

C++

Summer course 2009 slide-guide

Short notes about the lecture slides, by Nerya Or.

Lectures+slides were given by Moti Freiman and Ofir Pele.

In this document are some of the things which in my opinion are important in the lecture slides. (+Extra tips in the end).

The idea is that this document is a sort of “table of contents” for the actual slides, so it’s best to print the slides as well.

Some of the contents may be inaccurate/wrong, so please don’t blame me if something goes bad...

This document is not officially endorsed by the course staff, use at your own responsibility.

Good luck in the exam!

Version 1.01

1 Class/Tirgul 1

1.1 Class 1

Slide 39:

Note: If we implement any constructor at all (no matter what signature it has), then the default constructor will no longer exist.

Slide 51:

Freeing an allocated **array** of pointers to objects **on the heap** -

```
int n=4;
for(int i=0; i<n; i++) {
    delete(arr[i]); //Delete each array member
}
delete [] arr; //Free the memory allocated for the array of pointers.
```

1.2 Tirgul 1

Class note: (not in the slides)

```
Point p; //Call to default constructor and create Point object.
Point p(); //Declaration of a function!! Not calling any constructor here...
```

Slides 11 - 19:

References.

Note that once a reference has been assigned to a variable, that reference can never “point” at a different location.

Also, a reference may only refer to “legitimate” objects - things on the heap/stack, but **not** NULL, for example.

Slides 21+:

`static` class members.

We can define static class members, which will behave like in Java: They belong to the class, and aren't associated to any specific class instances.

This is done as follows (from ex2):

File: PhoneCompany.h:

```
:
class PhoneCompany {
    private:
        static int _totalIncomesAllCompany;
:
};
```

File: PhoneCompany.cpp:

```
#include "PhoneCompany.h"
#include <otherThings...>
#define STUFF
int PhoneCompany::_totalIncomesAllCompany = 0; //Init the static data member.
: //The rest of the file...
```

By initializing the static member in the .cpp file as opposed to the .h file, we avoid a different definition in different .o files.

2 Class/Tirgul 2

2.1 Class 2

Slides 1 - 8:

Initialization list. Explanation and examples.

In slide 8 we see that we can initialize a pointer to heap-allocated objects, using the syntax: “B::B(int i) : _a1(new A(i)) {...}”.

This is considered to be more safe, since if the allocation of `_a1` fails, then we get that B's allocation also fails, and that's better than what would have happened in case we've used "new" inside the constructor's body: We would have just stayed with an incomplete B object and we'd have to somehow report back that the allocation of it has failed, from within the constructor.

Slide 13:

Never return a reference to a local (stack) variable! Its lifetime will be over once the stack folds down...

Slides 15 - 20:

`const`.

About `const` functions: We can have 2 versions for a function: A `const` version, and a non-`const` version.

The compiler automatically chooses the correct version for the object we're using - a `const` object will call the `const` version of a function, and a non-`const` object will call the non-`const` function.

A non-`const` object may use a `const` function, but a `const` object may not use a non-`const` function!

Slide 19: Note that if an object has pointers to heap-allocated objects inside it, then only the pointers will have `const` apply to them, if the 'father' object is `const`. This means that the actual heap-allocated objects that we point at are not affected by `const`.

Slide 20: `const` objects with references. If a `const` object has reference data-members, then the actual contents of the reference are not `const`, but just the reference itself. **This means we can change them.**

Slide 24:

`protected`. Class members that should be accessible by subclasses are declared `protected`.

Slide 30:

new and delete are global operators.

If we have class `Parent` and class `Derived` that inherits from `Parent`, and we set `Parent`'s c'tor as `protected`, and we also want to have a (`Parent*`) data-member inside `Derived`, we will **not** be able to call "new" in `Derived`'s initialization list, since `new` is a global operator, so the `protected`-ness of `Parent`'s c'tor is applied to it, and thus `new` can't "see" this c'tor.

(Same goes for `protected` d'tors and `delete`).

Slides 57 - 58:

Destructors and `virtual`. (Slide 57: "Which destructor is called?" answer: `Base`'s destructor, because the d'tor wasn't defined as `virtual` in this example).

Note: once we define a `virtual` d'tor, the derived classes **must** declare their own destructor! (It says so in the slides, but in reality it can compile and run, but we might get leaks if we're not careful).

Slide 64: Virtual methods - tips:

- If you have `virtual` methods in a class, always declare its destructor `virtual`.
- Never call `virtual` methods during construction and destruction. (Why? In c'tors, we first call the parent class c'tor. At first, the vtbl of our object is that of the parent class. So any calls to `virtual` classes will be directed to the parent vtbl, and not the derived class vtbl. Only later when we call the derived class c'tor, will the vtbl be changed to the derived class vtbl).

- Use pure virtual classes to define interfaces.
- (more in the slide)

2.2 Tirgul 2

-Nothing of particular interest. -

3 Class/Tirgul 3

3.1 Class 3

Slides 2 - 14:

```
operator=
```

Slides 16 - 20:

Copy-constructors.

Note: The copy-c'tor is also called when initializing a new object -

```
Mystring str2 = str1; //Copy constructor called here!
```

is equivalent to writing:

```
MyString str2(str1);
```

Note the example on slides 27-28, in which we see that it's important to call the copy-c'tor of datamembers, in the copy c'tor of a class.

Slides 30 - 36:

Tailored casts...

Slide 36:

The dangers of implicit casting - important example.

Slide 37:

`explicit`: a directive to the compiler to not use constructor in implicit casts.

For example:

```
class IntArray {
public:
    explicit IntArray(int size); //Constructor
    :
};
```

This means that the c'tor can only be called explicitly, meaning, only if the programmer has asked for it - don't do tailored casts.

3.2 Tirgul 3

Slide 7:

There exist operators for casting to primitive types. For example:

```
class Fraction {
    :
    //Fraction → double conversion
    operator double() const {
        :
    }
    :
}
```

Slides 8 - 12

friends.

We can also create **friend classes**. These are classes that have **friend** access to our current class. Friendship is one-sided.

Slides 13+ :

inline.

4 Class/Tirgul 4

4.1 Class 4

Slides 8 - 21:

Templates. Basic info and examples.

Slides 21 - 30:

Some basic info about iterators and copying lists using them.

Slides 31 - 37

Template variations - `<class T1,class T2>`,`<class T,int Size>`,`<class T, T val>`, setting up default template behavior...

Note: if we have something like:

```
template<class T,int Size=1024> class Buffer {...};
```

Then, `Buffer<char>` and `Buffer<char,1024>` are **the same type**. The compiler is smart enough to realize that.

Slides 38 - 40:

Template specialization. (Defining specific behavior for a certain type, in a template).

Slides 41 - 44:

typename. Why we need it, and some examples.

4.2 Tirgul 4

General note:

```
int(); //Returns a const literal of 0.  
double(); //Returns a const literal of 0.0  
... and so on, for primitives.
```

This is **not** a default constructor, because just declaring an 'int' will result in grabage, and not 0.

Pg. 2:

STL containers hold **copies** of elements, and assume elements have copy-c'tor, and operator=.

Pg. 3 - 4:

```
std::vector.  
Note about the first slide in page 4:  
    std::vector<Fred> vec(10,Fred(5,7));  
    //Takes one Fred object, and then copies it 10 times.  
    //Does not perform initialization 10 times.
```

Another general note:

There's a problem with things like vectors and polymorphism. If we have class Base and Derived, and we create a `vector<Base>` and fill it with Derived objects, we will get what's called 'slicing', under certain circumstances. That is, only the Base part of our Derived object will be used or stored. This is bad.

5 Class/Tirgul 5

5.1 Class 5

Slides 7 - 10:

Comparators (using templates).

Slides 11 - 15:

Function objects ('functors').

`unary_function` and `binary_function` define a sort of interface for function objects. (More about it in slides 42-45).

Slides 16 - 39:

Iterators.

Note about slide 24: In a "unique **sorted** associate container", the command "`c.insert(pos,x);`" inserts x into the set, with pos as a **hint** to where it will be inserted. This means, that we don't actually affect where the element x will be placed (since it's a sorted container), but we can help out the data structure in giving a clue as to where to start searching for its final position.

Slides 40 - 45:

Adaptors. Also some more info about function object interfaces (`unary_function`, `binary_function`...).

Slides 46 - 57:

Algorithms.

Note about slide 51-52: We get an error when trying to perform `copy`, since `copy` doesn't do 'insert', but it actually performs assignment on the elements.

Another note: We must pay attention to which iterators we are using. Some algorithms require 'random access' iterators, and for example, `list` iterators are bidirectional, and not random access. This is why we get an error in slides 59 - 63.

Slides 65 - 75:

Streams.

Slides 76 +:

Examples about binary files...

5.2 Tirgul 5

Slides 2 - 5:

`static_cast`, `const_cast`, `reinterpret_cast`.

`static_cast` is meant to convert **values** only. (won't cast `int*` to `float*`, but will cast `int` to `float`).

`const_cast` adds/removes `const`-ness. Usually it's a bad idea to use this, as it probably means your design is flawed.

`reinterpret_cast` is used to explicitly cast things, byte after byte. It circumvents C++'s type checking, and is basically brutally casting one thing to another. Can be dangerous!

Slides 6 - 15:

RTTI (Run Time Type Information): `dynamic_cast`, and `typeid`.

`dynamic_cast` can only be used on **pointer** or **reference** types. Also, can **only** be used on types with **virtual functions!** (Slides 11-12).

(It **will not compile** if we try to use `dynamic_cast` on a non-polymorphic type; a type with no virtual functions).

`typeid`: Similar to Java's `instanceof()`. It returns an object of the `type_info` class. (Slide 14).

Don't misuse RTTI! Remember OOP concepts and polymorphism...

6 Class 6

6.1 Class 6

Slides 3 - 10:

Exceptions.

- Catching by **reference** ("`catch (CollectionE &e){...}`") enables **polymorphism**.

- The syntax “`catch(...) {doStuff;}`” allows us to catch **anything**. We can’t do anything with what we’ve caught, though, since we don’t know what it is...
- Declaring thrown exceptions in a function: The syntax is “`int foo() throw (BadArg, int*, stuff);`”. A function that doesn’t throw anything: “`int foo() throw();`”.

Slides 11 - 13, and 32 - 37:

`auto_ptr`.
 If `p` is an `auto_ptr`, then `p.release()` causes `p` to stop pointing at its target, and returns its value. Operator= of `auto_ptr`s transfers the ownership from one `auto_ptr` to another (calls `release()` internally).
`auto_ptr`s can be dangerous, must be used carefully (tossing it around scopes may end up deleting our variable if we don’t pay attention...).

Slides 14 - 16

Exceptions in constructors.

Slide 17:

In this example, we learn that throwing 2 exceptions simultaneously, completely crashes the program. Avoid it!

Slides 21 - 30:

Smart pointers, introduction (not interesting...).

Slides 32-37:

More on `auto_ptr`. See above ↑.

Slides 38 - 54:

Reference counting, using “clone-on-write”.

7 Class 7

7.1 Class 7

Slides 2 - 7:

Nested classes.

Note: There’s no runtime relation between the nested and the outer class. This means a “child” doesn’t have any connection to his “parent”, unless we specifically hold a “`_parent`” pointer as a data member, or use some other solution like this. (Unlike in Java!).

Slides 8 - 13:

Namespaces.

Note: Don’t use the `using` command in `.h` files, since it could cause problems to other files that include the `.h` file.

Slides 14+:

Performance and optimization.

8 Class 8

8.1 Class 8

No new material was taught in this class, just went over some old material.

9 Other notes

Note 1:

A virtual function in a class takes up the same space as a `void*` pointer, that is, `sizeof(void*)`.

Note 2:

Example of generics and working with iterators:
(Taken from labcpp2007a exam, question 8b).

If we want to create a function that gets two iterators for range (over a sequential container for example), and an extra pointer to the start of an array to insert results from a calculation, as well as something like a comparator, our definition will be as follows:

```
template <class In, class Out, class BinaryPredicate>
void foo(In first, In last, Out out, BinaryPredicate comp) { ... }
We will then be able to write:
std::list<int> l; //Create list.
int out_arr[OUT_ARR_SIZE]; //Output array.
Cmp_class cmp_list; //Create comparator.
//Fill list:
l.push_back(1); l.push_back(2); l.push_back(4); l.push_back(3);
//Call foo.
foo(l.begin(),l.end(),out_arr,cmp_list);
//In=list<int>.iterator, Out = int*, BinaryPredicate = Cmp_class.
```

Note 3:

Operator syntax:

- `T& operator=(const T& other);` (remember to **check** “`this==&other`”!)
- `friend std::ostream& operator<<(std::ostream& os, const T& c);`
- `T (const T& other);` (copy ctor)
- `const T operator-() const;` (unary minus)
- `T& operator+=(const T& other);`
- `friend const T operator+(const T& obj1, const T& obj2);`
- `RetType operator() (Params...);` (Lots of freedom with operator())
- Operator++:
`T& operator++();` //Prefix form: `++x`
`const T operator++(int);` //Postfix form: `x++`. The `int` is a ‘dummy’ argument.

- `T &operator[](size_t i);`
`const T &operator[](size_t i) const;` //Remember to add `const` version!!
- `friend bool operator==(const K& c1, const T& c2);` (binary, global friend version)
`bool operator==(const T& right) const;` (member function. Note: `K==T` will work, but `T==K` won't! This applies in many cases.)

Note 4: Ctor/Dtor order. labcpp2006b:

```
#include <iostream>
class A {
public:
    A() { std::cout << " A ctor" << std::endl; }
    ~A() { std::cout << " A dtor" << std::endl; }
};
class B: public A {
public:
    B() : A() { std::cout << " B ctor" << std::endl; }
    ~B() { std::cout << " B dtor" << std::endl; }
};
class C: public A {
public:
    B _b; C() : _b(), A() { std::cout << " C ctor" << std::endl; }
    ~C() { std::cout << " C dtor" << std::endl; }
};
int main() { C c; }
```

Will produce the printout:

```
A ctor
A ctor
B ctor
C ctor
C dtor
B dtor
A dtor
A dtor
```

Note 5: slabcpp2006b q8: If we want to override some function from STL, we should use the `namespace{...}` block:

```
/* File: swap_ComplicatedContainer.h */
/* Overrides the template <typename Assignable> void std::swap(Assignable& a, Assignable& b) function. */
#ifndef SWAP_COMPLICATEDCONTAINER_H_
#define SWAP_COMPLICATEDCONTAINER_H_
#include "ComplicatedContainer.h"
namespace std {
    template <typename T>
    void swap(ComplicatedContainer<T>& a, ComplicatedContainer<T>& b) {
        a.swap(b);
    }
}
#endif /*SWAP_COMPLICATEDCONTAINER_H_*/
```

Note 6:

If we have a `const` datamember in a class, it must be initialized in the initialization list of the ctor.

Note 7: `virtual` must be declared at the top. The following will only display “B foo” if `A::foo()` is virtual. If `A::foo()` isn’t virtual, we’ll get “A foo” displayed.

```
class A {
public:
    virtual void foo() { cout<<"A foo."<<endl; }
};
class B : public A {
public:
    virtual void foo() { cout<<"B foo."<<endl; }
};
int main() {
    A* bPtr = new B;
    bPtr->foo();
}
```