C++ çekirdeğinde C’yi barındıran ancak farklı dillerden, farklı öğeleri alarak multi-paradigmalı (generic programlama, object-oriented programlama, functional programlama) bir programlama dilidir.

C++ kodlaması yapılırken aşağıdaki checklist’e uyum yapılmalıdır.

**Checklist**

* Uniform initialization kontrolü

C++’da C’den farklı olarak direct initialization, uniform initialization gibi ilklendirmeler yer almaktadır. Aşağıdaki ifadelerin hepsi aynı anlama gelmektedir.

int x = 10;

int x(10);//direct initialization

int x{10};//uniform initialization

Garbage value ile init edilmesi yerine, bir değer ile başlatmak için kullanılan ilklendirmeler (aritmetik ise ‘0’ , pointer ise ‘nullptr’ veya bool ise ‘false’ vb.),

1. Daraltıcı dönüşümü (narrowing conversion) engellemek için

double dVal = 3.4;

int x{dVal};//double to int narrowing conversion error

1. Most Vexing Parse (MVP), parametre bildiriminin fonksiyon bildirimi olarak algılanıp, önceliği yüksek olan fonksiyon bildiriminin önüne geçmek için

x(10);//x isimli fonksiyon var ise, fonksiyon bildirimi olarak işletilecektir.

İnitialization işlemleri uniform initialization olarak yapılmalıdır.

NOT: *initializer\_list türünden parametresi olan constructor (Ör. vector, string) için ( ) ve { } küme parantezleri arasında farklılık olmaktadır.*

std::vector<int> ivecx(10);//vector(size\_t n) constructor çağrısı, size=10 olan vektör {0,0,0,….,0}

std::vector<int> ivecx{10};//vector(std::initializer\_list<int>) ctor çağrısı, size=1 olan vektör {10}

Ek olarak class içerisinde verilerin initilization işlemi için, constructor initializer list (member init. List, MIL) kullanılmalıdır. Constructor içerisinde yapılacak atamadan genel olarak çok farkı olmasa da bu durum bazı senaryolarda verimi düşürebilir.

class Mydata {

int mx,my;

public:

Myclass();

}

Myclass::Myclass : mx{0} {} //mx hayata ‘0’ değeri ile (init) gelecektir.

Myclass::Myclass {

mx = 0; //init değil, atama (assign) yapılmıştır.

}

NOT : *İlk değer verme biçimleri aşağıdaki gibidir.*

T x;//default init, *sınıf nesneleri default ctor ile primitive türler indetermine değer ile başlar*

T x{};//value init, *sınıf nesneleri default ctor ile primitive türler “0, false, nullptr vb.” değer ile başlar*

T x(expr);//direct init

T x = expr;//copy init

T x{expr};//direct list init

* Referans kullanımı

C++’da syntax’a yönelik araç olarak referans semantiği yer almaktadır. C’de yer alan pointer kavramı olarak karşımıza çıkmaktadır. Aşağıdaki belirli temel farklılıklar haricinde kavram olarak aynıdır.

1. Pointer gibi nullable olma semantiği yoktur. Dolayısıyla refere ettiği nesne ile initialize olmalıdır.
2. Referans array olamaz. (C’de pointer array kavramı)
3. Ref. to ref. yoktur. (C’de pointer to pointer kavramı)

C’de geçerli olan pointer kavramı C++’da da geçerli olsa da, pointer yerine referans kullanımı yapılmalıdır.

int x =10;

int &ref = x; //int \*ptr =&x;

* Const correctness kontrolü

Değişkenin önüne eklenen **const** anahtar sözcüğü bizlere nesnenin kullanımı hakkında bilgi sağlamaktadır.

type &ref = object; //set amaçlı kullanılacaktır, yazma yapılacaktır

const type &ref = object;//okuma amaçlı kullanılacaktır.

void func(type &ref)//mutator, set fonksiyon veya setter, nesne değiştiren yazma amaçlı fonksiyon

void func(const type &ref)//access fonksiyon, accessor okuma amaçlı fonksiyon

Const semantiği amacına uygun olarak kullanılmalıdır. Üye fonksiyonlar için her zaman gizli argüman sınıf nesnesi düşünülerek kodlama bu kurala uygun yazılmalıdır.

class Data{

public:

void func();//gizli parametre ile void func(Data \*p) anlamı taşır. **Set** fonksiyondur.

void func() **const**;//void func(const Data \*p) yazılamayacağından const anahtar sözcüğü fonk.

tanımından sonra eklenir. **Get** fonksiyonudur.

}

* Type deduction kontrolü

C++’da C’de olmayan otomatik tür çıkartımı için kullanılan ‘auto’ belirteci (specifier) dil içerisinde yer almaktadır.

Kod okunabilirliği için auto specifier kullanımı sınırlı sayıda tutulmalıdır.

*Dipnot: C++ akronimlerinden AAA(****A****lmost* ***A****lways* ***A****uto) presibiyle ters düşmektedir.*

Ayrıca tür çıkartımı için kullanılan ‘decltype’ anahtar sözcüğü de C++ diline özgü bir araçtır. Bu şekilde tür çıkartımı kullanımı da sınırlı sayıda tutulmalıdır.

int x= 10;

decltype(x) y;//int y

* Kapsam sızıntısı kontrolü

C’de bilinirlik alanı, kapsam (scope) kavramı genişten, dar bilinirliğe göre,

File scope

Block scope

Function prototype scope

Function scope

iken, C++’da

Namespace scope

Class scope

Block scope

Function prototype scope

Function scope

şeklindedir. Bir ismin kulanıldığı yerin dışına taşması kapsam sızıntısı olarak tanımlanmaktadır. Kapsam sızıntısının (scope leakage) önüne geçmek için, ilk olarak kullanıldığı yerde değişkeni bildirmek kaliteli kodlama için önemlidir. Scope gereksiz genişletilmemiş olacaktır.

FILE \*f;

//kod

f = fopen (“sercan.txt”,”r”);

C89 öncesi bildirimler kodun en üst kısmında yapılması zorunluydu, ancak kapsam sızıntısı için bildirim, kullanımın ilk yapıldığı yerde olması daha uygun olacaktır.

FILE \*f =fopen(“sercan.txt”,”r”);

Scope leakage için C++ 17 ile eklenen “if with initializer” da önemli bir araçtır. Örnek olarak bir ‘x’ değişkeni sadece if statement için kullanılacak ise scope genişletmemek amacıyla if deyimi içerisinde tanımlanarak, parantez dışında { } block scope’u tamamlandığından, kullanımında derleyici isim aramasında bulamayacaktır ve syntax hatası alınacaktır.

int foo();

int main()

{

int x = foo();

if(x > 10)

{

//code

}

x = 25; //’mx’ değişkeni yerine yanlışlıkla x yazılırsa, geçerli kod olacaktır.

}

if(int x = foo(); x>10)//if with initializer

{

//code

} //x scope sonu

x =25;//syntax hatası

* Function overloading kontrolü

Fonksiyon overloading için, fonksiyonların

1. scope (bilinirlik) alanı aynı,
2. function signature (imzaları, parametrik yapıları) farklı olmalıdır.

Overload edilmek istenen işlev yüklemesi yapılacak fonksiyonun, function re-decleration olmadığından emin olunmalıdır.

Fonksiyon çağrılarında aşağıdaki sıranın takip edileceği bilinerek, fonksiyon çağrısının yapıldığı yerler için tam uyum olmadığı durumların kontrolü yapılmalıdır.

1. Match
   1. Exact match (tam uyum)
   2. Promotion (yükseltme)
   3. Conversion (dönüşüm)
2. User defined conversion
   1. Conversion construtor
   2. Typecast operator functions
3. Variadic conversion

Eğer yukarıda yer alan eşleşmelerden herhangi birinin yakalanmadığı fonksiyon çağrısı olursa kod derlemesi esnasında ‘ambiguity error’ hatası oluşacaktır.

* Genelleştirilmiş sabit ifade kontrolü

C++’da C’de olmayan bir diğer anahtar sözcük ‘**constexpr**’ ile genelleştirilmiş sabit ifadelerin derleme zamanında oluşturulması sağlanır. Genelleştirilmiş sabit ifadeler (*Ör. #define makrolarının doğrudan bağlandığı isimler, vb.*) için bu decleration specifier kullanılmalıdır.

*NOT : Sınıfın statik olmayan veri elemanı constexpr olarak tanımlanamaz.*

* Enum Bildirimleri ve basetype kontrolü

C’de enum bildirimleri underline type ‘int’ türüdendir. C++’daa basetype aşağıdaki syntax ile bildirilebilmektedir.

enum Color : int {

White,

Black

};

Enum bildirimleri basetype ifadesi ile kullanılmalıdır.

Ayrıca forward decleration yapılamaması (redefinition previous definiton was enumarator error), implicit tür dönüşümü ve enumaratorlerin scope’unun olmaması problemlerine çözüm olarak kapsamlandırılmış enum “enum class” bulunmaktadır.

enum class Color {

White,

Black,Gray

};

Enum kullanımında yer alan numaratör eğer başka bir enum için de anlam ifade ediyor ise, kapsamlandırılarak kullanılmalıdır.

* Tür dönüştürme kontrolü

C++ programlama dili C’ye göre daha katı type kontrolü (strict type control) yapmaktadır. C’de örtülü (implicit) olarak yapılan type dönüşümleri C++ derleyicisi tarafından hata olarak algılanacak olsa da tür dönüşümleri kodlamanın hataya açık yerleridir. Tür dönüşümleri için C’de mevcut tür dönüşüm operatörü (*type*) yerine C++’da yer alan aşağıdaki dönüşüm operatörleri kullanılacaktır.

**static\_cast <target\_type> (val);**//statik tür dönüşümleri için kullanılmaktadır.

**const\_cast <target\_type> (val);**// const value dönüşümleri için kullanılmaktadır.

**reinterpret\_cast <target\_type> (val**);// nesnenin bellek alanındaki bitleri başka türden ifade ediyormuş gibi davranmak amaçlı kulanılmaktadır.

**dynamic\_cast <target\_type> (val);**//down casting için kullanılmaktadır

Bu şekilde kod okunabilirliği arttırılacağı gibi, farklı amaçlara sahip casting operatörlerinden herhangi biri üzerinden yapılacak code review’de casting check işleminde hangi amaca yönelik yapıldığı anlaşılabilecektir.

* if-Check kontrolü

if operatörü kullanımında bir sabit ile eşit kontolü yapılırken value expression kontrolü sağ tarafına yazılmalıdır. Bu şekilde yanlışlıkla “==” yerine “=” operatörü yazılırsa, sol operand L value expression olmadığından syntax hatasına takılacaktır.

Örnek olarak ‘x’ değerinin tamsayı değeri ‘10’ kontrolü için;

if(x == 10) yerine if (10 == x) yazılarak, olası bir yazım hatası syntax hatası ile tespit edilebilecektir.

if (x = 10)//always true case

if (10 = x)//syntax error

* this pointer’ın kullanımı

C++ programlama dilinde this, diğer programlama dillerinden (Java, C#, vb.) farklı olarak nesneyi değil adresini temsil eder. Sınıfın bir öğesine erişim için kullanılabilir. Ancak sınıfın öğesine erişim için block scope’ta aynı isime sahip bir öğe yok ise, maskelemeyeceğinden ‘this’ pointer kullanımı gereksiz olacaktır. *Verbose (gereksiz öğe)* olacağı için kullanılmamalıdır.

//data.h

class Data{

int mx;

void func(int x);

}

//data.cpp

void Data::func(int x)

{

mx =x; // this->mx =x;

}

* Shallow copy kontrolleri

Özel üye fonksiyonlarından ‘copy constructor’ kopyalayan kurucu işlevi, bir sınıf nesnesi eğer hayata aynı türden bir başka sınıf nesnesinden değer alarak geldiği durumlarda, derleyici tarafından default olarak yazılmaktadır.

class Sercan{

Public:

int x;

*Sercan(const Sercan &other);//copy constructor derleyici tarafından default yazılır*

}

int main ()

{

Sercan sx;

Sercan sy = sx; //Sercan sy(sx); veya Sercan sy{sx}; şeklinde de init edilebilir. copy constr. çağrısı

}

Derleyicinin yazmış olduğu copy constructor, ‘memberwise copy’ veya ‘shallow copy’ sığ kopyalama yapacaktır. Ancak sınıf veri elemanlarında pointer veya referans bulunuyorsa, hayatı biten nesne için kaynak geri verildiğinde (*RAII idiom*), hayatı bitmemiş nesnenin de kaynağı teslim edileceğinden, pointer ‘dangling pointer’ olacak ve kaynaksız kalma durumu oluşacaktır. Bu durumda pointer ve referansları kopyalamayıp, onların gösterdiği kaynakları kopyalayarak yeni kaynaklar edinmesini sağlayan ‘deep copy’ işlemi uygulanmalıdır.

Default constructor çağrılarının implicit (örtülü) yapıldığı kod satırları (aynı türden bir başka sınıf nesnesi ile ilklendirmeler, ilgili class’ın nesnesini parametre olarak kullanan fonksiyonlar) bu doğrultuda incelenmelidir. Gerekli ise ‘copy constructor’ user-defined şekilde yapılmaldır.

Burada destructor’ı kendimizin yazmak durumunda olduğumuz, kaynak kullanan sınıfın (*allocate edilen bellek alanını geri teslim etmek amacıyla*) default copy constructor’ı programcının işine gelmeyeceğinden, kendisinin tanımlaması gerekmekte şeklinde düşünülebilir. Big-3(!) kontrolü tamamlanmalıdır.

* Big-3 kontrolü

Eğer bir sınıf için destructor’ı kendimiz yazmak durumunda kalınmış ise, Shallow copy ve resource leakage önlemek amacıyla Copy constructor ve Copy assignment’ı da kullanıcı tarafından yazılmaldır (*Big-3*). Bununla birlikte program verimliliği sağlamak için, gereksiz kopyalama işlemi ve gereksiz kaynağın teslim edilmesini (*resource relase*) önlemek amacıyla hayatı bitmekte olan ve başka bir kod tarafından kullanılma ihtimali olmayan nesnelerin kaynağını alma, çalmak için move semantiği özel üye fonksiyonları Move constructor ve Move assigment’ta programcı tarafından tanımlanmalıdır. (*Big-5*)

* explicit kontrolleri

Tek parametreli constructor’lar (*conversion constructor*) örtülü dönüşüm ile sınıf türünden ifade edilebileceğinden, implicit (örtülü) dönüşümü engellemek amacıyla ‘explicit’ anahtar sözcüğü kullanılmaktadır. Eğer conversion constructor ile bir türü, sınıf türüne dönüştürülmek isteniyor ise bunun casting operatörü kullanılarak yapılmasını zorunlu kılmak amacıyla bu anahtar sözcük kullanılmalıdır.

class Counter{

public:

**explicit** Counter(int);//conversion constructor

}

int main()

{

Counter c(12);

c = 23; //örtülü dönüşüm geçersiz, syntax hatası

c = static\_cast<int>(23);

}

Ek olarak eğer sınıfa tür dönüşüm operatörü, operator overloading edilerek yazılacaksa bu işleç yüklemesi de explicit edilmelidir.

**explicit** operator T() const;

* Operator overloading kontrolü

Aritmetik türler için operatörün verdiği çağrışımdan sınıf türleri için de faydalanmak amacıyla operatör işleç yüklemesi yapılabilmektedir. Operatör overloading işlemi iki şekilde yapılabilmektedir;

i. Sınıfın üye fonk. şeklinde, Member operator fonksiyonu,

ii. Global operator fonksiyonu

Burada tekrarlı kod yazmamak için, simetrik operatörler için global operator fonksiyonu, sınıf nesnesini değiştiren operatörler için member operator fonksiyonu yazmak prensip edinilmelidir.

* Sınıfın statik veri elemanı erişim kontrolleri

Sınıfın statik veri elemanı, sınıfın bir üyesi (member) ancak instance’ın bir member’ı değildir. Sınıfın statik veri elemanı fiilen sınıf içerisinde değildir, sınıfın nesnesi olsa da olmasa da hayata gelecektir. Nesne yönelimli programlamaya uyum (*sınıf ile ilişkilendirilmesi, erişim kontrolü, isim arama kontrolleri yapılabilmesi*) açısından global değişken yerine sınıf içerisinde tanımlamalar yapmaktayız. Sınıfın statik veri elemanı, instance’ın elemanı olmamasına rağmen instance ile veya pointer ile kullanılabilmektedir.

Sınıfın statik veri elemanını, “.” Operatörü veya “->” operatörü ile kullanmayarak, çözünürlük operatörü “::” ile erişim daha uygun olacağından, bu şekilde kullanımı önerilmektedir.

* Include bağımlılığı kontrolleri

Proje dosyalarında farklı sınıf türlerinden veri elemanlarının yer aldığı bir sınıf içerisinde, bu tanımlamanın geçerli olabilmesi için, nesnelerin interface tanımının yer aldığı başlık dosyaları (header file) sınıfın başlık dosyasına include edilecektir. Proje kapsamında eğer bir nesne yalnızca bildirime ihtiyaç duyuyor (incomplete type decleration) ise prototip tanımıyla bildirilerek (*Ör. Struct Pimpl\* mp*) başlık dosyası eklemeden kullanılabilir ve böylelikle bu sınıfı kullanacak diğer proje dosyalarında incude tanımlamaları üstel olarak arttırılmamış olacaktır.

C++’da pimpl (pointer implementation) idiomu, sınıfın private bölümünü gizlemeye yönelik bir idiom’dur. Bu idiom kapsamında sınıfın private bölümünde tanımlanan başka sınıf türünden nesnelere ait header dosyaları include edilmeden proje derlenebilecektir. Bu nesneler, sınıfın doğrudan elemanı yapmak yerine sınıfın nested type’ının birer elemanı haline getirilecektir ve bu include edimek zorunda kalınan başlık dosyaları sınıfın interface’inin yer aldığı başlık dosyası içerisinden alınarak implementasyonun yapılacağı .cpp dosyasına eklenecektir.

Eğer böyle üstel artacak nitelikte bir bağımlılık oluşacaksa bu idiom kullanılmalıdır.

Gereksiz include dosyalarının eklenmemesiyle, compile time uzamasının, doğrudan hizmet alınmadığı başlık dosyalarında bağımlılık (coupling) önlenecek ve ileride yaşanabilecek bir değişiklik sonrası negatif sonucundan etkilenilmemiş olunacaktır.

* Global namespace pollution (kirlilik) kontrolleri

C dilinde, eğer farklı modüllerden aynı isimler gelirse isim çakışması syntax hatasına sebep olacağı için, birbirinden bağımsız modüllerde dışarıya açılan varlıklar kötü isimlendirme ile eşsizleştirilerek özelleştirilmektedir. C++’da isim alanları (*namespace*) yaratılarak bu özelleştirilme sağlanabilmektedir.

Kodun okunabilirliği açısından, global isim alanında yer alan isimleri bir araya toplayan container, namespace kullanılarak özelleştirilmelidir. Birden fazla kaynak dosyanın kullandığı ortak modül var ise isim alanı oluşturulmalıdır.

* Namespace bildirimi kontrolleri

Namespace bildirimleri (*Ör.* *using namespace std*) başlık dosyasında yapılırsa, bu header file’ı include eden tüm dosyalara da bu durum empoze edilmiş olacağından, başlık dosyalarında çözünürlük operatörü (::) ile isim nitelendirilmeli ve using namespace bildirimi kullanılmamalıdır.

* Override kontrolleri

*Virtual* anahtar sözcüğü ile tanımlı taban sınıf sanal işlevinin, türemiş sınıf tarafından geri dönüş değeri ve fonksiyon imzası aynı olacak şekilde bildirilmesine *override* işlemi denir. Override işlemi için dile Modern C++ (*C++11 ve sonrası*) ile eklenen **override** bağlamsal anahtar sözcüğü (*contextual keyword*) bildirilmese de syntax hatası olmayacaktır ancak derleyicinin bazı kontrolleri (fonksiyon taban sınıfta sanal mı? , fonksiyon imzası aynı mı? vb. gibi ilave kontroller ile işlemin gerçek anlamda override olup/olmadığı) zorunlu yapması için bu keyword kullanılmalıdır.

class Base{

public:

virtual void func(int);

}

class Der : public Base{

public:

void func(int)**override**;

}

* Sanal gönderim (*virtual dispatch*) kontrolleri

Taban sınıf sanal işlevinin, türemiş sınıf tarafından override edilip, sanal fonksiyona taban sınıf pointer’ı veya taban sınıf referansı ile yapılan çağrı sanal gönderim (*virtual dispatch*) olarak isimlendirilir. Aşağıda sanal gönderimin devreye girmediği durumlar yer almaktadır.

* 1. Taban sınıf pointer’ı veya referans yerine taban sınıf nesnesi ile yapılan çağrı (*object slicing*),
  2. Taban sınıfın üye fonksiyonu, sanal fonksiyonu çözünürlük operatörü ile çağırıyor ise,
  3. Taban sınıfın kurucu işlevi içinde yapılan sanal fonksiyon çağrıları,
  4. Taban sınıfın sonlandırıcı işlevi içinde yapılan sanal fonksiyon çağrıları,

sanal gönderim (*virtual dispatch*) devreye girmez. (a) kaçınılması gereken durumu ifade etmekte iken,

(c), (d) gibi özel üye fonksiyonları içerisinde sanal fonksiyon çağrıları yapılmamalıdır.

* Taban sınıf destructor kontrolleri

Eğer türemiş sınıf nesnesi, taban sınıf adresiyle (*upcasting*) dinamik olarak elde edilirse, operator delete ile silinmesi durumunda aşağıdaki problemler meydana gelecektir.

Base \*baseptr = new Der; //Base constructor çağrısı, Der constructor çağrısı

delete baseptr; //Base destructor çağrısı

1. Der destructor çağrılmadığı için, Der() constructor’ı ile elde edilen kaynak geri verilmeyecektir.
2. delete baseptr; kodu derleyici tarafından aşağıdaki koda çevirileceği için,

baseptr->~Base();

operator delete(baseptr);

Taban sınıf nesnesinin adresi, türemiş sınıf nesnesinin adresiyle fiziksel olarak aynı olmasının zorunluluk olmadığı düşünülür ise, tanımsız davranış oluşturacaktır.

Bu gibi problemlerin oluşmaması için aşağıdaki çözümlerden herhangi biri uygulanacaktır.

1. Türemiş sınıf nesnesi, taban sınıf pointer’ı ile işlenmez ise böyle bir durum hiç oluşmayacağından, bu davranış syntax hatası oluşturması için taban sınıf sonlandırıcı işlevi protected bölümünde tanımlanır.

class Base{

…

Protected:

~Base();

}

ii. Destructor virtual yapılacaktır. Böylece delete baseptr; kod satırında virtual dispatch mekanizması devreye girecek, taban sınıf pointer’ı hangi nesnenin adresini tutuyor ise o destructor, ~Der() sonlandırıcı işlev çağırılacaktır. Bu destructor içerisinde derleyici tarafından en sona zaten ~Base(); sonlandırıcı işlevi de yerleştirilmiş olacağından problem meydana gelmeyecektir.

virtual ~Base();

Taban sınıfların destructor’ı ya public virtual ya da protected non-virtual olmalıdır.

* Error Handling (try-catch, throw) kontrolleri

C++ çalışma zamanı hatalarının işlenmesi için gelişmiş geleneksel hata işleme mekanizmasına sahiptir. Hata zorlayıcı (forcing) mekanizmasının bulunması (*Ör. malloc() fonksiyonu return değerinin nullptr olmadığı kontrolünün programcılar tarafından yapılmaması*), program akışını hatayı tespit eden kodtan, hataya müdahale eden koda dallandırması, hata işleyen kod ile işi yapan kodun iç içe geçmemesiyle sağlanan okuma kolaylığı gibi avantajlar sağlamasıyla tercih edilmektedir.

Standart kütüphane hata sınıf hiyerarşisine sahiptir ve en üstteki taban sınıf exception sınıfı içerisinde sanal (virtual) what(); fonksiyonu yer alır. Kalıtım yoluyla bu sınıftan türeyen sınıflar bu fonksiyonu override ederek hata tanım ifadesini göndermektedir.

class exception{

public:

virtual const char\* what()const;

}

1. Kalıtım yoluyla elde edilen sınıflar türünden throw edilen exception’lar kendi sınıf türüne ait catch bloklarında yakalanacağı gibi taban sınıf türüne ait catch blokların da yakalanmaktadır. Derleyici bu kontrolleri yazım sırasına göre yapacağı için, catch blokları özelden genele doğru yazılmalıdır.

try{

}

catch(const std::out\_of\_range &ex)

{

//bad alloc error handling

}

catch(const std::logic\_error &ex)

{

//logic error handling

}

catch(const std::out\_of\_range &ex)

{

std::cout << “exception caught : ” << ex.what() << “\n”;

}

catch(…)

{

//catch all

}

1. Referans semantiği dışında catch mekanizması kullanılır ise, object slicing (Taban sınıf pointer’ı veya referans yerine taban sınıf nesnesi ile yapılan çağrı) olacak ve virtual dispatch mekanizması devreye girmeyeceğinden, cath mekanizması referans semantiği ile kullanılmalıdır.

try{

}

catch(const std::exception &ex)

{

std::cout << “exception caught : ” << ex.what() << “\n”;

}

1. Exception gönderildiği noktayla (*throw*), exception’ı yakalayan (*catch*) kod arasında, stack frame içinde oluşturulmuş tüm otamatik ömürlü yerel nesneler için derleyici tarafından destructor çağrılacaktır (*stack unwinding*). Dinamik ömürlü nesneler stack unwinding’ten etkilenmeyecektir. Raw pointer yerine smart pointer’a bağlanırsa delete işlemi (smart pointer sınıf nesnesi için destructor çağrısı) garanti altındadır. Kaynak sızdırmamak (*resource leakage*) için exception handling mekanizmasının kullanıldığı kod çağrıları stack framelerinde oluşturulan/edinilen (*new*) dinamik nesneler smart pointer’a bağlanmalıdır. NOT: Constructor nesneyi hayata getiremeyecekse, exception throw etmelidir. Hayata gelmemiş nesne(!) için destructor çağrılamayacaktır. Exception throw eden noktaya kadar kaynak edinilmiş ise, bu kaynakları geri veren destructor ise, çağrılamayacağından bu noktada da kaynaklar smart pointer’a bağlanmalıdır.
2. Stack unwinding sürecinde, bir sınıfın destructor’ı exception throw ederse dilin kuralı doğrultusunda terminate(); fonksiyonu çağrılmakta, program sonlandırılmaktadır. Bu sebeple bir sınıfın destructor’ı exception’ı dışarı sızdırmamalıdır (*no emit/propogate*). Sınıfın destructor’ı aşağıdaki maddelerden birisini uygulamalıdır (*nothrow guarantee*):
3. Hiç exception throw etmeyecek,
4. Destructor içerisinde catch bloğunda exception’ı handle edecektir.

* İterator geçersiz (invalidation) kontrolleri

İteratörler, işaret ettiği kap şeklini dahili olarak değiştirdiğinde, yani öğeleri bir konumdan diğerine taşıdığında ve ilklendirilmiş iteratör hala eski geçersiz konuma işaret ettiğinde geçersiz hale gelir.

Kulanılan container’lar için aşağıdaki iteratörü geçersiz kılan durum kontrolleri yapılmalıdır.

Vektörde iteratör geçersiz kılma şu durumlarda gerçekleşir,

1. Vektörün boyutu (size) kapasitesine (capacity) ulaştığında yeniden yer tahsis (re-allocation) edilecektir. Dinamik dizi başka bir yere taşınacağından iteratör geçersiz olacaktır.
2. Re-allocation olmayacaksa, Herhangi bir konumda vektöre bir öğe silindiğinde (erase) / eklendiğinde (insert), o iteratör ve öncesi geçerli ancak sonrası geçerli olmayacaktır.

Deque’da iteratör geçersiz kılma şu durumlarda gerçekleşir,

1. Baştan/sondan ekleme yapılmıyorsa iteratörler invalid olur ancak pointer/referans geçersiz olmaz. Ekleme sondan yapılıyorsa (*push\_back()*) iteratör ve pointer/referans geçersiz olur.
2. Silme sondan yapılıyorsa (*pop\_back()*) end konumu ve silinmiş öğe konumu geçersiz olur. İlk eleman siliniyorsa (*pop\_front()*) silinmiş öğe konumu geçersiz olur, ortadan silme yapılırsa, tüm öğe konumları geçersiz olur.

List ve set ikili arama ağaçları (binary search tree) containerlar’ı node temelli oldukları için, iteratör geçersiz kılma, ilgili elemanın silinmesiyle gerçekleşir ve veri ekleme/silme sonrası end konumu geçersiz olur.

* Kod verimliliği kontrolleri
  + Ham Döngü kontrolleri (Avoid Raw Loops)

Standard Template Library’de (STL) bir çok fonksiyon şablonu “algorithm.h” başlık dosyasında toplanmıştır. Ek olarak, fonksiyon çağrı operatörünü “()” overload eden birçok *functer, functional object* ’ler de “functional.h” başlık dosyasında mevcuttur.

Bazı temel işlemleri (sort, transform, vs.) döngü ile tarayıp yapmak yerine, algoritmaların tercih edilmesi daha iyi olacaktır. (*avoid raw loops*)

Örnek olarak bir vektör içerisindeki tüm elemanları yarıya bölmek istediğimizde, Standard functional object başlık dosyasında yer alan transform(); fonksiyonu nu aşağıdaki şekilde kullanabiliriz.

**transform**(ivec.begin(),ivec.end(),ivec.begin(), [](int x){return x/2;});

NOT: *Aynı işlevi sınıfın üye fonksiyonu da yapıyor ve algoritmalar da gerçekleştirebiliyorsa, sınıfın üye fonksiyonunu kullanmak verim açısından daha iyi olacaktır. Çünkü örnek olarak bir bağlı listeyi ters çevirmek istediğimizde list.reverse(); sınıfın private elemanına erişerek bu işi gerçekleştirecek (pointer’ları değiştirerek) iken algoritmalar iterator tabanlı çalıştıklarından, reverse(); işlemi için swap uygulayacaktır.*

* + Dinamik dizi verimlilikleri

Standard Template Library’de belirli türden öğeleri bir arada tutan kaplara container denir. Bu container’lar sequence containers (vector, deque, list, forward\_list, array, string), associative containers (set, multiset, map, multimap), unordered associative containers (unordered\_set, unordered\_multiset, unordered\_map, unordered\_multimap) olabilir. Genel olarak benzer arayüz sunmaktadırlar.

Dinamik dizilerin tutulduğu vector sınıf şablonudur. Vektör cache bandlidir. Cache hit (*istenen verinin ön bellekten başarı ile alındığı durum*), Cache miss (*Ön bellekten veri almak için istekte bulunulduğunda verinin ön bellekte olmadığı durum*) oranı özellikle bağlı listelere, list sınıf şablonuna göre kat kat yüksektir. Özel bir kullanım gerekçesi olmadıkça, vector tercih edilmesi program verimliliği açısından olumlu olacaktır.

Vektörler, allocate edilen bellek adresi başlangıcı, sondan ekleme yapılacak adres ve allocate edilen bellek adresi sonu adreslerini tutmaktadırlar. Container büyüklüğü (*con.size()*), kapasiteye (*con.capacity()*) ulaştığında, 1.5 ~ 2 kat kadar kapasiteyi arttıracaktır. Kapasite başlangıçtan belirli ise, sürekli taşıma maliyetinden kurtulmak için başlangıçta alan rezerve (*con.reserve()*) edilmesi program verimliliği açısından olumlu olacaktır.

Kapasite arttırılıp, ardından container içerisinden veri silinerek, büyüklüğü azaltıldığında (*con.size()*), kapasiteye (*con.capacity()*) düşmeyecektir. Bu sebeple, gereksiz alan kullanmamak için kapasite, büyüklüğe büzülmelidir (*con.shrint\_to\_fit()*).

* Attribute kontrolleri

Modern C++ (C++11 ve sonrası) ile standart hale getirilen attribute (niteleyiciler), derleyicinin uyarı mesajı vermesini zorunlu hale getiren kod sözcükleridir. Ana akım derleyicilerinin (GCC, Clang, MSVC) hemen hepsi C++11 standartlarından önce kendi belirledikleri sentaks ile bir eklenti olarak desteklemekteydi. Bazı attribute’lar kullanılarak derleyicinin daha etkin kod üretmesi sağlanabilmektedir. Aşağıda bazı standart attribute’lar ve kullanılması durumunda kontrolü yapılması gerekli durumlar yer almaktadır.

* + [[noreturn]]

Fonksiyonun kendisini çağıran koda geri dönüş yapmayacağı, çağrı sonrası yerleştirilen bir kodu derleyicinin “unreachable code (ulaşılamaz kod)” olarak işaretlemesinin önüne geçmek için kullanılan noreturn attribute, bu şekilde nitelenen fonksiyonun return etmesi tanımsız davranış (undefined behaviour) oluşturacaktır.

* + [[depreceted]]
  + [[depreceted (“reason”)]]

Kullanılabilir ancak kullanılmaması önerilen varlıklar (class, üye fonk, global fonk, enum sabitleri ve tür eş isimleri namespace alanları) için kullanılır. Bu varlıklar kullanıldığında derleyici uyarı mesajı üretmeye zorlanmaktadır.

* + [[fallthrough]]

Switch-case deyimlerinde, durumlardan herhangi birinden, kodun aşağı devam etmesi istenmiş ise (break; deyimi bilinçli eklenmediği durum) kullanılır. Bu attribute’un kullanılması olası bir problemin önüne geçmiş olacaktır.

switch (state)

case1: f1(); [[fallthrough]];

case2: f2(); break;

case2: f2(); break;

default: break;

* + [[nodiscard]]
  + [[nodiscard (“reason”)]]

Fonksiyonun geri dönüş değerinin discard edilmemesi gerektiğini nitelemektedir. Geri dönüş değerinin kullanılmadığı durumda uyarı mesajı oluşturur.

* + [[maybe\_unused]]

Kullanılmayan bir öğe için uyarı mesajı vermemesi için kullanılır.

* + [[likely]]
  + [[unlikely]]

Dallanma deyimlerinde (if, switch vb.) bir kodun yürütülme ihtimalinin daha yüksek (likely) /düşük (unlikely) olduğunu belirtir. Derleyici bu tahminden daha etkin kod üretmesi sağlanacaktır.

* + [[no\_unique\_address]]

Statik olmayan bir veri elemanın diğer veri elemanlarından ayrı bağımsız bir adresi yoktur. Derleyicinin bu attribute ile nitelendirilmiş varlık için bir yer ayırmaması gerektiği bildirilir. Bu varlık sebebiyle alignment requirement sağlanması amaçlı yukarıya align edilen boyutlara sahip nesnelerde, EBO (*empty base optimizastion*) alternatif olarak değerlendirilebilir.

* + [[optimize\_for\_synchronized]]

Fonksiyonun tanımının senkronizasyon sağlanmış bir kodtan çağrıldığı için, daha iyi bir optimizasyon yapılabilir.

* Syntax basitleştirme kontrolleri

1. C’de yer alan typedef tür tanımlaması yerine C++’da using anahtar sözcüğü ile syntax basitleştirilmektedir.

**typedef** int Word;

**typedef** int\* (\*Fptr)(int\*);

**using** Word = int;

**using** Fptr = int\* (\*)(int\*);

Şablon tür eş ismi (*template alias*) için de kulanılabilmektedir.

template <typename T>

**using** minQueue = std::priority\_queue<T, std::vector<T>, std::greater<T>>;

1. Eğer bir sınıf içerisinde birden fazla kurucu işlev (constructor) tanımı yer alıyor ve class’ın statik olmayan veri elemanının başlangıç, initialize değeri belirli ise her kurucu işlev içerisinde constructor initializer list ile tanımını yapmak yerine, default member initializer ile sınıf içerisinde init edilerek *verbose* kodun önüne geçilebilir (*in class initializer*).
2. C++’da sabit ifadeleri binlik ayıracı kullanılırsa, okuma kolaylığı sağlayacaktır.

int x = 100’000’000;// 100000000

1. C++’da C’de kullanılan dizi varlığını korumakta iken, std::Array dizi sarmalayıcı (array wrapper) ayrıca sequence container’ların bir türü olarak sunulmuştur. Kod yazma/okuma kolaylığı sağlaması, STL’e uyumu (container olarak kullanımı, interface sağlaması), array dikeyden (array to pointer convertion) kaçınmak, atama ve taşıma semantiği gibi kullanım avantajları sebebiyle standart C dizileri kullanmak yerine tercih edilmelidir.

**Referans**

C++ Reference, <https://en.cppreference.com/w/>

C++ Core Guidelines, <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

C++ FAQ, <https://isocpp.org/faq>