Generic Programming
SPRING SEMESTER 2021

INSTRUCTOR: NS KUMAR
nskumar@pes.edu

1: Introduction - 11 Jan 2021

Marks allotment

ISA 1 - after 7 weeks - 3 units, 50% of test is lab component, live coding

ISA 2 - after 4 weeks - 2 units, same as above

ESA - project (20 marks) and exam scaled down to 30 marks

All tests and exams are open notes. No assignments.

Need C++ primarily, will not program in Java, C#

Generic programming is a study of data structures and algorithms, but in a generic style.

Project - ...

Books -

C++ templates : Josutis

STL Tutorial and reference : Musser ...

Awesome reference: C++ FAQ (isocpp.org)

2: Paradigms, types - 12 Jan 2021

Paradigm is a way of thinking.

A type is a set of values and operations on them. It does not specify how the operations must happen. A class is a mechanism to make our own type but with the implementation for the operations too. An object is a variable of the class's type.

Interface defines what to do and implementation specifies how to do it.

Types of programming paradigms -

- **Procedural** - defines problems into smaller parts hierarchically and defines routines to solve them. Just contains a series of steps to be carried out.
- **Object based** - supports abstraction, encapsulation and composition
- **Object oriented** - all of object based but also inheritance and polymorphism
- **Generic** - data structures and algorithms work make use of generic types
- **Functional** - a function is a first class citizen, it can be assigned to a variable, passed as an argument, returned from a function. A pure function is independent of its environment and does not modify it either. A reentrant function is one which can be preempted in its execution and resume without any trouble. A thread safe function can be used in multiple threads without external locks.
- **Logic programming** - uses facts, axioms etc

C++ supports procedural, functional, object oriented i.e. multiparadigm. It has type safety and is an efficient language.

The popular versions are C++ 98, TR1, 11, 14, 17, 20

Abstraction means giving only as much information as required at any level. An entity may look different based on what scope we look at it etc

Fancy - currying is a way of taking a function that takes multiple arguments and making it into a sequence of functions that take one argument each.

Quicksort is in-place if programmed iteratively, in recursive it uses O(nlogn) stack space.

3: Object oriented concepts - 13 Jan 2021

Pre-class

C does not support functional programming because there are no closures. It is not object oriented either. If we want OOP in C we need to implement it ourselves and decide upon the conventions as the compiler cannot support us there.

An interface, once published, cannot be withdrawn. Withdrawing will break the existing code. Therefore new arguments must be made as optional arguments. A header file in C/C++ contains the interfaces(signature) for functions, classes etc.

A data structure's interface is its most important part.

Encapsulation is required so that users don't need to know when the interface has changed. It **encapsulates behaviour and attributes together.** It's not a security concept.

It hides what can change i.e. the implementation

And exposes what will not change i.e. the interface

4: Variables - 14 Jan 2021

We have names for abstractions to make communication easier. A single phrase can hold a lot of information.

Composition is an OOP concept where we store an object within another object.

The attributes of a variable

- Name - Some variables can have no name e.g. temporary variables, dynamic variables i.e. variables which are accessed only through a pointer.
- Location - Could be on stack, heap, data segment. This is a runtime mechanism.
- Type - A type defines a set of values and operations for an object.
- Value - Variables on the data segment are all initialized to zero as that segment is mapped to a zero-filled page in the start. Using an uninitialized variable is undefined behaviour. Thus, create a variable only when you know how to initialize it.
- Size - This is implementation dependent. The sizeof operator is a compile time table lookup operator.
- Scope - Defines the visibility of a variable.
- Intent - Create a variable only when required.
- Lifetime - Time of existence. How long is it taking up memory in the program? Just because it has a lifetime does not mean it can be accessed. That is decided by scope.

5: Starting with C++ - 16 Jan 2021

declaration: indicates the type; only for the compiler

definition: causes allocation of memory, it is also declaration

initialization: putting a value to a variable at the point of definition, efficient, not an expression

assignment: putting a value to a variable anywhere else

C++

<< is the insertion operator, AKA the put to operator. It returns an ostream object. cout is a variable of the ostream type. It is a member of the std namespace.

An operator has rank, arity, precedence, association

Initialization

Can be done as an assignment i.e. int x = 10. Can be done in the constructor format i.e. int x(10). A newer type is the uniform initialization, which does not let the value to lose precision when being assigned to the new variable. I.e. int c{30}.

6: Overloading - 18 Jan 2021

Pre-Notes -> From iostream.h to iostream, the difference is that iostream adds all functions and constants to the std namespace but the header file just added them to the global namespace which was seen as polluting. To include a c header to the std namespace, prefix a c to the name and remove the .h e.g. <math.h> becomes <cmath>

Matching a function call to the list of available prototypes. Match arguments to the parameters. This can be done through overloading.

Matching arguments to parameters & Choosing the right definition

1. Exact matching or trivial conversion. E.g. convert array to pointer, variable to const.
2. Matching a generic function
3. Promotion to a higher type, i.e. without any loss of precision. E.g. integral types to int, float types to double. Int and double are the preferred types to use.
4. Standard conversion. Numeric conversion which might lead to loss in precision. There are no rankings within this.
5. User defined conversions.
6. Unsafe conversions e.g. a pointer to a void *

Overloading is a compile time mechanism which allows functions to have the same name but different signatures (no of args, their types and order). Their names are then mangled by the compiler. The linker is oblivious to mangling.

A function or set of functions declared in the nearest scope overshadow all other functions with the same name but in an outer scope. Declaration can be repeated many times, definitions cannot.

A template function is a compile time mechanism which leads to different functions with the same name but different types. Use this overloading when all the overloaded functions have the same implementation and interface except for types.

Global scope functions and variables can be accessed through the global scope :: .

7: Template functions - 19 Jan 2021

Template functions are instantiated at compile time. That is, whenever the compiler sees a call which can be satisfied by a generic function and it notices that it has not been instantiated before it "instantiates" a new function from the template where all generic variables are replaced with types.

Template functions produce less code in a compiled file than macros. Less bloat.

Implicit instantiation is when the compiler deduces the type based on argument-parameter matching rules.

Explicit instantiation is when the user specifies what types the generic variables should be in that function call. Must be specified in angled brackets right after the function name.

Pointers and references

A pointer can exist without instantiation, needs explicit dereferencing and can point to some other variable or memory location. It can be set to nullptr. The type is int*

A reference variable must be instantiated, is explicitly dereferenced, cannot point to any other variable. It cannot be set to nullptr. The type is int& Conceptually it is an alias to the referent variable. Any change to it is reflected in the referent. Including assignment.

LValue in C++ => if the result of an operation is in a variable and is not a temporary variable.

When you want to pass a structure to a function but do not want to change it, the best way is through a const reference to that structure. A reference variable occupies much less space.

8: Template instantiation - 20 Jan 2021

A pointer to a reference is not possible, neither is a reference to a reference. This is because a reference implicitly converts into the referent. We don't if it even exists after the compile stage. Int is not promoted to float as float is stored approximately.

When the implementation uses the same algorithms, it's good to use templates. When they use different algorithms use overloading.

Can specialize a generic function fully or partly by instantiating it manually after writing the generic function definition. Arrays in C++ are like C and they degenerate to pointers when passed as arguments.

C++ 20 does not support partial specialization.

Template declarations and definition must be in the header file itself. It cannot be put in a separate server.c file because templates are instantiated at compile time itself while linking the server.o file happens in the next stage.

When references are passed to a function any changes to the values affect the original variables, but when values are passed the original values are unaffected.

Check the qb.txt file. There are few questions I could not answer.

9: Pointers - 21 Jan 2021

Garbage AKA memory leak (on heap) => location is present but no access to it.

Dangling pointer => We have access to a location which is not owned by the program. Accessing it is undefined behaviour.

A dangling pointer itself is not dangerous. Useful things can be done with it. The dangerous part is accessing the memory location of the dangling pointer.

Preferred way to pass an array to a function ->

```
template<typename ptr_t>
void disp(ptr_t first, ptr_t last)
{
	while(first != last)
	{
		cout << *first << "\t";
		++first;
	}
	cout << "\n";
}
```

The variables first and last could also be of type ptr_t *

11: Decoupling templates - 22 Jan 2021

Value semantics - Comparing values of the variables

Reference semantics - Comparing the memory locations of the variables

Instead of coupling both return values and parameters, use separate generic types for them. Especially useful when using complex types like linked lists, trees. A default value of a generic type can be initialized through `T var = T()` which is 0 for int, 0.0 for double etc.

A generic type can also take in functions as arguments, they don't need any special syntax for functions. They can take any type of function, the signature of the callback function does not have to be used in the definition. It is defined as needed during compile time through instantiation.

When using multiple template types in a generic function, it is possible to explicitly instantiate a few of them.

12: Pointer to functions and orthogonality - 23 Jan 2021

A function call through a pointer to a function is only resolved at runtime.

A variable of type pointer to a function returning void and with 0 parameters -

```
void (*p)()
```

This variable p needs to be assigned a function for `p()` to be valid. The type of p is fixed at compile time but the value of p will only be fixed at runtime. When optimizations are applied the compiler may replace calls of p with the calls of the function it is pointing to.

Arguments passed to the function are used to initialize their corresponding parameters. When a function returns a value it is copied to a temporary variable, that temporary variable may or may not be used as an RVALUE later on. This is also initialization and not assignment. The temporary variable can then be used for assignment later on.

13: Unit2 && Classes - 25 Jan 2021

A class is a user defined type + implementation of the methods for that type. They are different from structures in that they have access control for members. All members in structures are public.

Constructors

C++ has multiple types of constructors. E.g copy, copy assignment, move etc. It initializes an object (does not create it). That means it initializes attributes of the object and acquires resources. It must be called when an object is created. They must have the same name as the class and have no return type. They can be overloaded. The default constructor is the one without any parameters or with all default parameters. It is possible to call these manually too but not through an object, only through the class name.

Destructors

This is defined to de-initialize the object when it dies. It releases all resources and de-initializes attributes if required. The name must be ~classname and they too have no return type. Destructors cannot be overloaded as they don't have parameters either. It is possible to call them manually too.

RAII

Resource Acquisition Is Initialization. This is a concept C++ supports which means constructors and destructors are used to acquire and release resources instead of having to do them manually. This is there so that the user does not have to write new or delete. It is the preferred method of using limited resources such as heap memory, locks or sockets.

Book-keeping

Book-keeping is meta information about dynamically allocated memory. It's created when memory is allocated through techniques like malloc and `new` with an array. In most cases it is stored in memory just before the block(s). Without this, `free` would be more complicated to call.

It is associated with a memory location and not the pointer to the location. It is also lost in the case of garbage memory.

New & Delete

The new operator is used to dynamically allocate memory in C++. It can also initialize the memory location. It in turn calls `operator new` which actually allocates memory. They are not the same things. When used with an array it uses `operator new` but now initializes memory in a loop. It may not use book-keeping for single objects to avoid memory overhead.

The delete operator is used to free allocated memory in C++. It in turn calls `operator delete` which actually de-allocates memory. Before that it uninitializes the variable or if it is called on an array of variables it uninitializes them in a loop.

14: Constructors and Destructors - 27 Jan 2021

The size of an object of an empty class is 1 byte. This is so that two objects of that class don't end up having the same addresses. When a subclass inheriting from the empty base class is not empty it might choose to ignore the size of the empty base class (Empty Base Optimization).

A constructor is called when the object is initialized and the destructor is called when the object dies. When defining a reference to an object, no constructor is called since no new object is being created.

When defining an array of objects, the constructor is called for each element in the array. Their respective destructors are called when the array dies.

No constructor is called when a pointer to a class is defined but it is called when the new operator is used to create an object and return a pointer to it. It is called multiple times when the new `[``]` operator is used to create an array of objects. The destructor is only called when the delete operator is used. It is called multiple times when the delete `[``]` operator is called. Switching up the destructors can lead to undefined behaviour as the normal new and delete operators may not use book-keeping but the array versions certainly do.

The constructor for a class provided by the compiler vanishes when you write **any** constructor for the class.

Private members of a class can only be accessed through members of the class. Constants attributes and reference attributes of a class must be initialized within the constructor only. Member initialization is the only way they can be initialized. The order of initialization is not the way we write it down there; rather it is the order in which they are declared in the class. Top to bottom, left to right.

Member initialization:

```
Classname::Classname()
: const_attribute1(value1), const_attribute2(value2)
{ /* Rest of the constructor */ }
```

Any function of the class gets passed the pointer to the object through which it is called. This pointer is called `this`. It's a keyword and a const pointer. It's useful for referring to member names which collide with local variable names.

Calling the destructor through an object only performs what is written inside the destructor like any other member function. It does not release the memory for that object. The problem with doing this is that the destructor of the object is again called just before it dies.

As put by isocpp - **Destructors are a "prepare to die" member function.**

15: Ctor and dtors contd. - 28 Jan 2021

A class interface should be in the header file, it can be wrapped by an `ifndef` preprocessor. All implementation specific attributes must be kept private and only the interface attributes should be public.

The implementation of class member functions can be in a separate `server.c` file which is compiled along with the client.

When passing a string to a constructor to be used as an attribute, the string should be copied into a memory that the constructor has allocated. This memory should be freed in the destructor. A default constructor is one that has no parameters or all parameters have default values.

A default object must be one which does not break any member function and has legal values for all attributes.

16: Overloading operators - 29 Jan 2021

```
Person(char *= "", int = 0);
```

Gives a compile time error because the lexer thinks `*=` is one operator. This can be avoided by giving a space between the two or just giving a name to the parameter.

`a---b` is not a compile time error because `a--` is executed first.

The default assignment operator for a class is a shallow copy (member wise copy), where all attribute values are copied. This is risky in case of allocated resources because they will then be pointing at the same resource (e.g. memory) which can cause:

- Changes in one going to the other
- When one object dies, the other's resources are de-allocated

Resources are not allocated in a struct through the compiler therefore assignment operators in struct having arrays is totally fine since each one has its own array and the elements are copied in a loop. In the case of a class, this is a problem since the constructor allocates the resources.

It is good to overload the assignment operator for better readability and convenience. It is good to take in the other object as a const reference. An assignment operator between two objects of class A looks like this:

```
y = x;  ->  y.operator=(x)  -> A::operator=(&y, x);
```

The 4 main steps which need to be followed inside a custom assignment operator are:

- Deallocate the LHS's resources
- Allocate new resources to the LHS
- For each resource, copy value from RHS so that an independent equivalent exists
- For all non-allocated resources, such as primitive types, just copy them

17: Assignment and copy constructor - 1 Feb 2021

To fix the issue caused by self-assignment for an object, just do nothing when the assignor and assignee are the same. A function that overloads the assignment operator must return a reference to the `lvalue`. We prefer a reference over a temporary variable as it saves space. The referent should have a lifetime at least as much as the callee.

Three ways to return a value from a function:

- By value - required when the variable is a local variable
- By reference - good to save space, but lifetime of the variable matters
- By reference to a const variable - this saves space and makes sure that the variable returned will not be modified. Lifetime matters here too

By default, a copy constructor does a shallow copy. The copy constructor is used when an object is initialized with another object. This is common when there are temporary variables upon returning from a function or passing as an argument to a function.

A copy constructor is passed a reference to a constant variable, that variable is the RHS variable (the one we want a copy of). It can then be used to make a deep copy.

To indicate and make sure that attributes of a class are not changed inside a member function we add a `const` to the end of its name, like this:

```
void Person::disp() const { ...

}
```

In this case `this` points to a constant variable.

Assignment goes from right to left. The `LVALUE` is the value of an assignment expression. It is different from initialization. They're similar in the sense that assignment is one way to do initialization e.g. `int` but it does not always work e.g. when an object holds resources. Initialization is done only once, when the object is created.

18: The complex class and interfaces - 2 Feb 2021

The default values of parameters should be provided during declaration so that the compiler can replace the call in client code with it appropriately.

When we have to decide between two interfaces we choose the one which can be used in the most intuitive way and will be hard to misuse. It should provide convenience to the user.

Any private member of an object is accessible through the member function of any object of that class which has scope in the member function. The private members of an object are accessible in the definitions of any member function of the class regardless of the fact that it is the one calling it or not.

For functions which just return a newly created object without any modifications to the object the compiler may optimize the call so that the object is not created inside the function but it is the temporary object which the function returns.

A function declaration can be within a block. This is to support the C type legacy.

All parameters coming after a default parameter must be default parameters.

19: Operator overloading - 3 Feb 2021

A constructor of an object cannot be called multiple times since initialization can only happen once.

Any object can be type cast to a user defined object if the user defined object has a constructor that takes a single argument. To prevent the compiler from implicitly performing this type casting you can add `explicit` before the declaration of this constructor. That means it can be type-cast only by the user explicitly doing so.

Returning a reference to a static variable is not a good idea because if the user uses the value returned without storing it in a new variable it can cause changes to the static variable (which may not be what the user wanted to do).

Arity, associativity and precedence of operators

Overloading an operator is only possible when at least one operand is a user defined function. Usually the left most operand becomes the caller, therefore the operator overloaded member function for that object is called. Therefore when operands are of multiple types, commutativity of an operator cannot be assured.

An operator function, if being defined or used should be useful in the domain it is being used in. Not all operators can be overloaded.

The precedence, associativity and arity of an operator are the same as in C++. When an operator has multiple arities, e.g. the `*` operator which can be used for dereferencing as well as multiplication, it can be overloaded for all those arities. The function call operator `(``)` is an operator too and can be overloaded as such.

20: Friend functions - 4 Feb 2021

A friend function is a part of the class but not a member of the class. It has access to private attributes of an object. The `friend` keyword must only be used in the class declaration, the friend function can be declared without the keyword outside the class. It is an interface of the class. It's recommended to use them sparingly. When to use them:

- For a binary operator overloading with no lvalue usage. That is, when an operator does not require an object on the left.
- When the position of operands are not what they are expected to be
- When the convention demands

In these cases, a member function cannot be used or is not preferred.

A friend function can be friends of multiple classes simultaneously. A member function of one class can also be a friend function of another class.

A friend function can be declared as a friend anywhere within a class. It does not get stored in the class' memory.

It cannot access member attributes directly but only through objects of that class which can be passed as arguments. A friend function is not within a class' scopre.

A friend class is an orthogonal concept to a friend function.

21: Operator expressions - 5 Feb 2021

Should an operator function of a class be a member or a friend or is it not possible to overload it?

- Friend - Binary arithmetic operators, Relational operators, Bitwise operators
- Member - Assignment operator(s) (As +=, ++ and -- involve assignment operators), logical NOT operator, dereferencing operators
- Cannot overload - Binary logical operators, ternary operator, sizeof and typeid since they can mess up program if done wrongly, scope and membership operators since they are evaluated at compile time

Evaluating an expression -

1. evaluation of operands - fetching operands to the registers of the CPU
2. evaluation of operators - depends on precedence and association

In C/C++ the order of evaluation of operands is not defined.

Some operators have the property of `sequence point` which means that the order of evaluation of operands is defined. It guarantees all side effects of evaluating an operand shall be complete by the time it is evaluated. The logical OR operator is a sequence point but the arithmetic operators are not. The assignment operator is not a sequence point either, therefore:

`` `a[i] = i++; ` `` is unspecified behaviour.

It is not allowed to overload binary logical operators since we cannot short circuit evaluation if both operands are passed to a function as arguments. All arguments to a function need to be evaluated before stepping inside the function.

22: Templates and operators - 8 Feb 2021

By default a template function will work with a user defined object provided that all operations being performed on the object within the function are either defaultly provided by the compiler or are overloaded by the programmer.

When declaring elements of an array using the `T[] x = { {...}, {...}, ... }` format, a constructor is called each time `{...}` is evaluated, with the arguments of the constructor being the values inside the curly braces. If there are an incorrect number of values there will be a compile time error.

Trial programs -

- Template function to read into an array, test with primitives and user defined types
- A template function to add elements of an array

23: Increment operators and type conversion - 9 Feb 2021

Note: the decrement operators have the same interface (except for `--`), only the implementation is different.

The increment operators should be members of a class since they need to perform assignment to the attributes. Since both of them become `operator++`, to differentiate between pre and post increment, the post increment operator function call is passed an integer 1 as an argument. This argument does not have to be used at all, it can be kept as a parameter without a name. Their return types differ too. The post increment operator returns a temporary object (i.e. by value) but the pre-increment operator returns a reference to the caller object so that further operations can be done on it. This is why `++++a` is valid in C++ while `a++++` is not.

It is a good idea to make the post increment operator call the pre-increment operator within it to reduce code redundancy. The pre-increment operator has no need to create an object but the post increment one will create 2 of them (A temp object with the value of the old object and a temp object due to return by value. The temp object in return by value can be done away with using return object optimization but it makes the implementation complex)

The logical comparison operator should be implemented as a friend function.

There is also a type casting operator. You can define what happens when the user type casts your object to any other type. The declaration for this operator looks like this -

`operator int() const` which means it will return an integer but not modify the caller object. This means that type casting is essentially different from passing your object to the constructor of the desired type.

24: Functors and templates - 10 Feb 2021

When defining a function to test equality on template types, there are three ways to make it work:

- Write a type cast operator function for that class so that they will be converted into objects of classes which have the equality operator defined. This happens when you define a type cast operator function for that class and it can be confusing because it will make that operator work even if you do not intend for it to work.
- Specialize the template. This means remove any `typename T` types and replace them with the type you want. This needs to be a friend function in that case. Partial specialization of template functions is not a feature of C++
- Add the required operator function to that class. Equality in this case.

Functors are function call operator functions. They are declared as `operator()` as a member function of the class. You can specify many of these as they can be overloaded the way normal functions are overloaded. A class which has a functor is called a functor class. It is possible to call any object of that class, even temporary ones.

You cannot call member functions of const objects unless those functions are guaranteed to not change them, which can be done by writing const next to the function declaration which means that `this` will be a pointer to a const object. Calling const objects means that the functor should be declared as a const for that class too.

A template can also handle a pointer to a function type. A pointer to a function can be called, dereferencing the pointer and then calling the result also works the same way.

```
void fun(char (*g)(int), int x) { g(x); (*g)(x); }
```

The above expression is valid and works as expected. The way to call a functor with an int parameter on an object x is `x(10)`

25: Templates and functors - 11 Feb 2021

Using a pointer to a function vs using a functor for template callbacks. A functor is better since it is available at compile time and the function can be inlined. The call is converted to `objectname.operator()(args)` during template instantiation. It can also keep state info over multiple calls since it is a member of an object which has multiple attributes.

Object closure is when a class defines `operator()` and that in turn calls another function. While calling that other function you can pass attributes of the object and so on. This is a callable object with state. It can be very useful in currying functions.

Note: putting a `= delete` after a function declaration inside a class means that the member function cannot be called. It is just there. It is possible to make a few constructors private so that they will not be used by the external program in ways not intended.

`a += b` is not the same as `a = a + b` in python. The first one is expected to be more efficient as it is supposed to modify the object a itself. In C++ the first one is `operator+=` but the second one is `operator=` called with the result of `operator+`

Be sure to define single argument constructors as explicit when you don't want objects of other types getting converted to this object through that constructor call.

26: Nested classes and functors - 12 Feb 2021

A class can be declared inside another class. Two reasons to do this are:

- Hide the inner class. Can only be accessed as OuterClassName::InnerClassName
- The implementation of the inner class depends on the outer class.

An example for the second one is an iterator class. It could be defined inside an linked-list class or an array class and have the same interface but both implementations are different.

The inner class does not have access to the attributes of the outer class. It is not a member of that class either. It is just within that scope.

The `*` operator can be overloaded as `operator*`, but it can be defined as dereferencing for LVALUE, in which case it must return an object by reference or it could be dereferencing for RVALUE in which case it can return whatever is convenient. These two don't have separate interfaces. This means that an object may or may not be able to be dereferenced for LVALUE, depending on the return value of `operator*`

Assignment

Implement the following functions.

- accumulate (reduce elements to a single element between a pair of iterators)
- find a position between a pair of iterators - compare for equality
- find the position of an element greater than a given element between a pair of iterators
- copy elements between a pair of iterators to another container

Try for both the vector and the list.

27: Unit 3 && Template classes - 15 Feb 2021

Function resolution in C++ happens in this order:

1. Find candidate functions
2. Find the best match among those candidates
3. Check for access to the best matched function. Fail if cannot access

Even in case of nested function calls in a statement order of execution of operands is not defined in C++. `` `printf("%d %d %d", a++, --a, ++a)` `` is undefined behaviour.

It is possible to create a template class and use template types within it as attributes, parameters and return types to member functions and inside member functions too. It is critical to make sure that the template being used supports all those operations. There is no concept of implicit instantiation for class template types, they all have to be explicitly instantiated.

When defining member functions of a template class outside the class definition, they can be specialized template class functions or be defined for all instantiations of the template class. Templates can have default parameters in case of a template class. Even while making use of the default template value, the `<``>` brackets must be present during declaration of such an object.

The `typeid` function is a runtime type identification mechanism. It takes an expression or a type as an argument. `typeid(expression)` has a function `char *name()` which returns a string representing the type of the expression. The result of typeid can be compared with results of other `typeid` function calls.

Access control is a compile time mechanism

Runtime mechanisms in cpp -

- Exceptions
- runtime class
- one more obscure thing with virtual classes

28: Template containers, STL introduction - 19 Feb 2021

To instantiate a template class, you can either:

- Have default template names, need not have `<``>` in this case
- Explicitly instantiate all types

One typename can also make use of the typenames defined before it. It can also be made of other template classes (called nested instantiation). Explicit instantiation can get messy sometimes, thus we use:

- Typedef to make template class type names smaller
- Keyword `using` , e.g. (Preferred approach)

```
template<typename T1, typename T2>
using p = MyPair<T1, T2>;
p<string, int> g("ramanujan", 1729);
```

- Using a template function to make an object of the class, this way the types can be taken from the arguments passed to the function. Commonly used e.g. `make_pair`. This approach is most useful when the value returned is stored in an `auto` variable.

STL goal: efficiency, no inheritance for polymorphic usage, no virtual functions, object based and not object oriented

STL has commonly used containers, algorithms, iterators, functors, adaptors, allocators.

Containers:

- Sequential containers
    - Array
        - C-style arrays, fixed size, const time access
    - Vector
        - Dynamic array, variable size, const time access, insert at one end is const time

    - Deque
        - Double ended queue, variable size, const time access, insert at either end is const time
    - List
        - Doubly linked list, variable size, linear time access, insert any place is const time
- Sorted associative containers
    - set
    - multiset
    - map
    - multimap
- Unordered containers
    - set
    - map

A pair of iterators abstracts away a container.

All the algorithms are defined as free functions. They work with iterators and have no clue about how the containers work.

29: STL philosophy - part 1 - 22 Feb 2021

The `find` function should return a valid iterator pointing to the element if it is found, else it should return an iterator pointing outside the valid range, usually passed as the second argument in the function call. Iterators are defined as classes within the container class, usually following this naming convention: `vector<int>::iterator it;`

Duck typing does not work in C++, it is supported at runtime in python.

The `copy` function copies elements from one container to another using two iterators for the first one to specify start and end and one iterator for the second one to specify the starting location. The destination should have enough memory or else `segfault`.

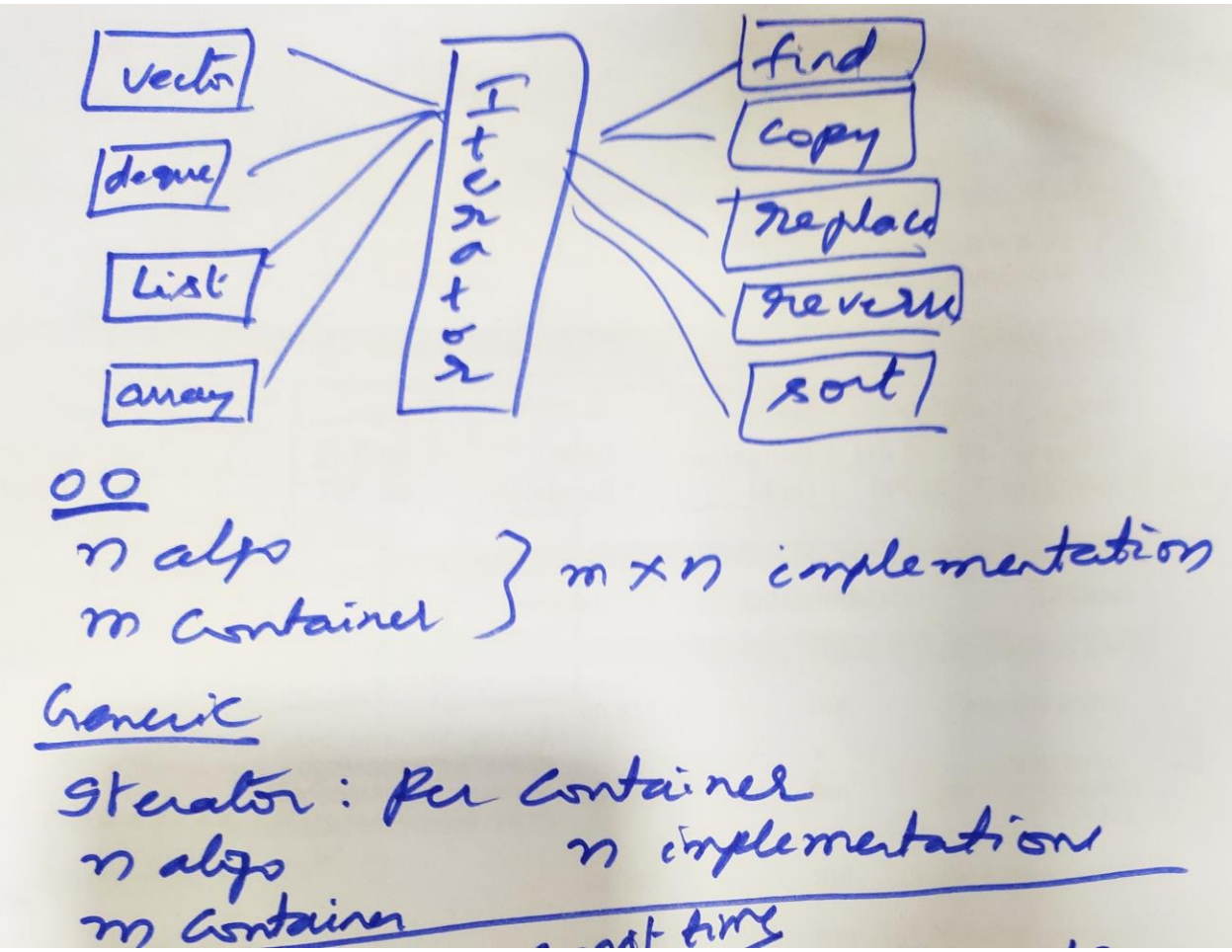
Some container types have the `begin` and `end` member functions which return iterators to the first valid location and the location after the last valid location, respectively. It is preferred to use the `begin` and `end` function provided in STL as they work for arrays too. In case of containers, where possible, it calls the `begin` and `end` member functions.

The `replace` function replaces elements equal to one value with another value. Both these values have to be passed as arguments.

The `reverse` function takes in a pair of iterators belonging to a container and reverses the order of all elements in between the two iterators, inclusive of the begin iterator.

The `sort` function takes in a pair of iterators and sorts the elements between them, inclusive of the begin element. This function does not work on the list container.

30: Containers, algorithms and iterators - 26 Feb 2021



In STL, there are containers, which are data structures implemented as classes, there are algorithms, which are implemented as functions. These are commonly used and are efficiently implemented inside STL. To make the merger between them easier and to have a minimal number of implementations while providing a similar interface, there are iterators. They are implemented once for each container. Algorithms have to be implemented only once. They take in a pair of iterators wherever a container is needed so they can work in the most generic way possible. Both containers and algorithms make extensive use of template classes and template functions.

There is a hierarchy of iterators, from least powerful to most powerful:

Input < Output < Forward < Bidirectional < Random access < “Unnamed superpower”

The properties of each iterator category are:

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([])	`a[n]`

Where `X` is an iterator type, `a` and `b` are objects of this iterator type, `t` is an object of the type pointed by the iterator type, and `n` is an integer value.

A container class should support the most powerful iterator it can efficiently. To support as many algorithms as possible.

An algorithm should demand the least powerful iterator it can play with. To support as many container types as possible.

This means that not all containers are supported by all algorithms, e.g. the generic sort algorithm does not work on the list container. But there is still hope for such containers. Whenever an algorithm can be efficiently implemented specifically for a container in some unique way, it is implemented as a member function. E.g. sort for list and find for set and map.

31: Predicates & Examples - 1 Mar 2021

The reverse algorithm is pretty cool to implement. Must try.

A predicate is a function that takes in 1+ arguments and returns true or false. A unary predicate takes one arg. A binary predicate takes two args and so on.

The generic sort function available in `<algorithm>` demands that either:

- That the container support `operator<` OR
- The container can be converted to a type that supports `operator<` OR
- A third argument is provided, a binary predicate

A predicate can be used as a type or it can be used in an expression. This is why they are implemented as functors usually. The predicate is `operator()` within that functor. From stackoverflow: A predicate is a special type of functor, it need not return a bool, but the return value should be able to be converted to a boolean value.

The `find_if` function takes in a unary predicate as a third argument.

A template function can be overloaded with the same number of arguments, just depending on the type of templates used for each parameter. E.g:

```
template<typename ptr_t, typename t>
void x(ptr_t a, ptr_t b, t c) {
     cout << "The pointer one\n";
}
template<typename ptr_t, typename t>
void x(ptr_t a, t b, t c) {
     cout << "The value one\n";
}
```

That's very nice!

32: Sorted associative containers - 5 Mar 2021

A set is implemented as a height balanced binary search tree. It can only contain elements which have unique keys, the ordering of elements is based on the predicate specified during the creation of the object. This is a case of the predicate being used as a type. By default, the predicate used is `greater<T>` which makes use of `operator<`. The set supports logarithmic time search, insertion and deletion.

A set object supports a bi-directional iterator, but it cannot be assigned to as that might break the ordering of the tree. It has a member function called `find` which makes use of the predicate to find the element. To compare for equality it uses this trick:

```
!(a < b) &&  !(b < a)    => a == b
```

It needs only `operator<` while the generic find uses the `operator==` member function.

A map is just like a set, other than the fact that it has a key-value pair structure. The predicate is defined over the key only. The indexing function i.e. `map[key]` provides an LVALUE pointer to the location, it creates an object with the key if it does not exist.

A multimap is like a map other than that keys do not have to be unique, i.e. a single key can have multiple values. It does not support `operator[]` and functions must be managed through map.insert, map.equal_range(key), which provides a pair of multimap iterators to traverse a temporary container which contains all the values belonging to that key.

Checkout assign.txt, it has interesting problems. Under day_36-37.zip.

33: Unit 4 && Lambda and bind - 8 Mar 2021

Bind is a technique to use a predicate with some arguments and wrap it with a wrapper predicate to convert it into a predicate with a different number of arguments. Since they are functors, it is possible to hold state. In a way, it is similar to function currying.

Note: In c++20 you do not have to explicitly instantiate template classes. They can get instantiated during the constructor call.

Bind is a usage of adaptors, they are a design pattern where you create an intermediary interface when two interfaces which need to work together do not match.

There is also a bind function in C++, which takes in a function as the first argument and the other arguments of the function as the other arguments. In case some of those other arguments do not have values that can be assigned during the bind call, they can be fixed later on, during call. For these attributes, put in `placeholder::_n` where n is the number of that attribute among other attributes of the function we are trying to bind. A bound function can be bound once more. Bind returns a variable of the type:

```
function<return_type(param1_type, param2_type “and so on”)>
```

To make use of bind, you need to include the `functional` header file.

Lambda functions are functions with no name, a new variable can be assigned a lambda if you really need to provide a name. It is of the format:

```
[](){}(); // [] : capture; () : parameter; { } : body; () : call
```

Most frequently, they are used to pass a temporary function as a callable. You can also specify the return type of the lambda by `[]() -> return_type {};`

Global variables are visible inside the lambda body, but local variables(by default) are not. To include a variable, it needs to be put inside the capture. By default it can only be used as an RVALUE, to use as an LVALUE, it needs to be placed by reference inside the capture. You can also

say that any variable used in the body should be captured. This can be done by placing `[=]` to include them all by value or `[&]` to include them all by reference.

34: More algorithms in STL - 12 Mar 2021

The range for loop goes over the whole container, it is not possible to get an iterator from it but it is possible to modify elements if the loop variable is a reference variable. If you try to modify the container(not individual elements, but operations like inserting and deleting!) while iterating through it with the range for it is undefined behaviour.

The algorithms of STL are divided into 4 categories:

- Non-mutating algorithms
- Mutating algorithms
- Sort related algorithms
- Numeric algorithms

The for_each loop applies a function to a container, which is represented by a pair of iterators. Its signature is: `for_each(ptr_t first, ptr_t last, fn_t fn)`

The remove function moves all elements, not equal to the argument, to the front of the container. It preserves the order of those elements while doing this. It returns an iterator which points to one element past the last valid element of the new array. Whatever is at this point is undefined. It is good to erase this part from the original iterator. This can be done in different ways for different containers, for vectors it is like: `v.erase(it1, it2).`

There is also min_element and min which are as follow: (Default predicate is greater<T>)

- `iterator min_element(iterator1, iterator2, optional_predicate_to_calculate_min)`
- `element min(element1, element2, optional_predicate_to_calculate_min)`

- `element min(initialization_list, optional_predicate_to_calculate_min)`

An example of the last one is: `min<string>({"rama", "krishna", "beena"});`

And so on… there are many algorithms in STL, check them out!

35: Playing with types - 22 Mar 2021

It is possible to partially specialize a template class. This is done by re-declaring the class but with different template types. It must take in the same number of template parameters while creating an object. Sometimes there can be an ambiguity between specializations. Overall, the compiler tries to match the most specialized template class to the object. E.g

```
template<typename T1, typename T2>
class A
{};
template<typename T>
class A<T, T>
{};
template<typename T1,typename T2>
class A<T1*, T2*>
{};
```

A class can have types within it, they are accessed by `A::B` where B is the sub-type. In a template function, if the type of `A` depends on a template e.g. `vector<T>`, any use of `A::B` must become `typename A::B`, else it thinks that B is a static variable of the class.

This can be further extended to create a mapping between types while defining a function. This is useful when we want specialized behaviour on a few types. This is useful when we want to specialize a few properties for some types without redefining the function. E.g.

```
// develop mapping of types
template<typename T>
struct MyTraits;

template<>
struct MyTraits<int>
{
     using RT = int;
};
template<>
struct MyTraits<char>
{
     using RT = int;
};
template<typename T, typename RT = typename MyTraits<T>:: RT>
RT sum2(T a[], int n)
{
     RT s = RT();
     for(int i = 0; i < n; ++i)
     {
          s += a[i];
     }
     return s;
}
```

Many STL containers define types within them which are used in the algorithms, iterators and other operations. E.g. a vector can have a value_type, a reference_type, a pointer_type and so on. In some cases, it is necessary to have different implementations for these types. E.g. a vector<bool> is a specialized form of vector<T>, it has its own pointer_type and reference_type since we cannot get a pointer to a single bit.

Must check what is a dependent type. How to determine types in case of dependent type.

36: Iterator traits - 26 Mar 2021

Given an iterator, how do you write a function that works on the data it is pointing to?

- Pass the dereferenced value to another template function
- Use auto for the variables
- Use `iterator_traits<ptr_t>::value_type`, where `iterator_traits` is a template class

Iterator_traits has many such dependent types. For reference, const reference, pointer type etc. These can be used when an iterator is available and you want to use variables of particular types dependent on the iterator. You can also use auto, which does not have to go through iterator_traits.

There is one pretty useful iterator trait - `iterator_traits<ptr_t>::category`.

It tells the category of the iterator `ptr_t` that you have. E.g. input_iterator_tag, forward_iterator_tag … random_access_iterator_tag. Using this it is possible to write functions where you are assured of the type of iterator you have. E.g.

```
template<typename ptr_t>
ptr_t my_advance_(ptr_t it, int n, random_access_iterator_tag)
{
      cout << "random access\n";
      return it + n;
}
template<typename ptr_t>
ptr_t my_advance(ptr_t it, int n)
{
      return my_advance_(it, n, typename
iterator_traits<ptr_t>::iterator_category());
```

```
}
```

All those iterator_tags are empty classes where the more powerful iterators inherit from the less powerful ones. Even if the iterator tag does not match exactly, the function can still match due to derive class to base class object trivial conversion. Assigning a derived class object with extra fields to a base class object is called object slicing.

Template metaprogramming

This is when we use template classes and compile time evaluation to perform complex operations when compiling code. The parameters used for templates here are not classes, they are called non-type parameters. Instead of `` `typename T` ``, we can use `` `int T` ``. We can use specialization in this too.

To assign values to fields of class during compile time, we use the eval hack:

```
template<int N>
struct What
{
     enum { val = N * N};
};

template<int N>
struct X
{
     enum { val = N };
};
```

Template metaprogramming can also be done recursively, e.g to calculate the factorial. All these parameters need to be compile time constants though.

37: Move semantics - 27 Mar 2021

To bring in better utilization of resources, C++ introduced move semantics as an alternative to copy semantics. The idea is to steal the resources of other variables, when they will not be used hereafter. This is useful in case of temporaries of objects with resources. To make move semantics possible, there is a concept called RValue reference.

RValue reference

Like an LValue reference, this is a reference to an RValue. The syntax to define an RValue reference is with &&. E.g. `int &&a = 5;` a itself is an LValue since it is a variable but it refers to an RValue. This means that if the RValue had fields, they could be modified. If the RValue is a temporary, it will not go out of scope till the reference also goes out of scope. As expected an RValue reference cannot refer to an LValue. To make an RValue out of an LValue you use the `move` function which takes in the LValue as an arg and returns the RValue.

The move semantics introduces two member functions -

- The move constructor - Steals resources of the param for itself
- The move assignment operator - Steals resources of the param for itself. Must discard its own resources before that.

The parameter for both of them is an RValue reference. The param is not a reference to const either as in the copy constructor and the copy assignment operator. Must keep in mind that the resources of that RValue reference will now change ownership. The referred object will lose those resources (e.g. a pointer will be grounded)

This saves resources when temporaries are required. E.g in a swap function where we use the move constructor and move assignment operator we save two sets of resources.

Example code:

```
class MyStr
```

```
{
      // Move constructor
      MyStr(MyStr&& rhs)
      : s_(rhs.s_)
      { rhs.s_ = nullptr; }

     // Move assignment operator
      MyStr& operator=(MyStr&& rhs)
      {
            if(this != &rhs)
            {
                  delete [] s_;
                  s_ = rhs.s_;
                  rhs.s_ = nullptr;
            }
            return *this;
      }
      char *s_;
};

void myswap(MyStr& lhs, MyStr& rhs)
{
      MyStr temp(move(lhs)); //move ctor
      lhs = move(rhs); // move assignment op
      rhs = move(temp);
```

```
}
```

38: Finer points of Vector, Unordered Maps - 29 Mar 2021

A vector has two values for representing the space it occupies:

- Size -> tells how many elements are there in the vector. Although for vector it is a constant time operation, it might not be so for lists and other types
- Capacity -> how many elements it can store with the array it currently has before it needs to reallocate a new array.

The reallocation that happens when changing capacity causes the current iterators, pointers and references to become invalid. When and by how much the capacity changes is defined within the vector class' dynamic table implementation which uses capacity and size to calculate the loading factor and compare it to a threshold.

If we don't want this resizing to happen anytime soon, we can reserve some memory for the vector using the `.reserve(int)` member function. This makes the capacity equal to reserve after that operation. If the amount specified in reserve is lesser than the capacity, the call is ignored. The capacity will increase normally after the threshold is hit.

When you `push_back` a temporary object, two calls of that object's constructor are made:

1. When the temporary object is created -> normal constructor call
2. When the vector puts that object in its array -> copy constructor call

If the object is large and has resources, we would prefer to use `emplace_back` and pass to it all the arguments we would pass to a constructor of that object. This means that only one constructor call is made and only one object is created. It happens through:

Variadic template arguments

Like variadic function arguments, this helps pass many types while calling. E.g

```
template< class... Args >

void emplace_back( Args&&... args );
```

Where emplace_back can then pass those args to the constructor as follows:

`klass(args…);`

Unordered_map

The unordered_map type has two important default template parameters:

- Hash -> The functor used to hash the keys of the map
- Equals -> The predicate used to compare keys which have the same hash value

By default, they use the `hash<key_type>` and `equal_to<key_type>` functor template classes. We can provide our own in case we need special behaviour. E.g.

```
class MyHash
{
     public:
     int operator()  (const string& s) const { return (int)s[0]; }
};
class MyEqual
{
     public:
     bool operator()  (const string& lhs, const string& rhs) const
     {
          return lhs[0] == rhs[0];
     }
```

```
};
unordered_map<string, string, MyHash, MyEqual> m;
```

39: Java Intro - 5 Apr 2021

Pre-class talk:

Emplace is a mechanism to avoid temporaries. It is guaranteed to be at least as fast as those methods which use temporaries. It is not possible in some containers, e.g. sorted associative ones, which need an object to pass to the predicate function.

Java

Java is a strictly object oriented language, meant to be architecturally neutral. It has no free functions, global variables or operator overloading.

It has a garbage collector, a low power thread which runs in the background and cleans up dangling memory occasionally. This means that it has no concept of pointers. It has an alternative called a reference, but no pointer arithmetic is possible on those. It does not allow destructors for a class either.

The best feature of Java is the JVM, a virtual machine to execute architecturally neutral instructions. It is highly optimized and used by many other languages, e.g. Scala. It uses zero address format instructions (The operands and operators come from the stack / predefined registers). It makes use of Just in time compilation and then runs that on the Java Runtime Engine. Both are written in C for most popular JVMs, although some JITs have been written in Java itself.

All functions are virtual, i.e. base class declares them but does not define them, except for constructors, private and static functions.

There are two major type groups in Java :

- Primitives, stored on the stack -> bool, int, char, double, float ... 8 such types
- Reference types, the variable could be anywhere but the memory it refers to should be on the heap -> array, object of the `Object` class and its children. Integer etc...

Generics in Java are only defined on reference types. Conversion between reference and primitives (value type) is trivial. It happens through boxing/unboxing:

- Boxing - conversion of value type to reference type e.g. int to Integer
- Unboxing - conversion of reference type to value type e.g. Integer to int

E.g.

```
Integer a; // No memory on heap allocated yet

int b;

a = b; // Memory allocated and value of b copied to it  *Boxing*

b = a; // Value copied from memory and stored in b *Unboxing*
```

Java code

All Java code has to be within classes. When executing a class, it must have a publicly available main function since that is the entry point from where execution starts. Since we don't want an object of that class to begin execution we make that main function static. The main function also takes in an array of strings which are passed from the command line.

First we compile the `.java` file to bytecode, because of which all the classes within it get new `.class` files. To run the class with the main function, we move to that directory and just run `java classname`. The Java compiler cares about file names, it looks for classes within the main class and then looks for a file with that class name as the file name.

Generics in Java

There are two ways to accomplish this:

By using an object of type `Object`, the base class for all classes. We can typecast our reference type object of any type to it easily (called upcasting, happens implicitly). But when we want our object back, we need to explicitly typecast from `Object` (called downcasting). If we perform a wrong downcast, we get a runtime exception `classcastexception`

A seemingly better method is to define a generic class.

This is like the template classes we make in C++ (although, not exactly the same). This has a few advantages over the previous method, in that there is no explicit cast needed and a wrong downcast is a compile time error instead. While creating an object of the generic class type, you need to specify the type though (Just like in C++, except that from C++ 17 onwards, it can deduce it automatically depending on the values passed to the constructor.

The bad part about this is that, even on compilation, there are no new classes instantiated. It just does typecasting automatically. This technique is called `type erasure`, which means that within the generic class, we can't have any static members referring to the generic type. It also means that generics only work for reference types and not value types, since only reference types are derived from `Object` and can be upcast to it. `Object` cannot be converted to primitive types, but the other way around is possible. You can get an `int` from an `Object` but only by type casting it to `Integer`.

Example code:

```
public class Example_3
{
	public static void main(String[] args)
	{
		System.out.println("hello world");
		Box<Integer> b = new Box<Integer>();
		b.put(25);
		int x = b.get(); // no casting required
		System.out.println("x : " + x);

	//	b.put("hello"); // compile time error

		Box<String> s = new Box<String>();
		s.put("hello");
		String s1 = s.get();
		System.out.println("s1 : " + s1);

	}
}
```

```
// Generic class

class Box<T>
{
      private T o;
      public T get() { return o; }
      public void put(T o) { this.o = o; }
}
```

40: More Java :(- 9 Apr 2021

41: C# Intro - 12 Apr 2021

42: Ultra modern C++ - 16 Apr 2021

In template metaprogramming we use the enum hack within a class to assign values at compile time. This is not the only way. We can use a constant static member too and assign it value at compile time, and unlike the enum hack you can assign it value of any type.

Template metaprogramming can also be done through template matching, i.e. when a compiler instantiates a particular template class for a statement it makes a programming decision. This can be used to sort of simulate runtime if statements at compile time.

SFINAE

Substitution Failure Is Not An Error. This means that when a compiler chooses a particular template class/function for a statement but catches an error in matching it, it does not tell that it is an error. Rather, it searches for other matching options. This is also widely used in template metaprogramming. The compiler goes through a template function twice. First to make sure the syntax and semantics are all okay, while not looking at class specific properties, second while instantiating for the particular object.

A concept is like a compile time statement that evaluates to true or false for some inputs. It is also used at compile time. In some template functions / classes, we can put the concept as a check to instantiating that function / class. E.g.

```
struct Trait
{
     static const bool value = true;
};
template<>
struct Trait<int>
{
     static const bool value =  false;
};

template<typename T>
concept myconcept =  Trait<T>::value;

template<typename T> requires myconcept<T>
class A
{
     public:
     A() { cout << "ctor of A\n"; }
};
```

The real power of concepts come in when you find the various inbuilt concepts available in the language from C++20 onwards. E.g. Check if a default constructor is available, whether an object can be copied and so on.

SFINAE and the sizeof operators are also frequently used with template metaprogramming.

Ranges are sort of like python generators, there are in built range functions such as `views::filter` for filter and `views::transform` for map. Full example:

```
std::vector<int> ints{0, 1, 2, 3, 4, 5};
auto even = [](int i){ return i % 2 == 0; };
auto square = [](int i) { return i * i; };

for (int i : ints | std::views::filter(even) |
     std::views::transform(square))
     {
          std::cout << i << ' ';
     }
```

There's also the spaceship operator available `<=>`, it's useful in stable sorting and such where we want to separate equal, lesser and greater values. It returns `strong_ordering::less` and other

`strong_ordering::` values depending on comparison. When we use its default definition for a class we get all 6 comparison operations defined for us by default -> `>, <, ==, !=, >=, <=`.

Concepts can also be used to check if certain statements are valid for some objects, e.g. operations. =>

```
template<typename T>
concept Addable = requires (T a, T b)  {
     a + b ; -> any number of statement can be here, none of them
should cause compile time errors
};
```

An `if` statement will not tell the compiler to selectively compile some part of the code, even if the expression can be calculated at compile time. The behaviour can be simulated with templates and specializations.

Note! Whenever a template is initialized, it is initialized only for that statement, other statements cannot use that template initialization, they will have to make their own ones. In that way, they are sort of like temporary objects.

To selectively compile an if statement, we can use a thing called `constexpr` e.g

```
template<typename T>
T double_it(T x)
{
     if constexpr(is_class<T>::value)
//   if constexpr(is_class_v<T>)
          return add(x, x);
     else
          return x + x;
}
```

And that's all about modern C++ features and template metaprogramming.

FIN