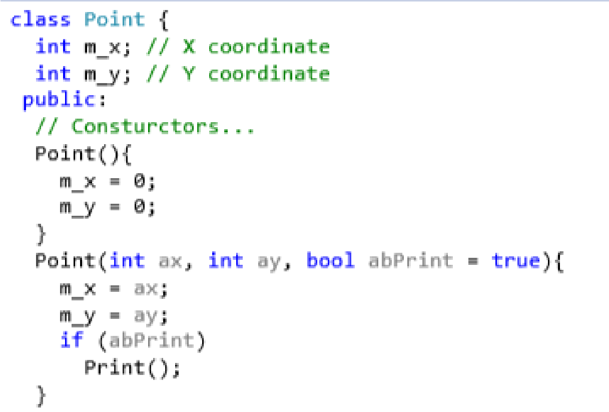
```
#include "point.h"
#include "rectangle.h"
...

void main()
{
        Point P1;
        ...
}
```

2) Ctors & Dtors

Constructor function: initialize variables, allocate memory, perform any action.

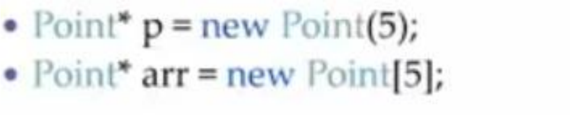
Properties:

- Has no return value and not return void!
- This is an automatic function, not used by us
- If something went wrong, the object won't be built



```
class Point {
  int m_x; // X coordinate
  int m_y; // Y coordinate
 public:
  // Consturctors...
  Point(){
    m_x = 0;
    m_y = 0;
  }
  Point(int ax, int ay, bool abPrint = true){
    m_x = ax;
    m_y = ay;
    if (abPrint)
      Print();
  }
```

The first constructor is initialize to zero or null constructor. The second one is getting values from outside.

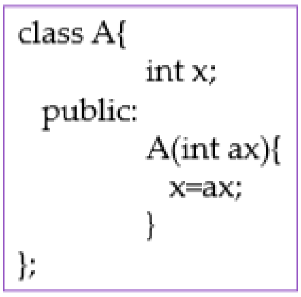


```
• Point* p = new Point(5);
• Point* arr = new Point[5];
```

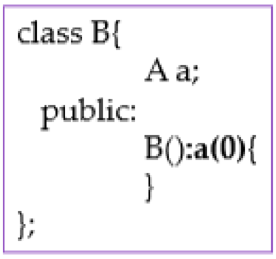
In the first case : Initialize by not default constructor that given one parameter.

In the second case: pointer to allocation array size 5. Must be default constructor.

Initialize line: When the object depend on another object, we need to call his constructor, SO we use in Initialize line.



```
class A{
          int x;
  public:
          A(int ax){
            x=ax;
          }
};
```

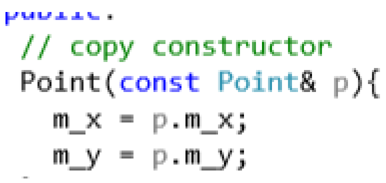


```
class B{
          A a;
  public:
          B():a(0){
          }
};
```

Destructor function: any actions we wish to do before the object is terminated.

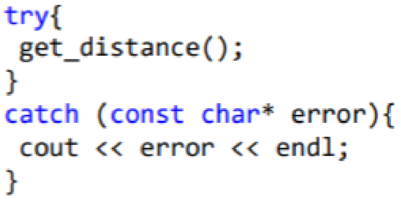
The name is similar to the name of the class with the prefix ~ and defined as public. This function can not return value and doesn't get any argument.

Copy constructor: gets the same type of objects by reference as an argument.



```
public:
 // copy constructor
 Point(const Point& p){
   m_x = p.m_x;
   m_y = p.m_y;
```

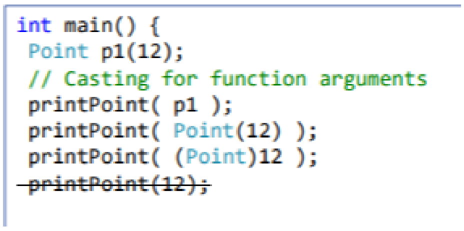
3) Exceptions



```
try{
 get_distance();
}
catch (const char* error){
 cout << error << endl;
}
```

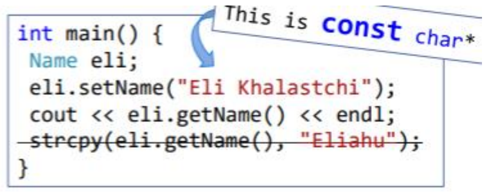
A try block followed by catch block that handles the exception.
An exception can throw again using throw command.

Auto casting :



```
int main() {
 Point p1(12);
 // Casting for function arguments
 printPoint( p1 );
 printPoint( Point(12) );
 printPoint( (Point)12 );
~~printPoint(12);~~
```

Const :



```
int main() {
 Name eli;
 eli.setName("Eli Khalastchi");
 cout << eli.getName() << endl;
~~strcpy(eli.getName(), "Eliahu");~~
}
```

Mutable:

This can be change even if it const variable.

Static:

Can be defined within a function and a class.
The static will not be included in the object size.
It can not be initialize in the initialize line.
If the static variable is private- we must use a public method to access it and the function should be static.

Friendship permissions:

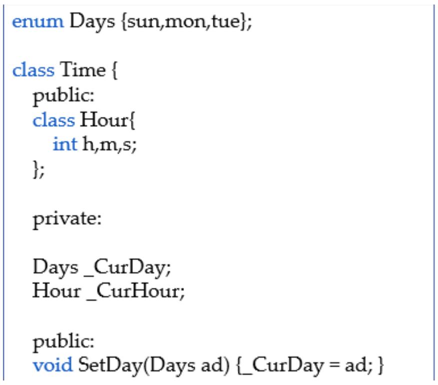
The class can allow certain functions or classes to access its private members using the keyword "friend".

Friends can not be defined as const.

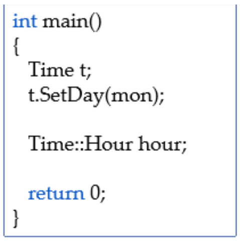
4) Operator overloading

Inner classes/ enums:

a public class can be used anywhere in the application by using its full name.



```
enum Days {sun,mon,tue};

class Time {
   public:
   class Hour{
      int h,m,s;
   };

   private:

   Days _CurDay;
   Hour _CurHour;

   public:
   void SetDay(Days ad) {_CurDay = ad; }
```

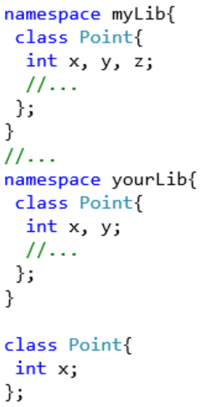


```
int main()
{
   Time t;
   t.SetDay(mon);

   Time::Hour hour;

   return 0;
}
```

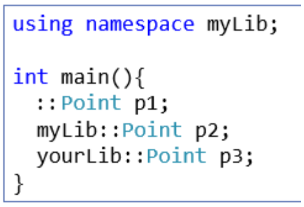
Namespaces:

Bigger than class .

#include "myLib.h". using namespace myLib

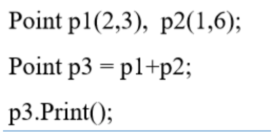


```
namespace myLib{
 class Point{
  int x, y, z;
  //...
 };
}
//...
namespace yourLib{
 class Point{
  int x, y;
  //...
 };
}

class Point{
 int x;
};
```



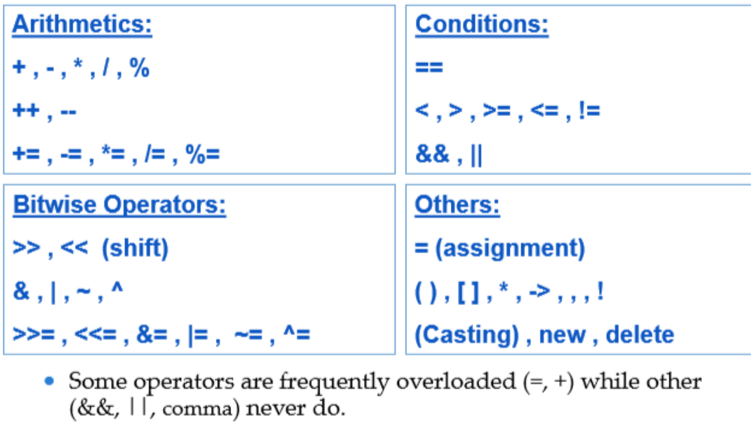
```
using namespace myLib;

int main(){
  ::Point p1;
  myLib::Point p2;
  yourLib::Point p3;
}
```

Operator overloading:



```
Point p1(2,3),  p2(1,6);

Point p3 = p1+p2;

p3.Print();
```

So far we have managed to show analogous behavior between fundamental types (variables) and classes (objects). Now we want to add the option to overloading operators.



Arithmetics:

+ , - , * , / , %

++ , --

+= , -= , *= , /= , %=

Conditions:

==

< , > , >= , <= , !=

&& , ||

Bitwise Operators:

>> , << (shift)

& , | , ~ , ^

>>= , <<= , &= , |= , ~= , ^=

Others:

= (assignment)

( ) , [ ] , * , -> , , , !

(Casting) , new , delete

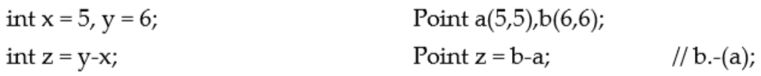
- Some operators are frequently overloaded (=, +) while other (&&, | |, comma) never do.

- An operator should be overloaded only when its behavior is obvious, Otherwise, a function should be used.
- Object parameters will be passed By Reference.
- Object parameters that should not be changed - will be defined as const.

Unary operation:

```
int x = 5;                     Point a(5,5);
int y = -x;                    Point b = -a; // -5,-5;
```

Binary operation:



```
int x = 5, y = 6;              Point a(5,5),b(6,6);
int z = y-x;                   Point z = b-a;          // b.-(a);
```

5) **Inheritance:**

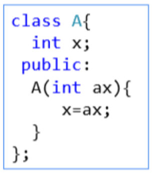
We may be unable to change the original base, There are similar entities (with a common base) we wish to represent.
The resulting code is easier to read and maintain and does not include redundant code.
All the data members of the base class also exist in the derived class.
They will still be included in the derived object size.

When won’t we be able to access the base class members:

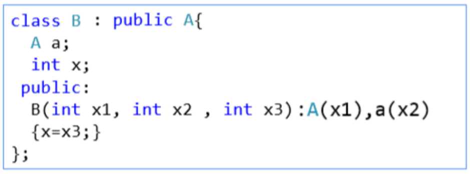
- Hiding – The derived class has defined a data member with the same name, We can still access the hidden data member using its full name (base_class::member)
- Private Permissions – Any private member of the base class cannot be accessed in the derived class Note: the private data member is still included in the derived class and can be accessed using a public method of the base class

C’tor and D’tor are not inherited.
The assignment operator is not inherited.

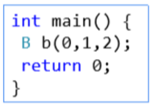
We must use an Initialization Line.



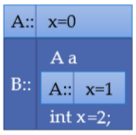
```
class A{
  int x;
 public:
  A(int ax){
     x=ax;
  }
};
```



```
class B : public A{
  A a;
  int x;
 public:
  B(int x1, int x2 , int x3):A(x1),a(x2)
  {x=x3;}
};
```



```
int main() {
 B b(0,1,2);
 return 0;
}
```

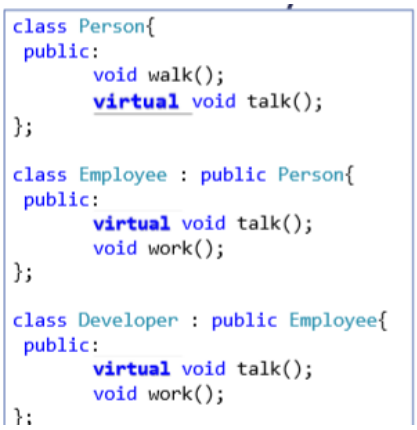


6) Polymorphism:

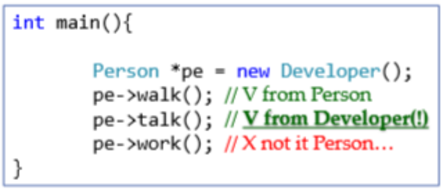
a pointer to the base class can also point to objects from a derived class, so we need to use polymorphism.

Identifying the type of the object we manipulate and then Associating the memory address of functions / variables.

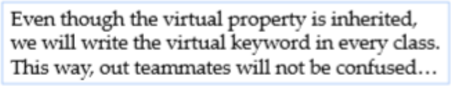
In C++ binding is static unless instructed otherwise By using the ‘virtual’ keyword.



```
class Person{
 public:
        void walk();
        virtual void talk();
};

class Employee : public Person{
 public:
        virtual void talk();
        void work();
};

class Developer : public Employee{
 public:
        virtual void talk();
        void work();
};
```



```
int main(){

        Person *pe = new Developer();
        pe->walk(); // V from Person
        pe->talk(); // V from Developer(!)
        pe->work(); // X not it Person…
}
```

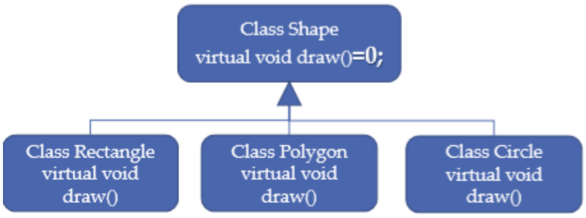


Even though the virtual property is inherited, we will write the virtual keyword in every class. This way, out teammates will not be confused…

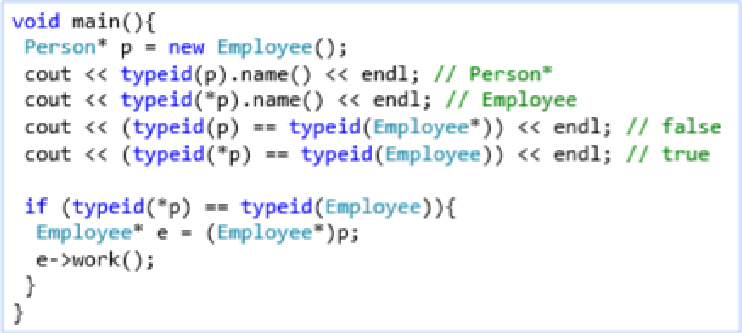
Abstract:

Do we need an implementation for draw() in Shape? No.

We can make these methods pure . Instead of m(){} we write m()=0; And thus the class is automatically an Abstract class . No one can make instances of the class . Or pass it by value…



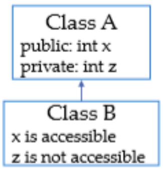
Type Id:

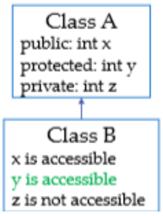


```
void main(){
 Person* p = new Employee();
 cout << typeid(p).name() << endl; // Person*
 cout << typeid(*p).name() << endl; // Employee
 cout << (typeid(p) == typeid(Employee*)) << endl; // false
 cout << (typeid(*p) == typeid(Employee)) << endl; // true

 if (typeid(*p) == typeid(Employee)){
  Employee* e = (Employee*)p;
  e->work();
 }
}
```

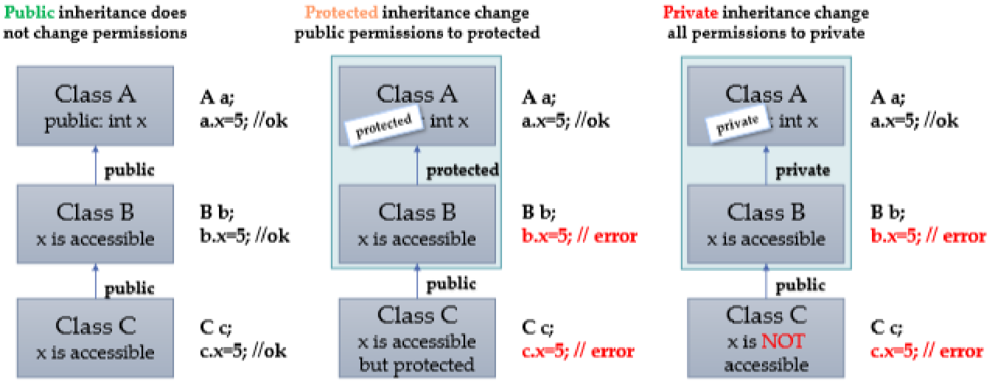
7) Multiple inheritance:

Permissions:

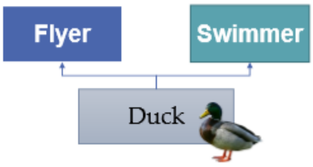




- A public member is accessible- in class, inheritance hierarchy, From an object of the class.
- A private member is only accessible- within its class, Not within the inheritance hierarchy, Not from an object of the class.
- A protected member is accessible- Within its class ,Within the inheritance hierarchy , Not from an object of the class.

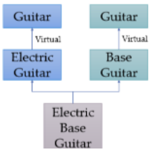


Multiple inheritance:



```
class Duck : public Flyer, public Swimmer
```

Common ancestor:



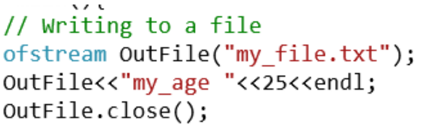
```
class ElectricGuitar : virtual public Guitar {…};
class BaseGuitar : virtual public Guitar {…};
class ElectricBaseGuitar : public ElectricGuitar, public BaseGuitar {…};
```

8) Files:

A file is a set of blocks of related data. Basically, an array of bytes, representing whatever we need.

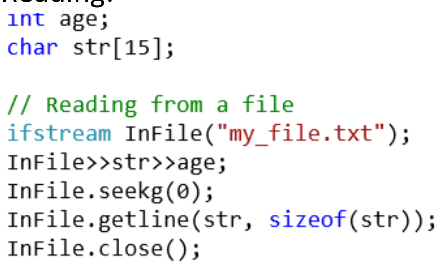
Txt:

Writing :



```
// Writing to a file
ofstream OutFile("my_file.txt");
OutFile<<"my_age "<<25<<endl;
OutFile.close();
```

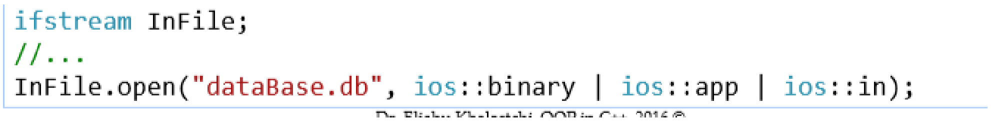
Reading:



```
int age;
char str[15];

// Reading from a file
ifstream InFile("my_file.txt");
InFile>>str>>age;
InFile.seekg(0);
InFile.getline(str, sizeof(str));
InFile.close();
```

- seekg() moves the reading marker
- seekp() moves the writer marker
- tellg() / tellp() tells us where are the markers
- getline() reads until ‘\n’ or buffer size

opening file:



```
ifstream InFile;
//...
InFile.open("dataBase.db", ios::binary | ios::app | ios::in);
```

Dr. Eliahu Khalastchi OOP in C++ 2016 ©

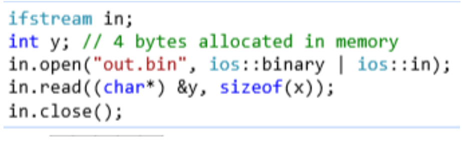
open(const char* filename, int mode);

mode – how to open (one or more of the following – using |)

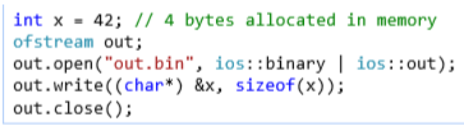
- ios::app – append
- ios::in / ios::out – (the defaults of ifstream and ofstream)
- ios:nocreate / ios::noreplace – open only if the file exists / doesn’t exist
- ios::trunc – open an empty file
- ios::binary – open a binary file (default is textual)

Bin:

Read and write :



```
ifstream in;
int y; // 4 bytes allocated in memory
in.open("out.bin", ios::binary | ios::in);
in.read((char*) &y, sizeof(x));
in.close();
```

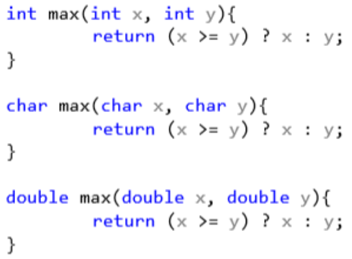


```
int x = 42; // 4 bytes allocated in memory
ofstream out;
out.open("out.bin", ios::binary | ios::out);
out.write((char*) &x, sizeof(x));
out.close();
```

9) templates:

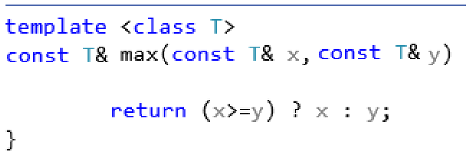
When we need to repeat on the same code.

Instead of :



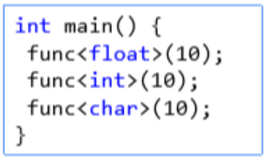
```
int max(int x, int y){
        return (x >= y) ? x : y;
}

char max(char x, char y){
        return (x >= y) ? x : y;
}

double max(double x, double y){
        return (x >= y) ? x : y;
}
```

Do :



```
template <class T>
const T& max(const T& x, const T& y)

        return (x>=y) ? x : y;
}
```

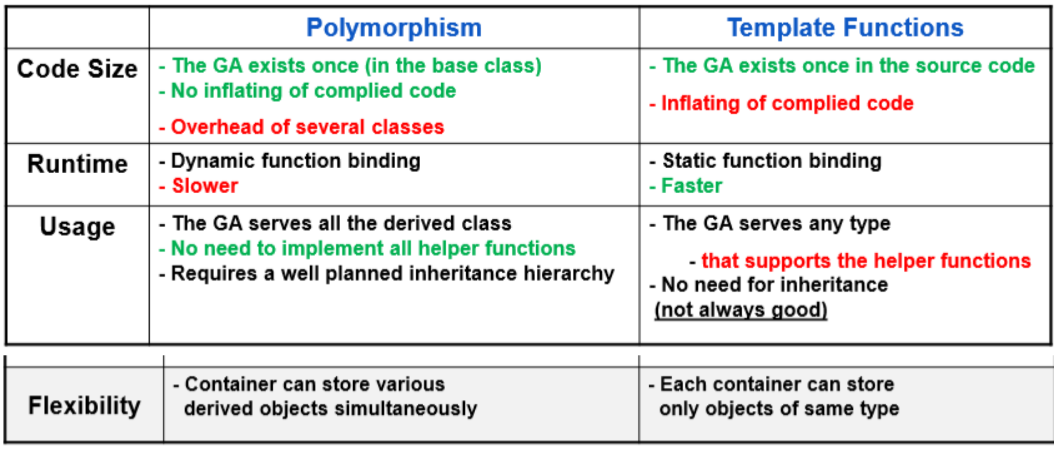
The compiler must be able to deduce the type of T. Otherwise, we should tell it!



```
int main() {
 func<float>(10);
 func<int>(10);
 func<char>(10);
}
```

Polymorphic generic algorithm:

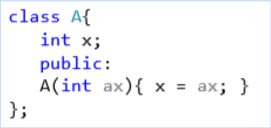
The code should appears only once in the code, but it has 2 different implementations.



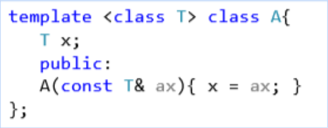
	Polymorphism	Template Functions
Code Size	- The GA exists once (in the base class) - No inflating of complied code - Overhead of several classes	- The GA exists once in the source code - Inflating of complied code
Runtime	- Dynamic function binding - Slower	- Static function binding - Faster
Usage	- The GA serves all the derived class - No need to implement all helper functions - Requires a well planned inheritance hierarchy	- The GA serves any type - that supports the helper functions - No need for inheritance (not always good)
Flexibility	- Container can store various derived objects simultaneously	- Each container can store only objects of same type

10) Creating a standard template library (STL)

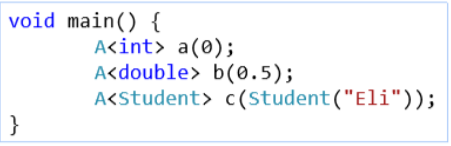
Like the template object we can use it for the template class.



```
class A{
   int x;
   public:
   A(int ax){ x = ax; }
};
```



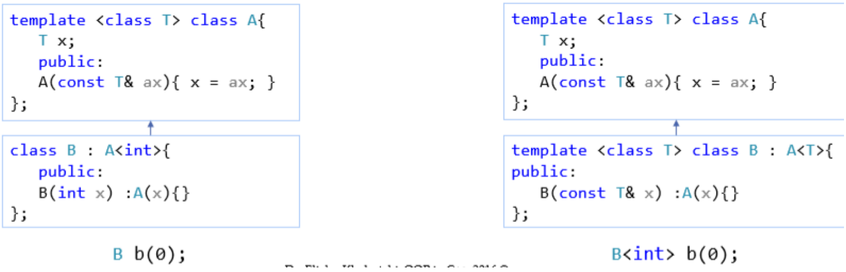
```
template <class T> class A{
   T x;
   public:
   A(const T& ax){ x = ax; }
};
```



```
void main() {
        A<int> a(0);
        A<double> b(0.5);
        A<Student> c(Student("Eli"));
}
```

Inheriting template classes :

By defining the type and By being an undefined template class.



```
template <class T> class A{
   T x;
   public:
   A(const T& ax){ x = ax; }
};

class B : A<int>{
   public:
   B(int x) :A(x){}
};

            B b(0);
```

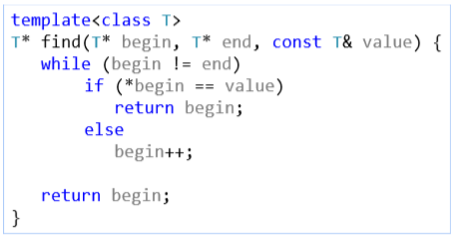
```
template <class T> class A{
   T x;
   public:
   A(const T& ax){ x = ax; }
};

template <class T> class B : A<T>{
public:
   B(const T& x) :A(x){}
};

            B<int> b(0);
```

This template function works only on arrays.

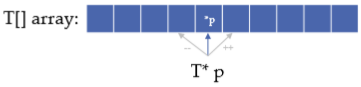
Regardless of T, T* is always a pointer

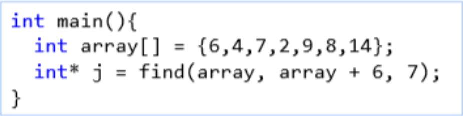
A pointer is an Iterator for arrays:

- ++ moves the pointer forwards
- -- moves the pointer backwards
- * returns (an object of) T
- ==, != compares memory addresses



```
template<class T>
T* find(T* begin, T* end, const T& value) {
   while (begin != end)
      if (*begin == value)
         return begin;
      else
         begin++;

   return begin;
}
```





```
int main(){
  int array[] = {6,4,7,2,9,8,14};
  int* j = find(array, array + 6, 7);
}
```

Iterators:

Each Data Structure must supply an Iterator □ An Iterator relates to the data structure.

As a pointer relates to an array.

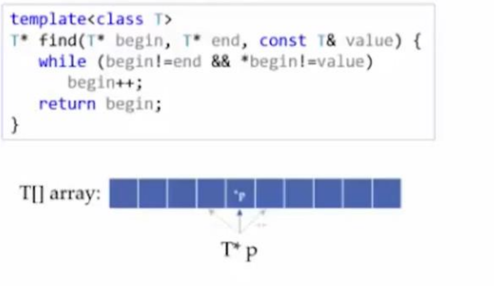
An Iterator must implement these operators: □ ++, --, *, ==, !=

Gives us the opportunity to move about list or array.

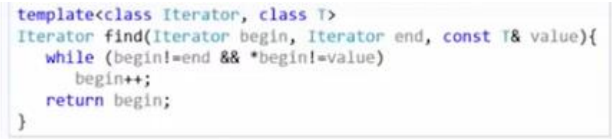
Gives us the opportunity to be not depend on the data structure.

Linked list example:

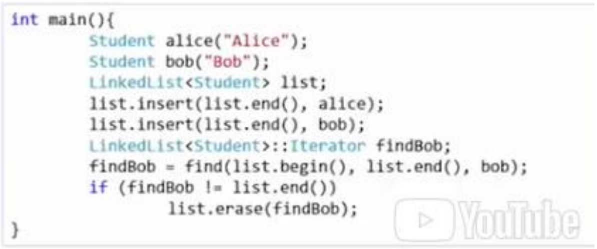
A double-linked list capable storing every type (T) and provide an Iterator to traverse it.



```
template<class T>
T* find(T* begin, T* end, const T& value) {
   while (begin!=end && *begin!=value)
      begin++;
   return begin;
}
```



```
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& value){
   while (begin!=end && *begin!=value)
      begin++;
   return begin;
}
```



```
int main(){
          Student alice("Alice");
          Student bob("Bob");
          LinkedList<Student> list;
          list.insert(list.end(), alice);
          list.insert(list.end(), bob);
          LinkedList<Student>::Iterator findBob;
          findBob = find(list.begin(), list.end(), bob);
          if (findBob != list.end())
                    list.erase(findBob);
}
```

Object functions:

- We want to pass a function as a parameter to another function

Template<class iterator, class operation>

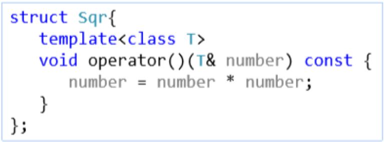
Void apply(iterator begin , iterator end, condt operation &function)

{

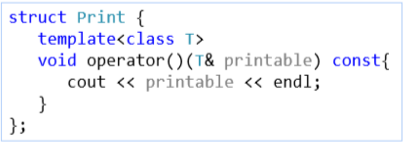
For(begin !=end; ++begin);

Function(*begin);

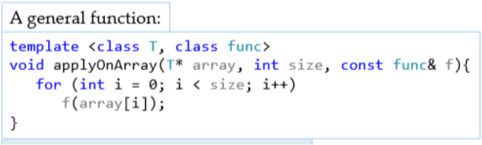
}



```
struct Sqr{
   template<class T>
   void operator()(T& number) const {
      number = number * number;
   }
};
```

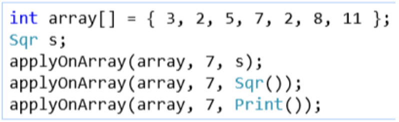


```
struct Print {
   template<class T>
   void operator()(T& printable) const{
      cout << printable << endl;
   }
};
```



A general function:

```
template <class T, class func>
void applyOnArray(T* array, int size, const func& f){
   for (int i = 0; i < size; i++)
      f(array[i]);
}
```



```
int array[] = { 3, 2, 5, 7, 2, 8, 11 };
Sqr s;
applyOnArray(array, 7, s);
applyOnArray(array, 7, Sqr());
applyOnArray(array, 7, Print());
```

1) Import on operator from class type of the function.
   class print;
   print MyPri;
2) Send: 'MyPri' as a function.

11) STL , object function , lambda experssions:

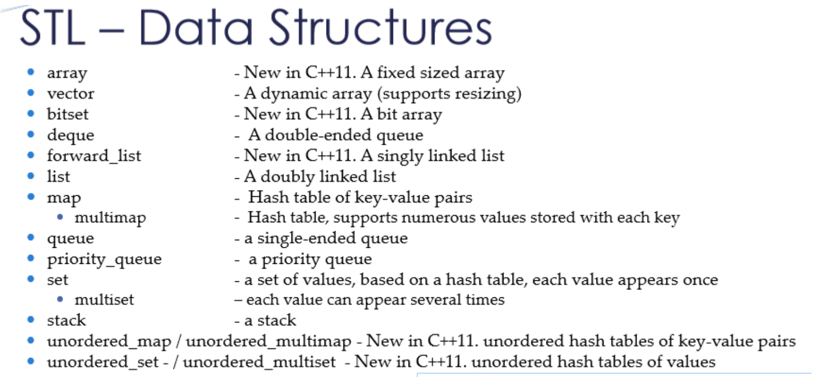
The STL is filled with generic containers, generic functions, and object functions Now, when you know how to build, STL style, you may use the STL library.

STL contains generic data structures like the LinkedList we have built

Now, you are allowed to use the things STL really includes

In general STL includes the following components:

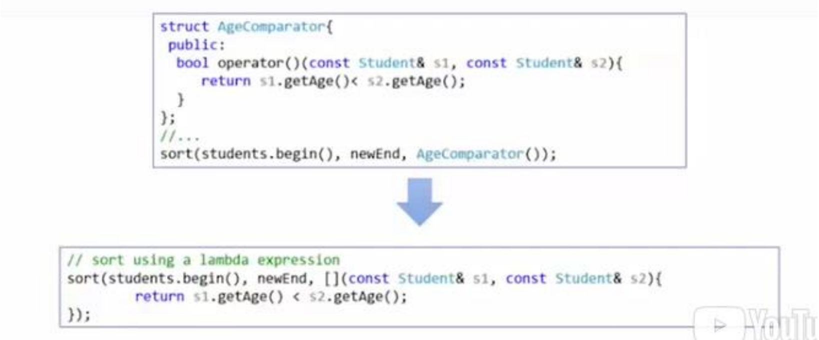
- Data Structures – (vector, list, set, …)
- Generic Algorithms – (for each, find, sort, …)
- Object Functions



STL – Data Structures

array	- New in C++11. A fixed sized array
vector	- A dynamic array (supports resizing)
bitset	- New in C++11. A bit array
deque	- A double-ended queue
forward_list	- New in C++11. A singly linked list
list	- A doubly linked list
map	- Hash table of key-value pairs
multimap	- Hash table, supports numerous values stored with each key
queue	- a single-ended queue
priority_queue	- a priority queue
set	- a set of values, based on a hash table, each value appears once
multiset	– each value can appear several times
stack	- a stack
unordered_map / unordered_multimap	- New in C++11. unordered hash tables of key-value pairs
unordered_set - / unordered_multiset	- New in C++11. unordered hash tables of values

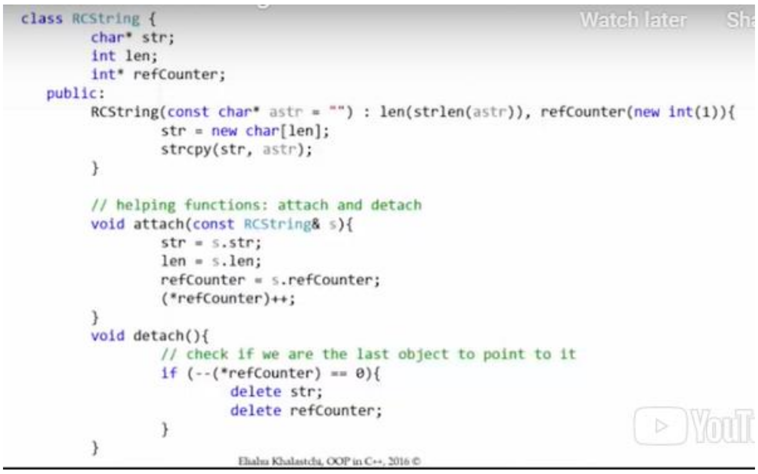
Lambda experssions:



```
struct AgeComparator{
 public:
  bool operator()(const Student& s1, const Student& s2){
     return s1.getAge()< s2.getAge();
  }
};
//...
sort(students.begin(), newEnd, AgeComparator());
```

```
// sort using a lambda expression
sort(students.begin(), newEnd, [](const Student& s1, const Student& s2){
        return s1.getAge() < s2.getAge();
});
```

12) Reference counting:



```
class RCString {
        char* str;
        int len;
        int* refCounter;
    public:
        RCString(const char* astr = "") : len(strlen(astr)), refCounter(new int(1)){
                str = new char[len];
                strcpy(str, astr);
        }

        // helping functions: attach and detach
        void attach(const RCString& s){
                str = s.str;
                len = s.len;
                refCounter = s.refCounter;
                (*refCounter)++;
        }
        void detach(){
                // check if we are the last object to point to it
                if (--(*refCounter) == 0){
                        delete str;
                        delete refCounter;
                }
        }
```

Elahu Khalastchi, OOP in C++, 2016 ©

הערות ממטלות:

types:

```
auto typeId = typeid((dogs[i])).name();
```

1. typeID(*arr[i]).name ------ מחזיר את הצאצא
2. typeID(arr[i]).name ------ מחזיר את האבא
3. typeID(x)==typeID(y) השוואה בין שני טיפוסים
4. B *b = dynamic_Cast<B*>(element); כך נבדוק את סוג האבא אם זה מלא בתוכן ולא נאל, ההמרה הצליחה.

```
T* t = dynamic_cast<T*>(array[i]);
```

iterators:

1. קודם לקדם את האלמנט ורק אז למחוק אותו.
2. lst.unique(); אחראי על שכל איבר יופיע פעם אחת בלבד
3. List<type>::iterator it = node,begin()
4. While(it!=node.end()&&(*it)->id!=id) {it++}
5. Node *temp = new Node(id); nodes.pushback(temp) הוספת איבר חדש
6. להגדיר טיפוס חדש לקומפיילר אם לא מזהה את האיטרטור Typename list<T>::iterator it
7. For_each הופך את הקוד לקצר זה נראה ככה במבנה הזה: for_each(begin,end,[&out](employee *e){e=>print(out);});

```
8.  template <class V>
9.  class Property {
10.         V value;
11.         list<Property<V>*> boundToMe;
```

Inheritance:

1. לא לשכוח להוסיף public למחלקה שירשנו ממנה.
2. חייב לממש פונקציית Virtual , יש חשיבות לסדר של המחלקות.
3. אם הולך לדיסטרקטור כל שניה, לשים לב אם קיבל מיקום. להוסיף לפונקציה הקודמת & ולהחזיר בערך החזרה *this
4. תמיד שזה get מחזירים במבנה הבא: get() const{}
5. אם יש מחלקה רביעית שיורשת מ2,3,1 יש להוסיף לראשונה virtual
6. Constructor לא תלוי בשום פונקציה אחרת.

```
class UnderGradStudent :public virtual  Student

class DirectStudent :public GradStudent, public UnderGradStudent {
```

7. ומוסיפים רק את הstudent לפעולה הבונה.

Files:

1. אם נדרש ממני לגלות גודל כלשהו מקובץ, עדיף לשמור אותו לפני בפעולת כתיבה.
2. לא לשכוח id אינדיקטיבי אם נדרש אחד כזה.
3. כאשר דורשים ממנו לטעון לתוך מערך, ראשית אבדוק את הטיפוס , ואקצה מספיק זיכרון.

Operators:

1. אם האופרטור לא בתוך המחלקה, יש לקבל את שני המשתנים

טיפים למבחן :

קודם להסתכל מה דורשים ממני בmain .

דוגמא להעברת פונקציה דרך פונקציה :

```
map.forEachValue([](const Student& s) {  cout << s.getName(); });

        template <class function>
        void forEachValue( function f) {}
```

דוגמא לכתיבה וקריאה של char בקובץ :

```
                virtual void save(ostream& out) const {
                out.write((char*)&this->color, sizeof(this->color));
        }
        virtual void load(ifstream& in) {
                in.read((char*)&this->color, sizeof(this->color));
        }
```

הכנסה לעץ בינארי :

```
void insert(Num number) {
                if (size == 0) {
                        root->number = number;
                }
                else {
                        Item* i = root;
                        while (i->left != NULL && i->right != NULL) {
                                while (number < i->number && i->left != NULL) i =
i->left;
                                while (number > i->number&& i->right != NULL) i =
i->right;
                        }
                        Item* t = new Item();
                        t->number = number;
                        if (number < i->number) i->left = t;
                        if (number > i->number) i->right = t;
                }
                size++;
```

מחיקה מעץ בינארי :

```
void deleteSubTree(Num number) {
                auto newRoot = root;
                Item* i = root;
                while (true)
                {
                        if (newRoot == nullptr)
                        {
                                break;
                        }
                        else
                        {
                                if (number < newRoot->number)
                                {
                                    newRoot = newRoot->left;
                                    i = i->left;
                                }
                                else if (number > newRoot->number)
                                {
                                    i = i->right;
                                    newRoot = newRoot->right;
                                }
                                else if (number == newRoot->number)
                                {
                                    delete i;
                                    break;
```

כתיבה וקריאה של רשימת אוביקטים :

```
void saveDogs(Dog** dogs, int size, ofstream& out) {
    out << size << endl;
    for (int i = 0; i < size; i++) {
        out << typeid(*dogs[i]).name() << endl;
        dogs[i]->print(out);

void loadDogs(Dog**& dogs, int& size, ifstream& in) {
    in >> size;
    dogs = new Dog * [size];
    char type[20], someText[20];
    for (int i = 0; i < size; i++) {
        in >> type;
        int id;
        float weight;
        bool property;
        in >> someText;
        in >> id;
        in >> someText;
        in >> weight;
        in >> someText;
        in >> property;
        if (type[0] == '7')
            dogs[i] = new Bulldog(id, weight, property);
        if (type[0] == '6')
            dogs[i] = new Poodle(id, weight, property);
        if (type[0] == '1') {
            bool intelligent;
            in >> someText;
            in >> intelligent;
            Bulldog* b = new Bulldog(id, weight, property);
            Poodle* p = new Poodle(id, weight, intelligent);
            dogs[i] = new BulldogPoodle(*b, *p);
```

קריאה מתוך פונקציה :

```
    Manager& addEmployee(Employee* newE) {
        lstEmpl.push_back(newE);
        return *this;
    }

  m2->addEmployee(new Engineer(3,1)).addEmployee(new Engineer(4,2));
```