БИБЛИОТЕКИ

Гл. ас. д-р Елица Ибрямова

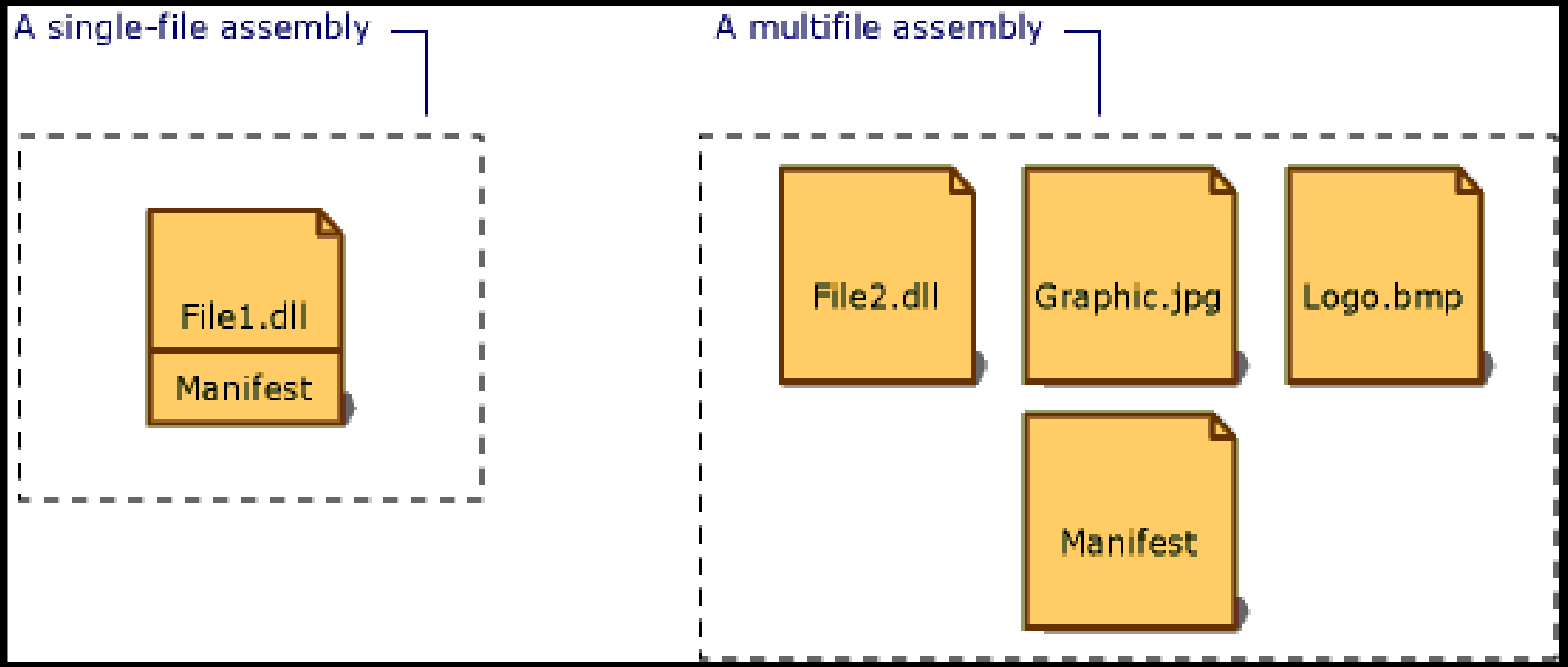
EIbryamova@ecs.uni-ruse.bg

АСЕМБЛИРАНИ БЛОКОВЕ

АСЕМБЛИРАНИ БЛОКОВЕ

- Агрегати за код (асемблирани блокове)
- Логическа структура, която съдържа компилиран код за .NET Framework.
- Може да се състои от един или повече файлове, от .dll или .exe файл.
- Всяко едно асембли (или множество от асемблита) включва манифест (manifest). Той представлява метаданни, описващи кода и ресурсите вътре в асемблирания блок. Това могат да са COM обекти, ресурсни файлове и др.
- Всички ресурси, необходими за приложението могат да бъдат включени в асемблирания блок.

АСЕМБЛИРАНИ БЛОКОВЕ



ФОРМАТ НА АСЕМБЛИТАТА

- Всяко асембли за .NET съдържа следните елементи:
- Заглавия за Windows (headers);

  dumpbin/header Library.dll
- Заглавия за CLR (clrheader);

  dumpbin/clrheader Library.dll
- Метаданни на типовете;
- Манифест на асемблито;
- Допълнителни вградени ресурси, например икони.

АСЕМБЛИРАНИ БЛОКОВЕ

- Не е необходимо да се занимавате с регистрирането, управлението и поддръжката на версии на асемблирания блок и приложението, което се създава.
- Регистрацията на приложението в операционната система се заключва в копирането на файловете в директорията на приложението.
- Асемблираните блокове могат да са общи или частни.

ЧАСТНИ АСЕМБЛИРАНИ БЛОКОВЕ

- Той е достъпен само за приложението, което го е създало.
- Когато се създаде частен асемблиран блок, трябва той да се предостави заедно с приложението.

ПРЕДИМСТВА НА ЧАСТНИТЕ АСЕМБЛИРАНИ БЛОКОВЕ

- Не е необходимо да се регистрира частния асемблирания блок;
- Частният асемблиран блок прави приложението сигурно, защото никое друго приложение не може да използва ресурсите от него;
- Не е необходимо да се спазват конвекциите за именуване на ресурсите.

ОБЩИ АСЕМБЛИРАНИ БЛОКОВЕ

- Асемблиран блок, който да се използва от повече от едни приложение.
- Може да се използва във всеки един от езиците в .NET, ако той е съобразен със стандартите на общата езикова среда.
- Съхраняват се в кеш на асемблираните блокове, който е специална папка във файловата система.

ОБЩИ АСЕМБЛИРАНИ БЛОКОВЕ

- Трябва да се внимава с управлението на версиите и именуването на такива ресурси: те трябва да имат уникални имена.

ПРЕДИМСТВА

- Само описание;
- Едновременност;
- Зависимост от версията;
- Домейн от приложения;
- Безпроблемна инсталация.

ОБРАБОТКА НА ИЗКЛЮЧЕНИЯ

ТИПОВЕ ГРЕШКИ

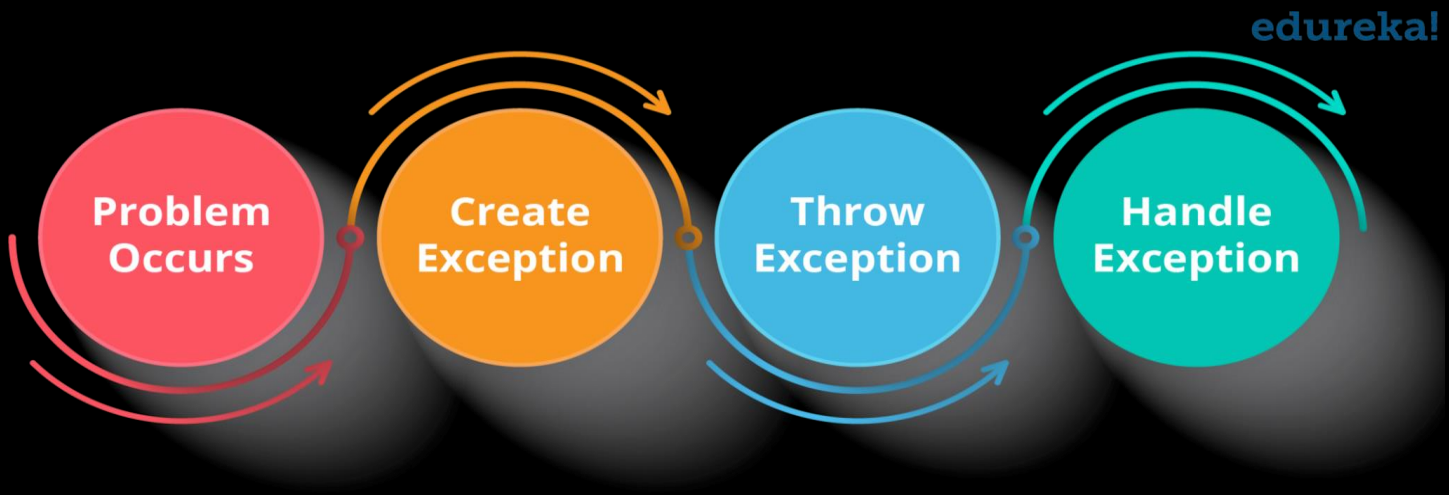
- Грешки на програмиста (off-by-one errors) или грешки в логиката на програмата. Например, ако броят на итерациите на цикъла е с единица по-голям или по-малък от обработвания масив. Програмистът може да забрави, че масивите започват от 0 и да започне от 1.
- Грешки от действията на потребителите. Например, некоректно въвеждане. (SQL Injection)
- Грешки по време на изпълнение (Runtime) – изключения или изключителни ситуации, които трудно могат да се предвидят по време на програмирането. Например, отваряне на повреден файл, опит за четене на таблица от БД, която вече не съществува или структурата й е изменена.

ИЗКЛЮЧЕНИЕ

- **Изключение** (exception) е уведомление за възникване на събитие, нарушаващо нормалната работа на една програма.
- Изключенията дават възможност необичайните събития да бъдат обработвани и програмата да реагира на тях по някакъв начин.
- Когато възникне изключение, конкретното състояние на програмата се запазва и се търси обработчик на изключението (exception handler).
- Изключенията се предизвикват или "хвърлят" (throw an exception) от програмен код, който трябва да сигнализира на изпълняващата се програма за грешка или необичайна ситуация.
- Например, ако се опитваме да делим на нула, кодът, който отваря за операция «деление», ще установи това и ще хвърли изключение с подходящо съобщение за грешка.

ОБРАБОТКА НА ИЗКЛЮЧЕНИЯ

- Обработката на изключенията се осигурява от специализирани конструктори в езиците за програмиране или механизми на хардуера.



.NET МЕХАНИЗМИ

- В .NET се поддържа стандартна методика за генериране и откриване на грешки в средата на изпълнение. Тя позволява на разработчиците при обработка на грешки да използват унифициран подход, общ за всички .NET езици.
- Благодарение на тази методика C# програмист може да обработва грешките по същия начин както и VB програмист.

ПРИХВАЩАНЕ НА ИЗКЛЮЧЕНИЯ

За обработка на изключения в C# се използват следните блокове:

- **Try**– съдържа код, който потенциално може да се натъкне на изключителна ситуация.
- **Catch** – съдържа кода, който обработва изключителната ситуация, настъпила е кода на блок try.
- **Finally-** съдържа код, изчистващ ресурси или изпълняващ други действия, които обикновено е необходимо да бъдат извършени в края на try или catch блоковете. Този код се изпълнява независимо дали е било генерирано изключение или не.

```
try
{
        // Error
}
catch (InvalidCastException e)
{
    // recover from exception
}
```

КЛАС EXCEPTION

System.Exception

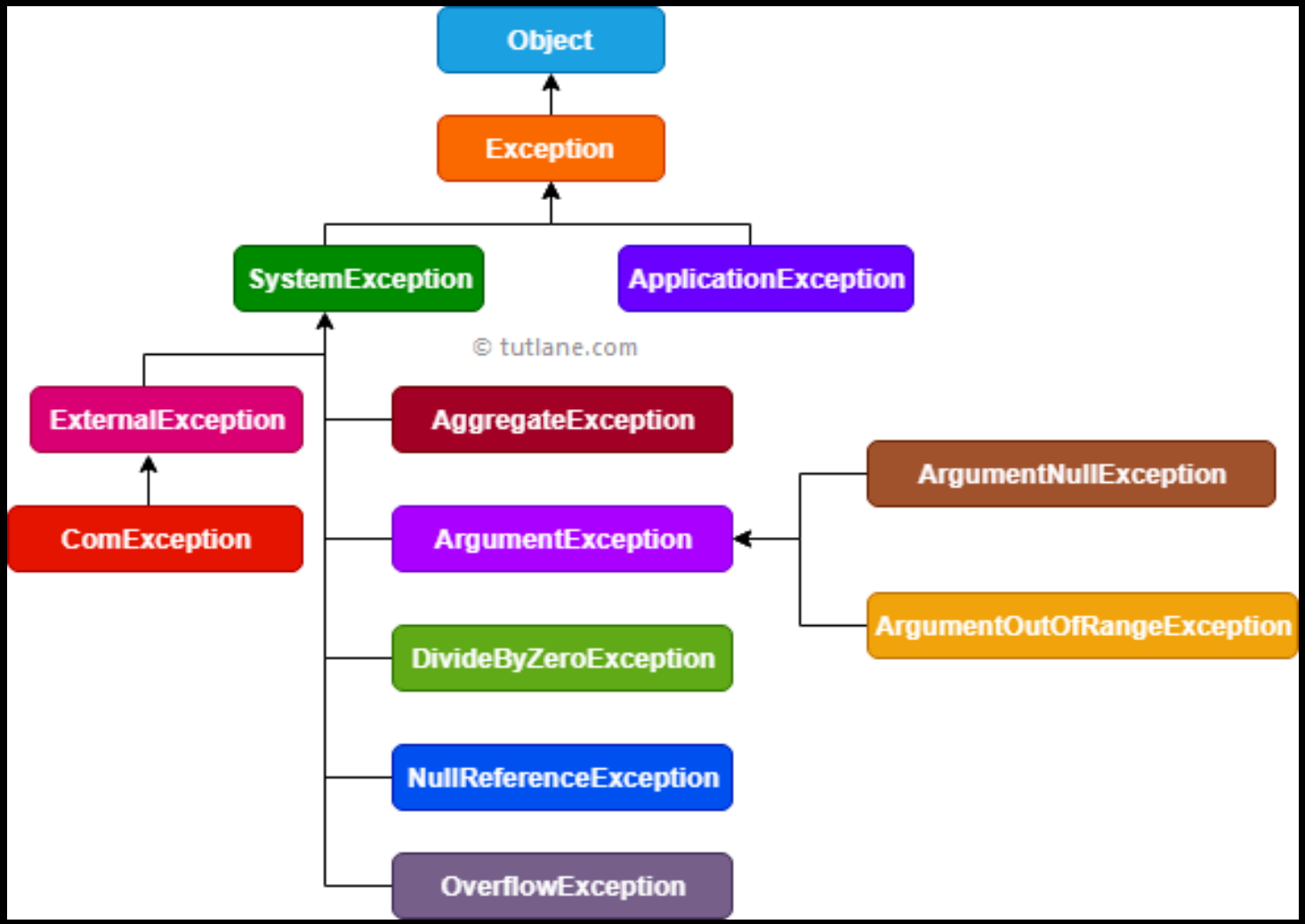
Свойство	Описание
Data	Позволява да бъде извлечена колекция от двойки ключ/стойност (предоставена от обект, имплементиращ интерфейс IDictionary), която предоставя допълнителна информация за изключението. По подразбиране тази колекция е празна и трябва да бъде определена от програмиста. Свойството е достъпно само за четене.
HelpLink	Позволява получаването или задаването на URL адрес с достъпен справочен файл или уеб сайт с подробно описание на грешката.
InnerException	Използва се за получаване на информация за предшестваща грешка, станала причина за възникване на текущото изключение. Свойството е достъпно само за четене.
Message	Съдържа съобщение за грешка. Достъпно е само за четене.

КЛАС EXCEPTION

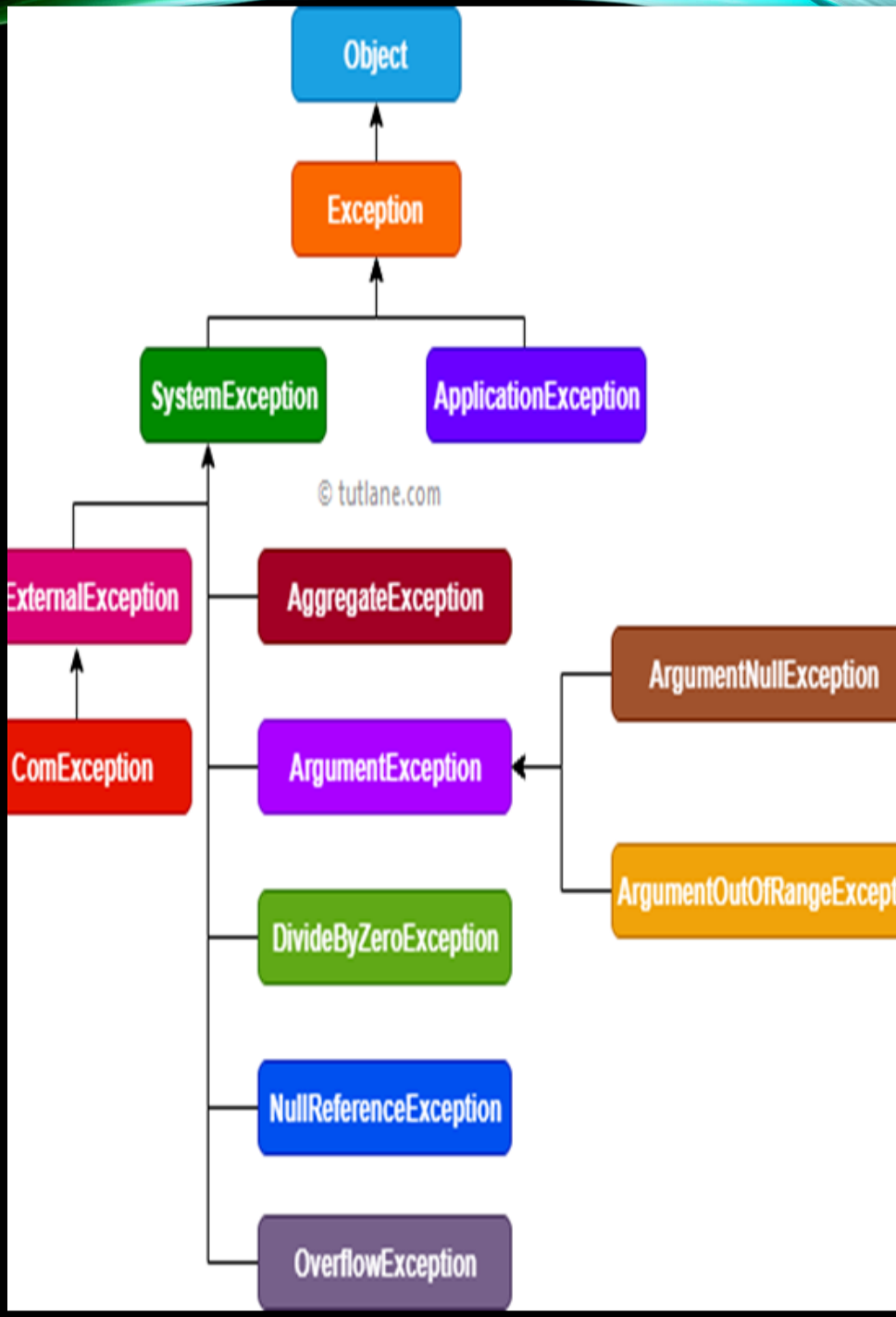
System.Exception

Свойство	Описание
Source	Позволява получаването или установяването на името на асемблирания блок или обекта, довел до възникване на грешката.
StackTrace	Съдържа низ с описание на последователността от изключения, довела до възникване на изключението. Достъпно е само за четене. Подходящо е за използване по време на дебъгване. Съдържанието на това поле може да се записва в log файл с грешки.
TargetSite	Връща обекта MethodBase с описание на многочислените детайли на метода, който е довел до възникване на изключението. Достъпно е само за четене.

ИЗКЛЮЧЕНИЯ НА СИСТЕМНО НИВО



- **Изключенията генерирани от самата платформа .NET** се наричат **изключения на системно ниво.** Такива изключения се смятат за фатални грешки. Те наследяват пряко базовия клас System.SystemException, който на свой ред наследява System.Exception.
- **Изключения на ниво приложение** (System.ApplicationException). Използва се при създаване на авторско изключение , предназначено за конкретно приложение. В този клас се съдържат само набор от конструктори. Единствената цел на този клас е да покаже източника на грешката.



ИЗКЛЮЧЕНИЯ НА НИВО ПРИЛОЖЕНИЕ

- Ако планирате да създавате свои класове с изключения, е необходимо да се спазват няколко препоръки на .NET. Потребителските класове са длъжни:
  - Да наследяват ApplicationException;
  - Да съдържат атрибута [System.Serializable];
  - Да имат конструктор по подразбиране;
  - Да имат конструктор за обработка на „вътрешни изключения“;
  - Да имат конструктор за обработка на сериализацията на типа.

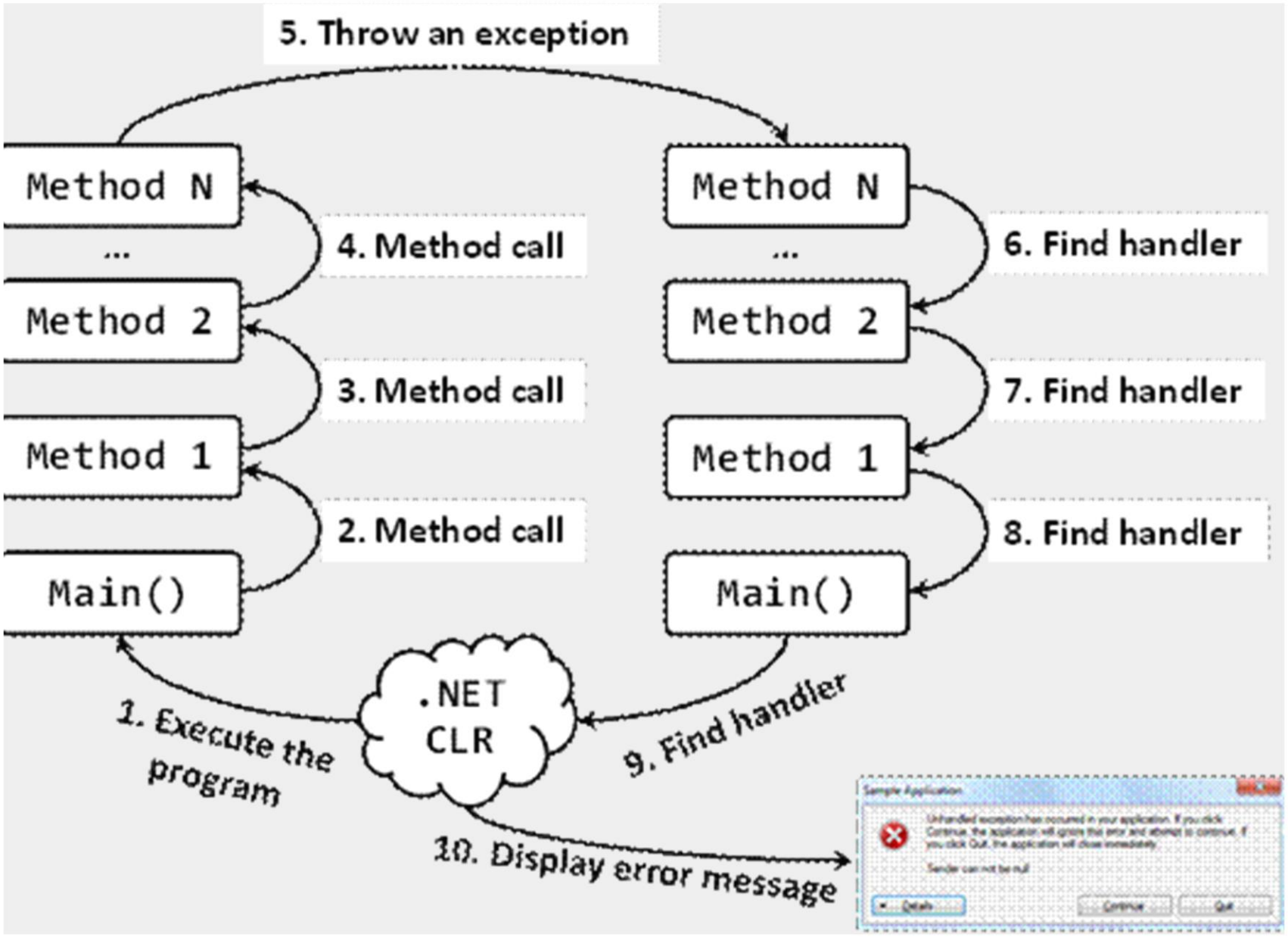
СЕРИАЛИЗАЦИЯ

- Сериализацията е процес на преобразуване на структури от данни или обекти до поток от байтове запазвайки състоянието на техните полета и свойства при тяхното съхранение.

КЛЮЧОВИ ДУМИ CHECKED И UNCHECKED

- В C# може да се укаже дали в кода да бъде генерирано изключение при препълване.
- Ако е необходимо да се укаже израз за препълване, тогава се използва checked.
- Ако трябва да се игнорира препълването, то ключовата дума е unchecked.

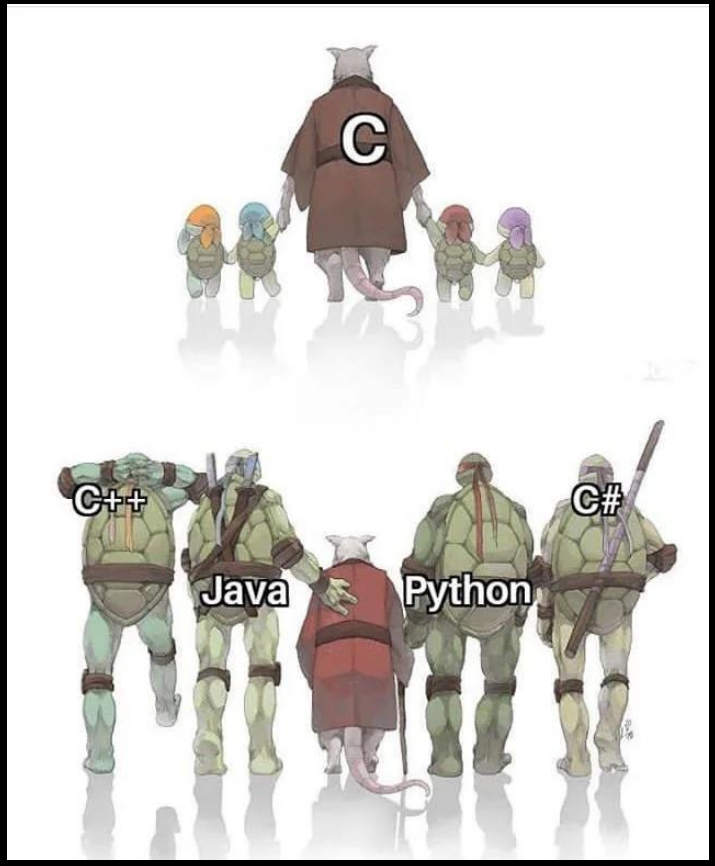
```
checked {
        unchecked {
        // Оператор
        // Оператор
}
        }
```


5. Throw an exception
Method N
...
Method 2
Method 1
Main()
4. Method call
3. Method call
2. Method call
Method N
...
Method 2
Method 1
Main()
6. Find handler
7. Find handler
8. Find handler
.NET CLR
1. Execute the program
9. Find handler
10. Display error message

STANDARD TEMPLATE LIBRARY (STL)

ВЪВЕДЕНИЕ

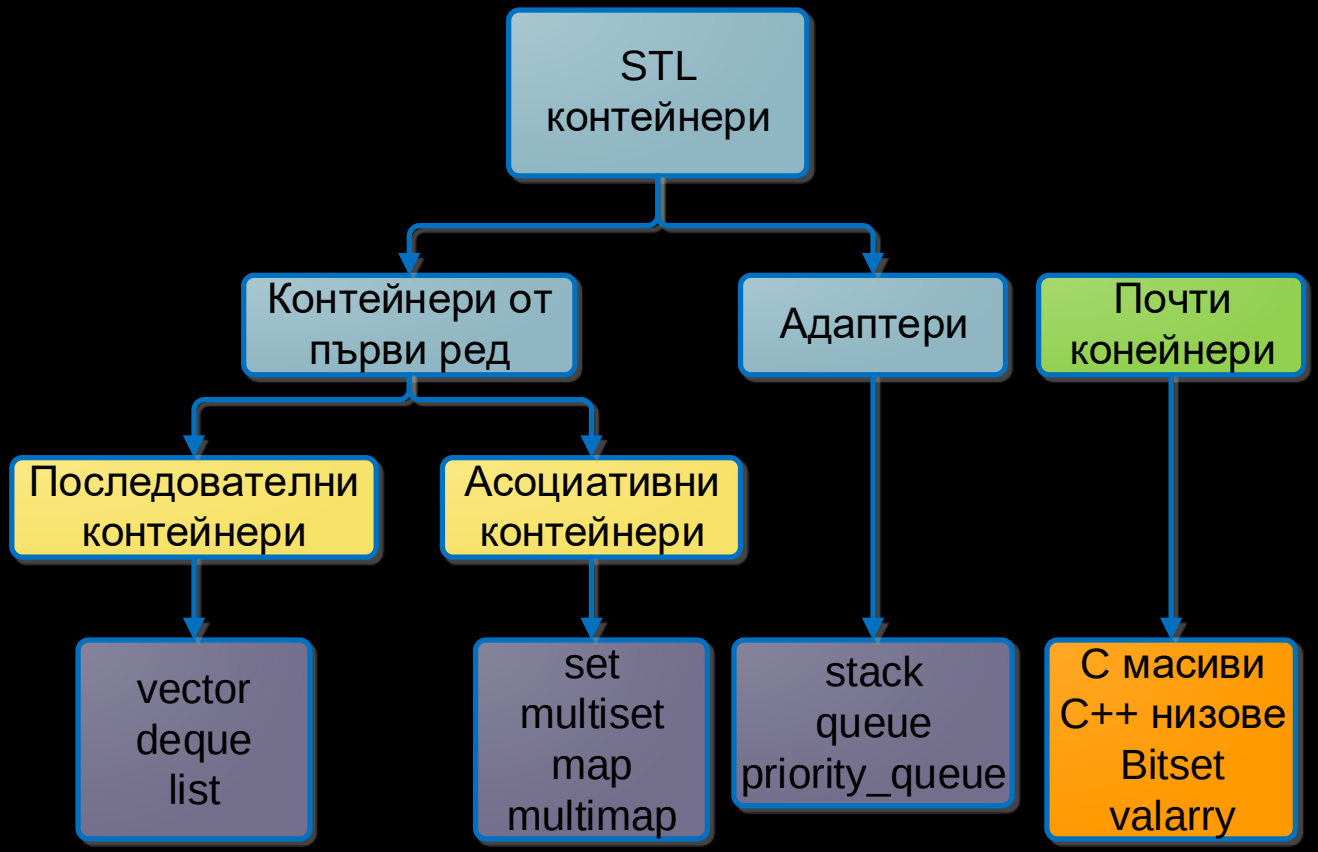
- STL е софтуерна библиотека, включена в стандартната библиотека на езика C++.
- Библиотеката се състои от класове и функции за работа с основни структури от данни, групирани в 3 категории:
  - ✓ Контейнери;
  - ✓ Итератори;
  - ✓ Алгоритми.


C
C++
Java
Python
C#

ВЪВЕДЕНИЕ

- Контейнерите и итераторите са класове шаблони (template classes);
- Алгоритмите са функции, които оперират с контейнери от произволни базови типове.
- Итераторите са класове, с обектите на които се осъществява достъп до компонентите на т.н. контейнери от първи ред.

ВИДОВЕ КОНТЕЙНЕРИ



STL КОНТЕЙНЕРИ

- STL съдържа десет контейнери, разпределени в три групи:
  - ✓ последователни контейнери,
  - ✓ асоциативни контейнери и
  - ✓ адаптери.
- Контейнерите от първите две групи се наричат още контейнери от първи ред.

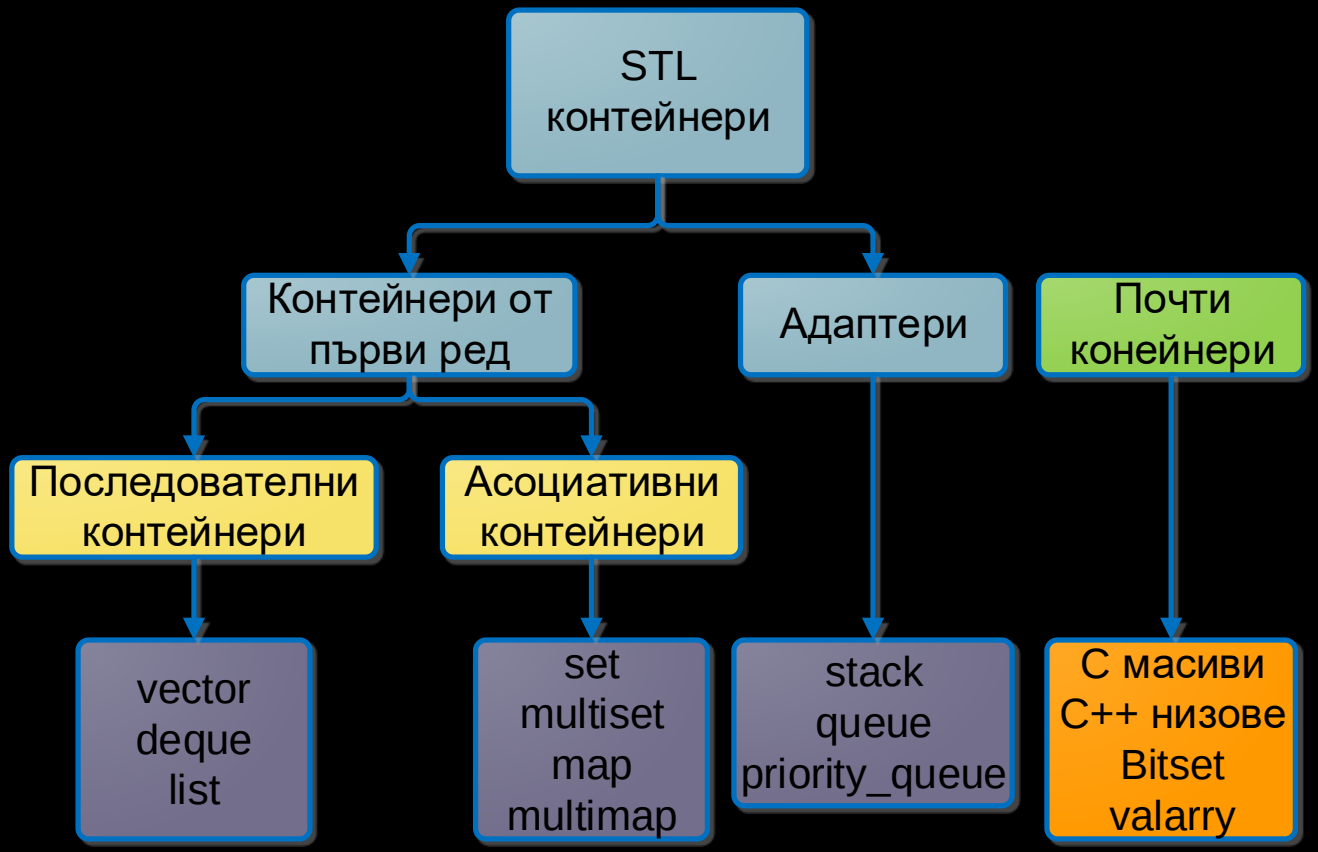
STL КОНТЕЙНЕРИ

- Освен 10-те контейнера STL предоставя още четири структури от данни наричани почти контейнери.
- Това са: C –масивите , C++ низовете, bitset (битови) низове и Valarry масиви.
- Последните се използват при програми, изискващи високоскоростни операции с вектори (n-мерни вектори, понятие от математиката).
- Наименованието почти контейнери е следствие на сходство на част от възможностите им с тези на контейнерите от STL.
- Всъщност почти контейнерите поддържат само ограничен набор от операции, характерни за контейнерите от първи ред.

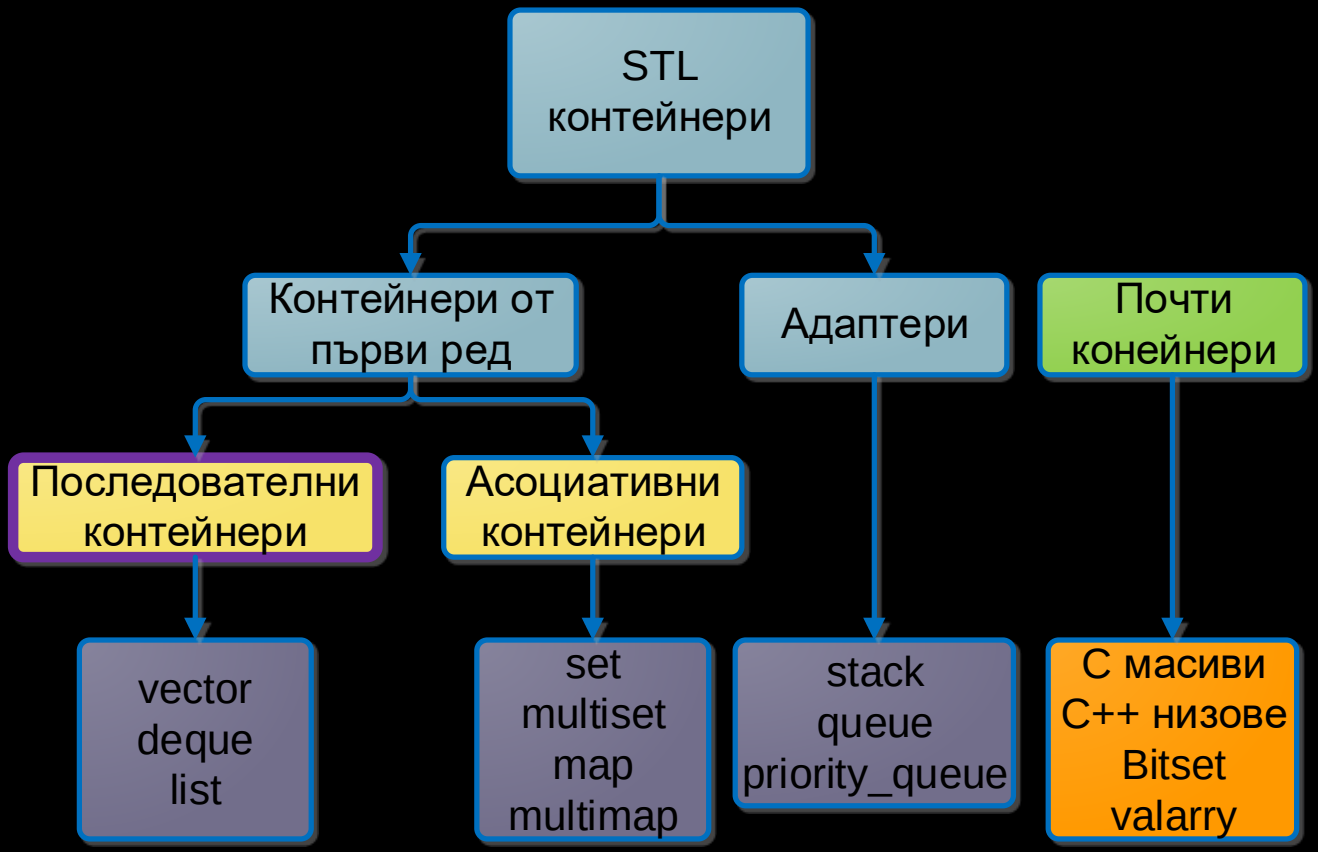
STL КОНТЕЙНЕРИ

- В STL контейнерите могат да се съхраняват данни от прости типове, както и обекти от произволни класове, в които има дефинирани public конструктор за копиране, деструктор и оператор за присвояване.

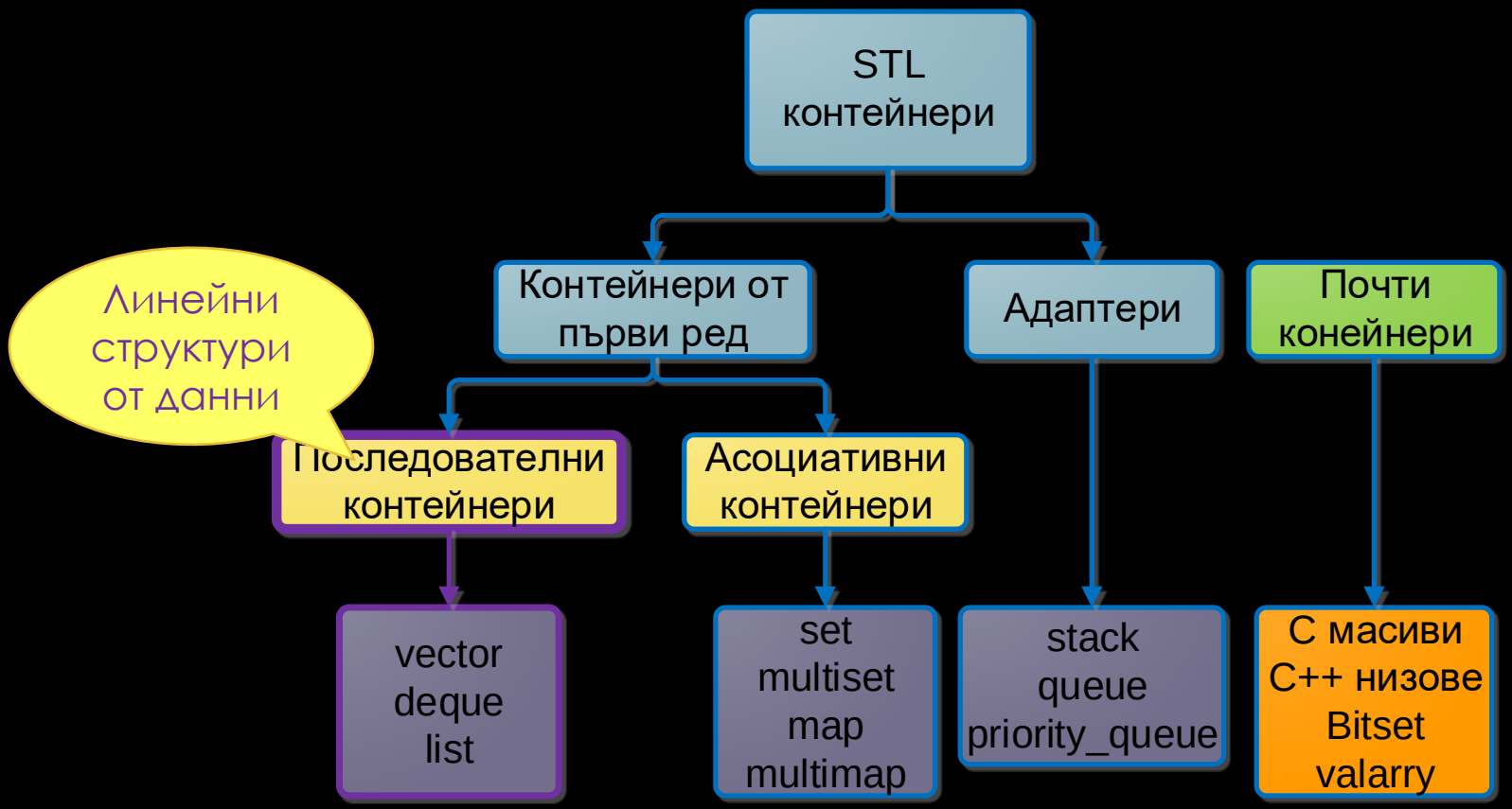
ВИДОВЕ КОНТЕЙНЕРИ



ВИДОВЕ КОНТЕЙНЕРИ



ВИДОВЕ КОНТЕЙНЕРИ



КОНТЕЙНЕР VECTOR

- Той е клас шаблон и чрез него могат да се декларират вектори с разнообразни базови типове.
- Векторите предоставят директен достъп до всеки елемент, като добавяне и отстраняване на елемент от всяка позиция и при необходимост може да се разширява автоматично.
- Някои от операциите с вектори се изпълняват със сложност O(1), като например добавяне в края и отстраняване на последния елемент (push_back, pop_back), а със сложност O(n) се вмъква **(insert)** или отстранява **(erase)** елемент, който е вътрешен за вектора.

КОНТЕЙНЕР DEQUE

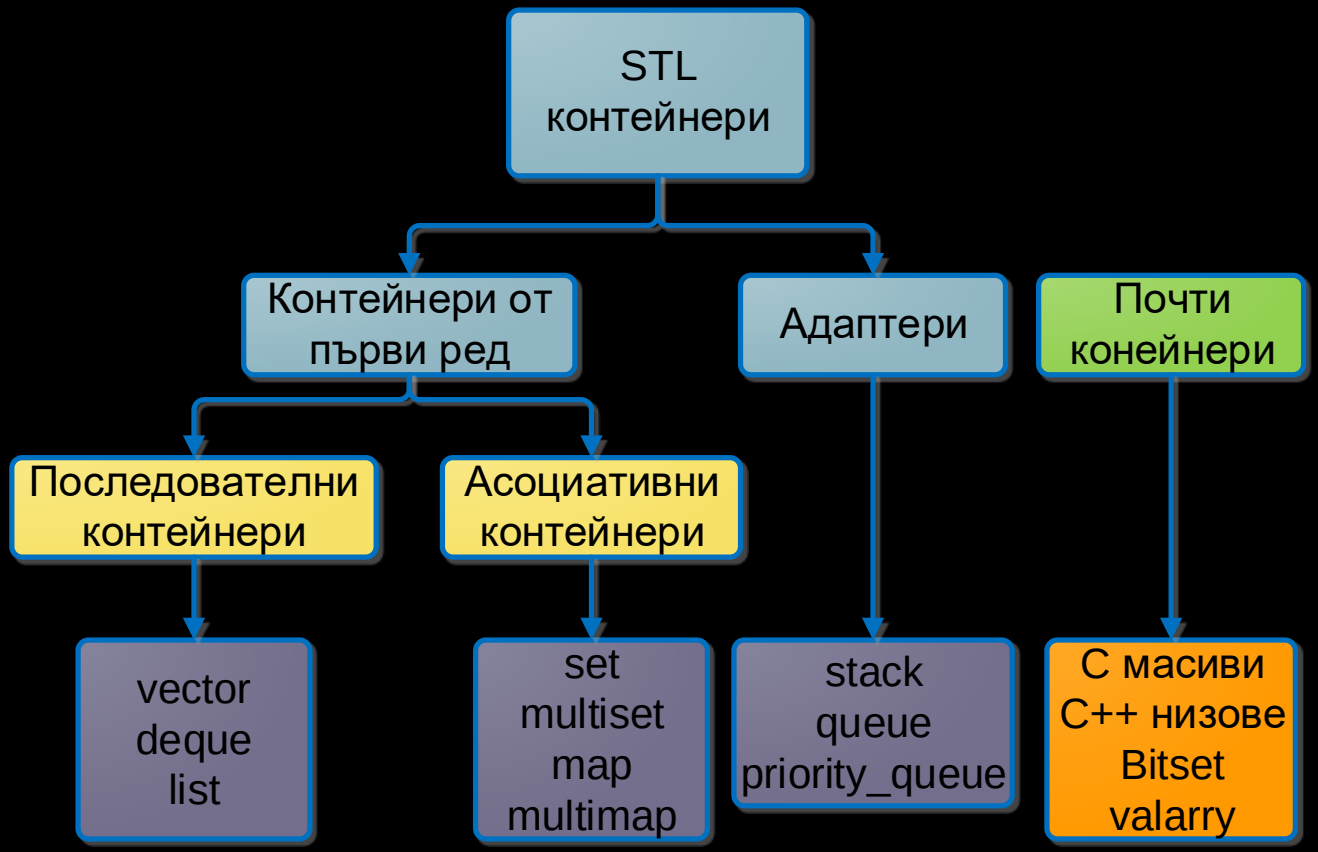
- Контейнерът **deque** (чете се „дек”) е линейна структура от данни, реализирана чрез множество от динамични масиви.
- Той предоставя директен достъп до всеки елемент чрез индекс, както при векторите.
- Декът има почти същото множество от операции като вектора, но **допуска добавяне (Push_front, push_back) и отстраняване (pop_front, pop_back) на елементи от двата края**.
- Налични са операции, с които се вмъкват и отстраняват вътрешни елементи от дек, но те са със сложност O(n).

КОНТЕЙНЕР LIST

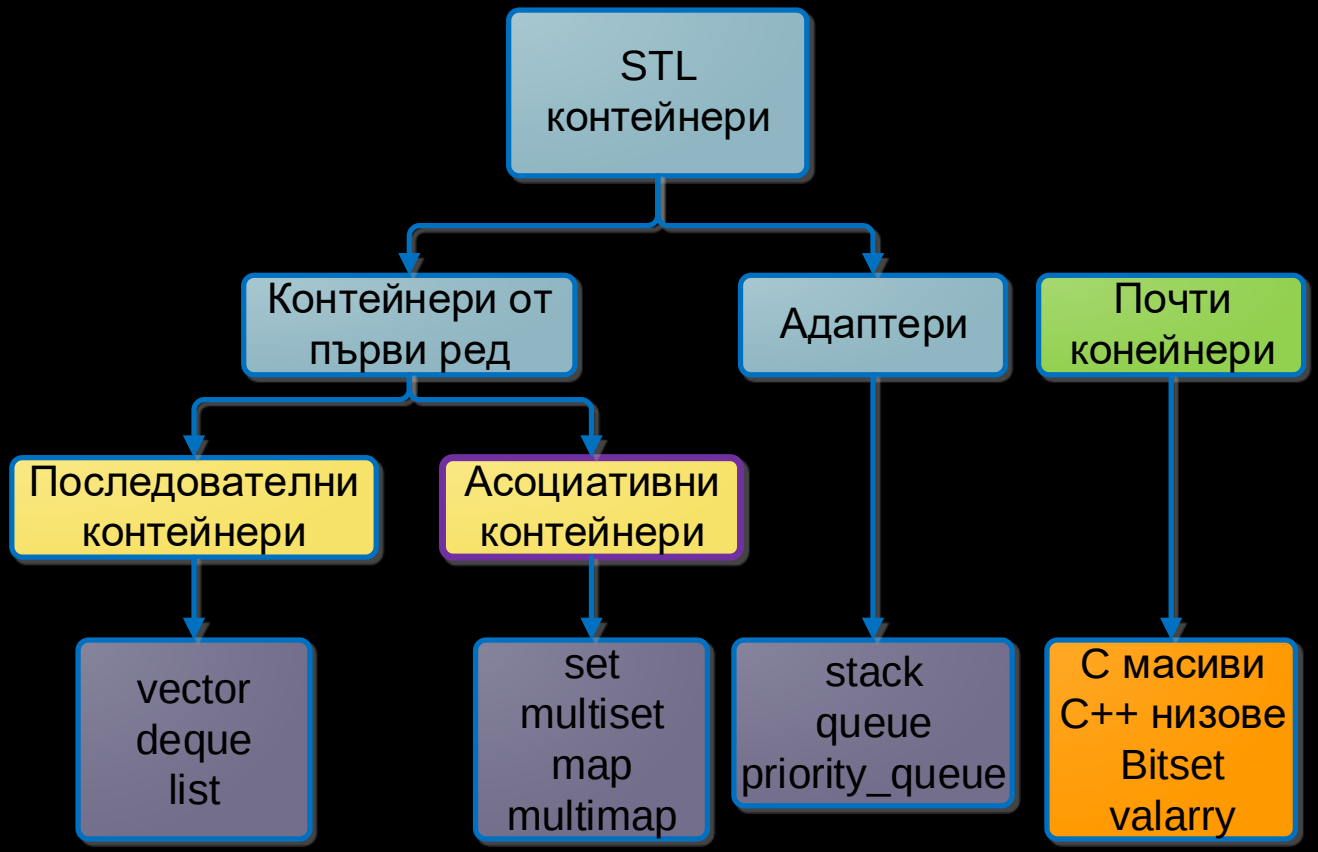
- Контейнерът **list** е линейна структура от данни. На физическо ниво на реализация се използва двусвързана верига.
- Обект на класа **list** ще наричаме линеен списък, а понякога за по-кратко само списък.
- Линейният списък предоставя ефективни операции за добавяне и отстраняване на елементи от всяка позиция на структурата.
- Обхождането на елементите на списък е последователно и може да се извършва и в двете посоки. Това се извършва от съответен двупосочен итератор.
- Съществена разлика на списъка от вектора и дека е, че списъкът не допуска директен достъп до елементите чрез индексиране и оператора квадратни скоби [ ].

- Съвсем преднамерено, множество операции, характерни за някои контейнери на STL са изнесени извън съответните класове.
- Те са реализирани като функции и формират множеството от алгоритмите в STL.
- По този начин контейнерите са олекотени и съдържат по-малко операции, а от друга страна едни и същи функции поддържат общи операции на няколко контейнера.
- Алгоритмите в STL оперират с итератори, които предоставят специфичните механизми за достъп до елементите им.

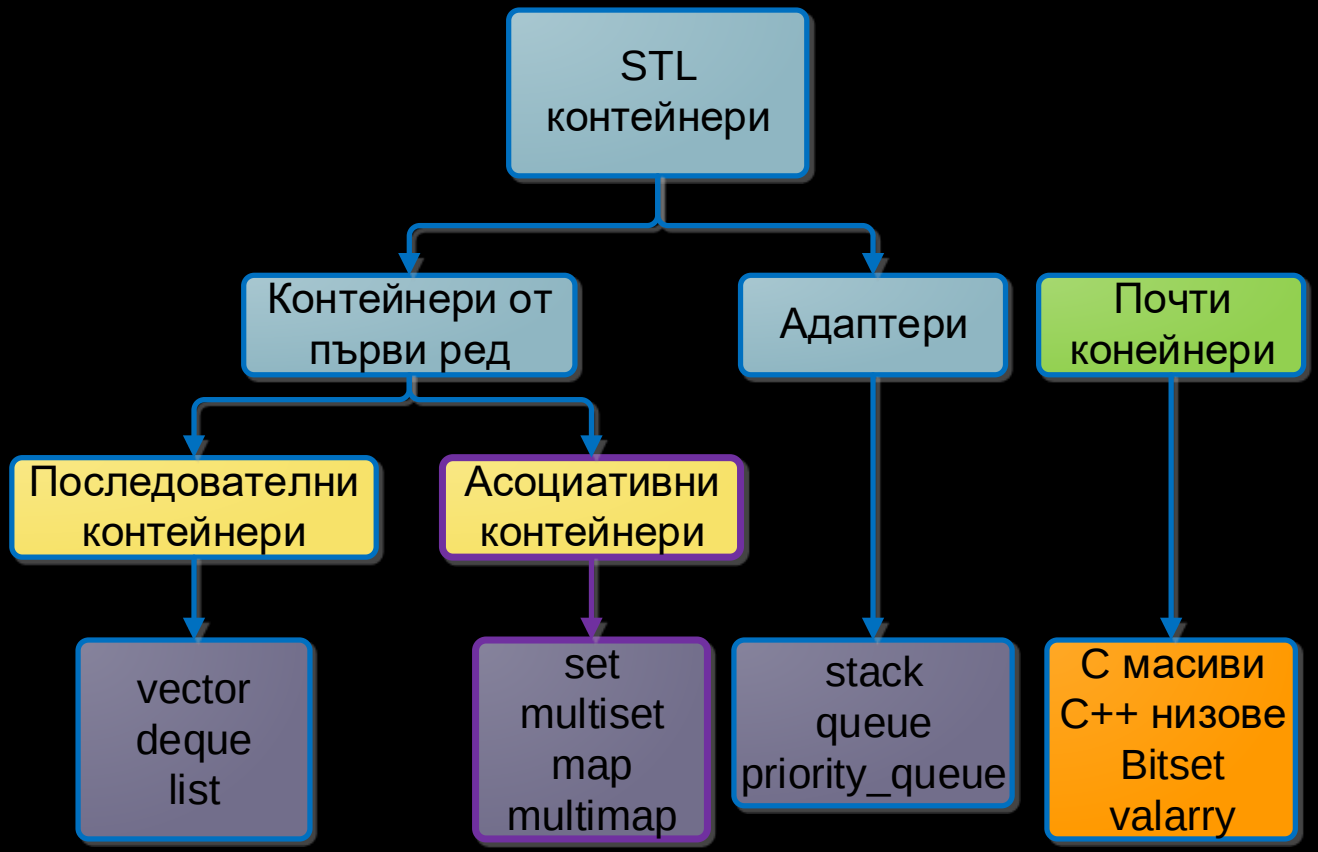
ВИДОВЕ КОНТЕЙНЕРИ



ВИДОВЕ КОНТЕЙНЕРИ



ВИДОВЕ КОНТЕЙНЕРИ



АСОЦИАТИВНИ КОНТЕЙНЕРИ

- Асоциативни контейнери не са линейни структури от данни.
- Водещата идея при тях е наличието на атрибут ключ.
- Той се използва за идентифициране на елементите от структурата и е в основата на алгоритмите за директен достъп и търсене на компоненти в структурата от данни.
- Елементите на асоциативните контейнери са подредени по стойностите на ключа.
- В STL има четири асоциативни контейнери: set и multiset, map и multimap.

АСОЦИАТИВНИ КОНТЕЙНЕРИ: SET И MULTISET

- Елементите в контейнера **set** са подредени в нарастващ ред на атрибута ключ, който е и единствен атрибут на класа.
- По подразбиране релацията за наредба е **„<”.** В случай, че се налага да се използва специална релация за наредба, тогава се налага предефиниране на стандартната релация.
- **Контейнерът set** се реализира с клас имащ същото **име set.**
- Стойностите на ключа са уникални за обектите от всеки **клас set ,** което означава, че не се допуска добавяне на елементи с еднакви ключове.
- Това е единствената му разлика от **контейнера multiset**.

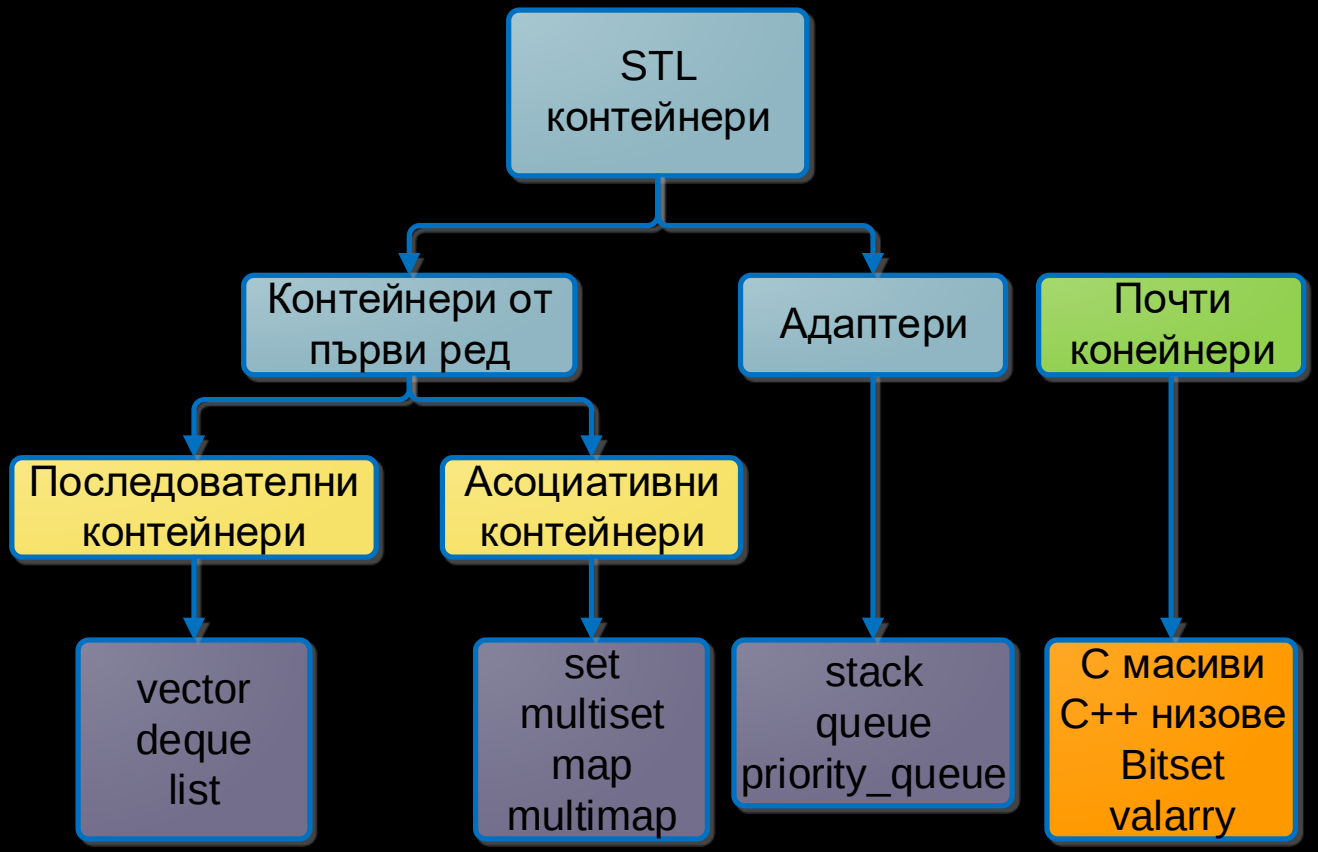
АСОЦИАТИВНИ КОНТЕЙНЕРИ MAP И MULTIMAP

- Елементите в контейнера **map** са двойки от вида **<ключ, стойност>** и затова понякога обекти от класа **map** се наричат и **речници.**
- *Ключът еднозначно определя стойността*. Например ако **id** е обект от клас **map**, с който се осъществява съответствие между идентификационния номер на служител и неговото име, то бихме могли да извършваме присвояване, подобно на това:

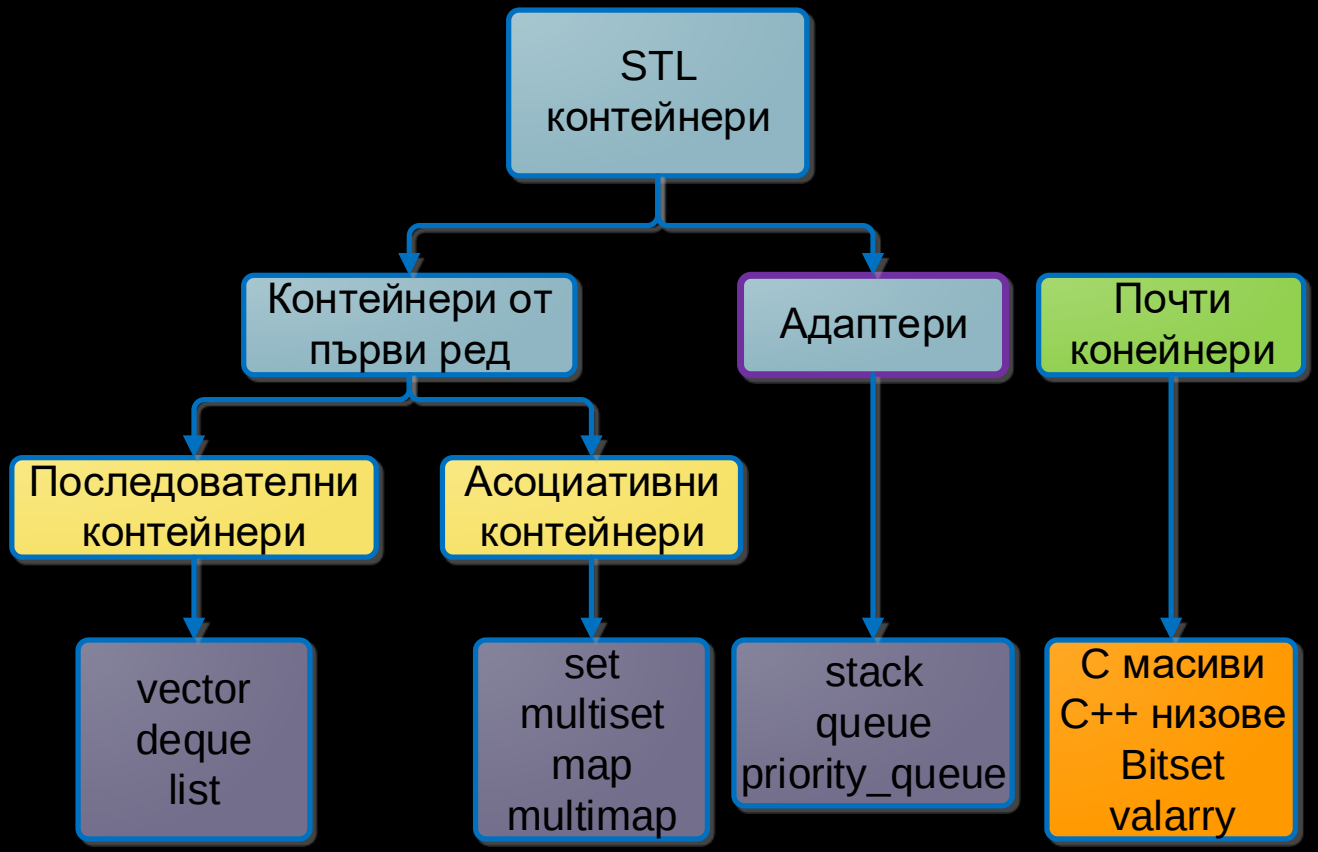
```
//Ключът е от тип int, стойността е от тип  string.
id[20305] = ‘Georgi Petrov”
```

- В обектите от клас **map** стойностите на ключа са уникални. Това означава , че не се допуска добавяне на елементи с еднакви ключове.
- Това е и единствената му разлика от контейнера multimap, в който на един ключ могат да съответстват повече от една стойност.

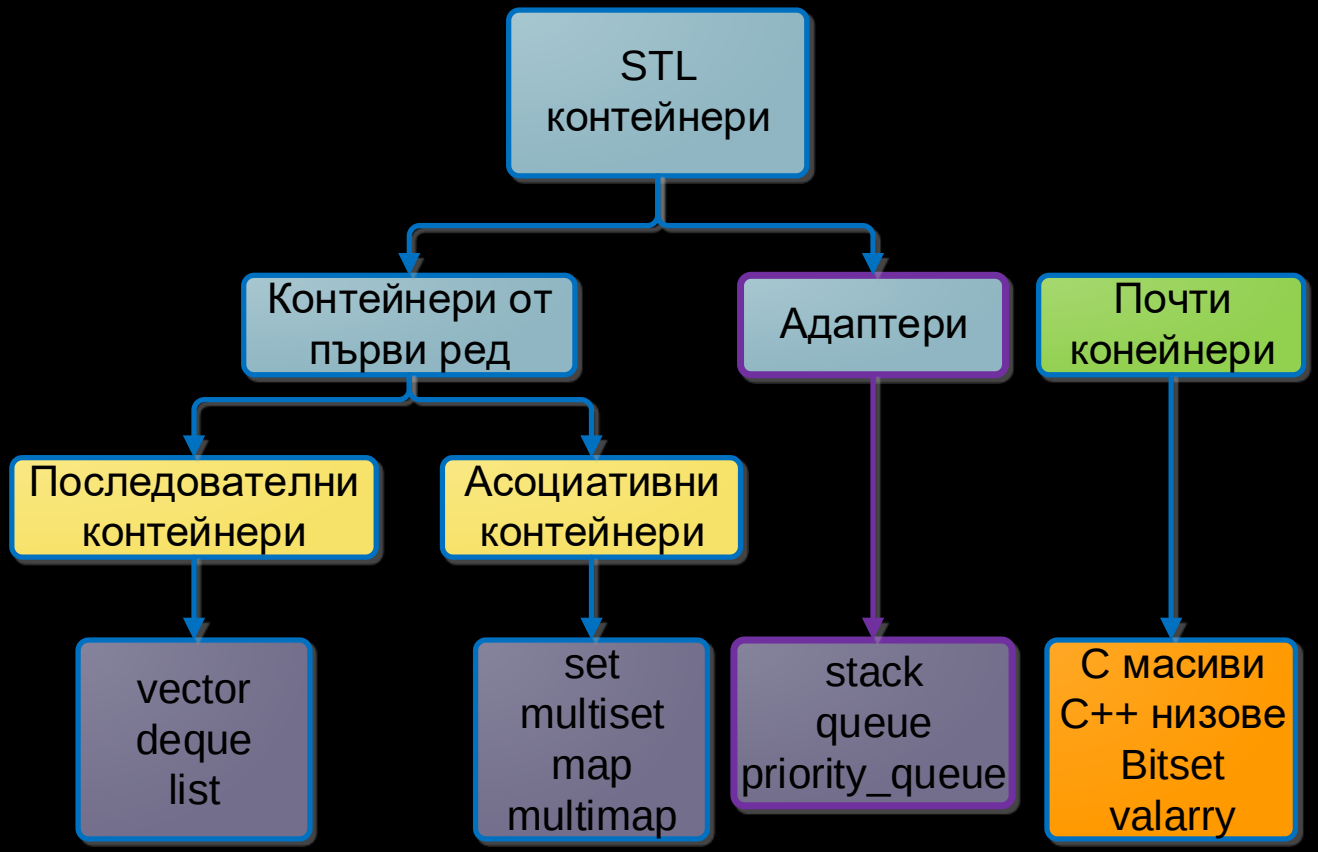
ВИДОВЕ КОНТЕЙНЕРИ



ВИДОВЕ КОНТЕЙНЕРИ



ВИДОВЕ КОНТЕЙНЕРИ



STL АДАПТЕРИ

- Адаптерите са контейнери, за реализацията на които се използват други контейнери.
- Всеки контейнер адаптер има свой интерфейс. Неговата структура и реализация се определят от съответен последователен контейнер.
- В STL са реализирани три адаптера : **stack, queue и priority_queue.**
- Заглавните файлове на контейнерите в STL са: **<Vector>, <list>, <deque>, <queue>, <stack>, <map>.**

STL АДАПТЕРИ

- Операциите общи за всички контейнери, са следните:

  - ✓ конструктори по подразбиране и копиране;
  - ✓ деструктор;
  - ✓ текущ брой на елементите на контейнер: **size**
  - ✓ проверка за празнота на контейнер: **empty;**
  - ✓ релационни операции: **operator<,operator<=,operator>,operator>+,operator ==, operator!=;**
  - ✓ присвояване: **operator=;**
  - ✓ размяна елементите на два контейнера**: swap**

ИТЕРАТОРИ В STL

- Достъпът до компонентите на всяка структура от данни е основна операция.
- Тя е необходима, за да се добави нова компонента, за да се отстрани компонента, да се прочете текущата стойност на компонента, да се промени текущо съдържание на компонента, да се търсят компоненти удовлетворяващи определено условие, да се обходят всички компоненти на структурата и др.
- Ето защо за всеки контейнер в STL има съответни операции за достъп до компонентите им.
- Тези операции са проектирани на базата на единни принципи, което ги прави използваеми не само в рамките на дадения контейнер, но и при STL алгоритмите, опериращи с контейнери посредством т.н. итератори.

ИТЕРАТОРИ В STL

- За итераторите в STL може да се каже, че те са „скрит” вид указатели, които предоставят унифициран достъп до компонентите на контейнерите.
- За всеки контейнер има дефинирани съответни итератори, които са класове шаблони.
- Всеки итератор за даден контейнер е обект, определен от класа на контейнера и неговия базов тип.
- Стойността на итератор е позиция на компонента от съответния контейнер. Чрез текущата стойност на итератор може да се получи и съответната стойност на компонентата на контейнера.

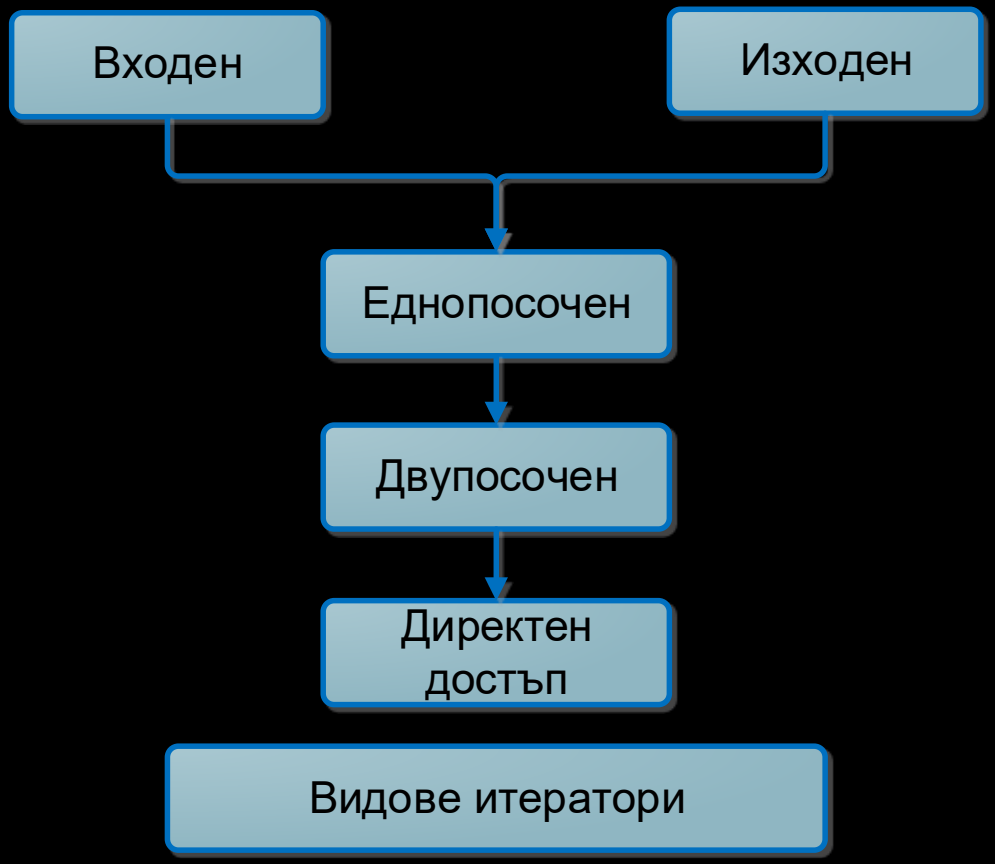
ИТЕРАТОРИ В STL

- Във всички контейнери, без адаптерите, се поддържат итератори, в които са дефинирани поне следните две функции:
  - ✓ begin(): функция която връща итератор, позициониран на първата компонента на контейнера;
  - ✓ end(): Функция , която връща итератор, позициониран една позиция след последната компонента на контейнера.
- Ще започнем разглеждането на итераторите в STL с представянето им в контейнера vector.

ВИДОВЕ ИТЕРАТОРИ В STL

- Входен итератор
- Изходен итератор
- Еднопосочен итератор
- Двупосочен итератор
- Итератор с пряк достъп
- Предефинирани итератори
- Итератори за въвеждане и извеждане на данни

ЙЕРАРХИЯ НА ОПЕРАЦИИТЕ НА РАЗЛИЧНИТЕ ВИДОВЕ ИТЕРАТОРИ



ЙЕРАРХИЯ НА ОПЕРАЦИИТЕ НА РАЗЛИЧНИТЕ ВИДОВЕ ИТЕРАТОРИ

- Всеки вид контейнер, записан по-ниско в таблицата, (схемата) притежава операциите на всички контейнери, които са над него.
- Различните видове контейнери използват различни категории итератори.
- Съответствието между контейнери и итератори, които се използват с тях е дадено по-долу:
  - ✓ итератори с пряк достъп: vector, deque
  - ✓ двупосочни итератори: list, set, multiset, map и multimap
  - ✓ не се поддържат итератори за: stack, queue, priority_queue;

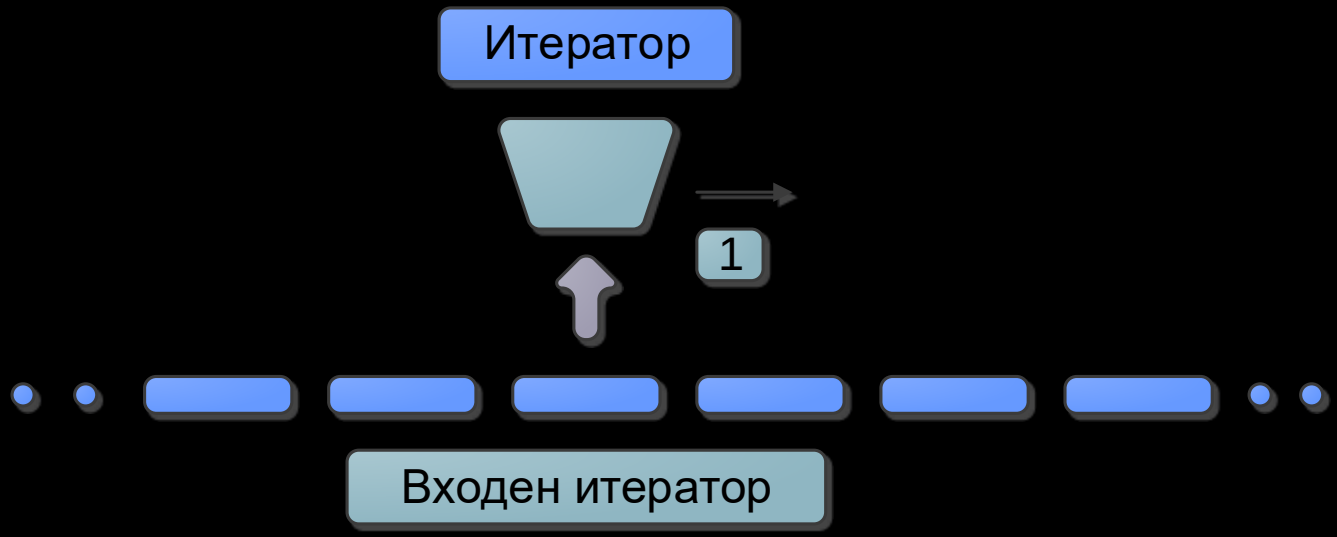
ВХОДЕН ИТЕРАТОР

- Входният итератор (input) се използва за достъп и последователно „четене" стойностите на контейнер.
  - ✓ Ако **p** е входен итератор за даден контейнер, то стойността на текущата компонента на контейнера, т.е. тази, която се сочи *от* **p**, се дава чрез израза ***p**.
  - ✓ Тук операция „*" е същата , която се използва при указателите в аналогични случаи.
  - ✓ Стойността ***p** на компонента на контейнера може да се присвои на променлива (обект) **x** от базовия тип **x** на контейнера:

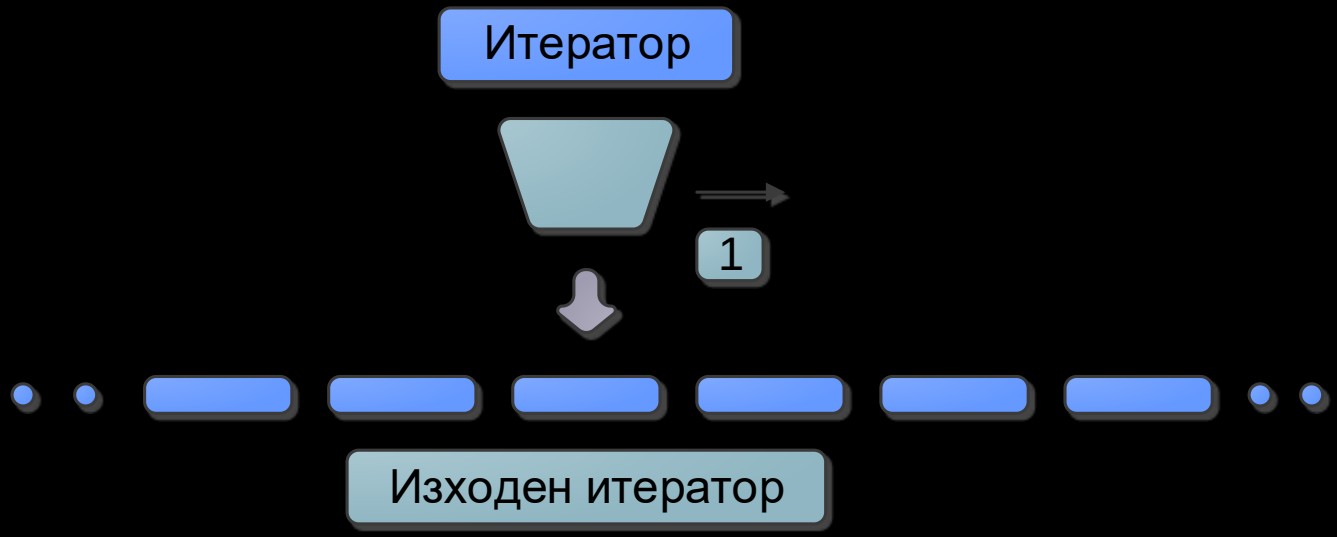
x = * p

- „Придвижването" на итератора по компонентите на контейнера се открива чрез сравняване на итератора с друг итератор, за който се знае, че „сочи" към несъществуващ елемент, непосредствено след последния.

ВХОДЕН ИТЕРАТОР



ИЗХОДЕН ИТЕРАТОР



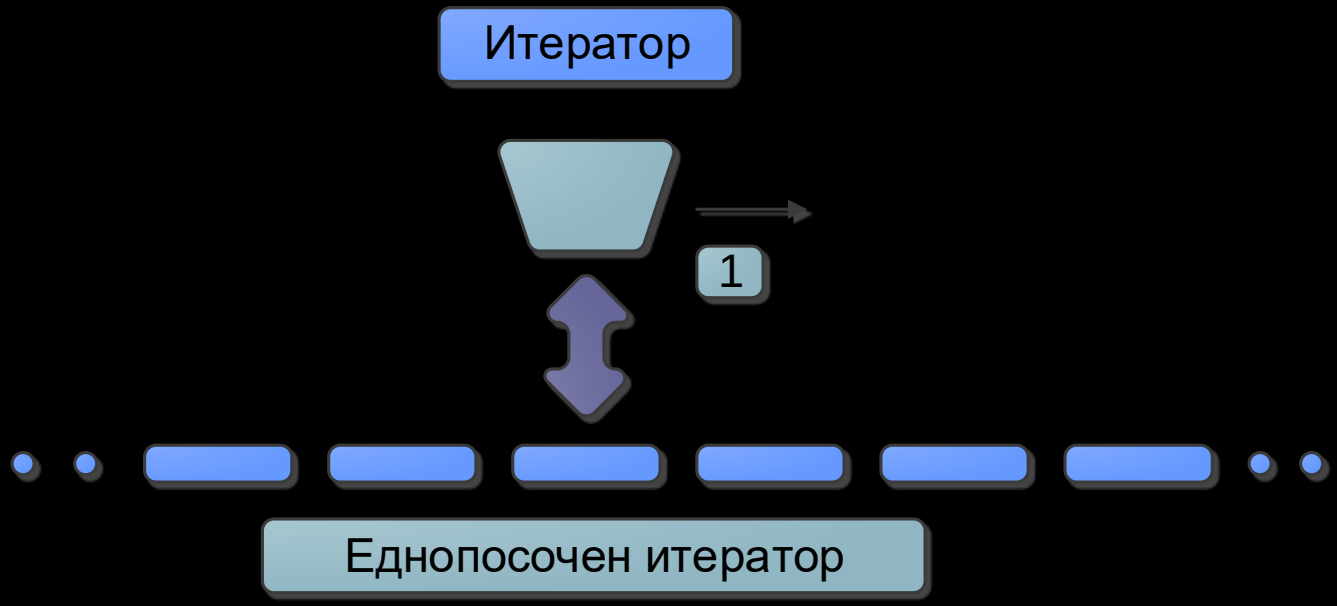
ИЗХОДЕН ИТЕРАТОР

- Изходният итератор (output) се използва за достъп и последователно присвояване (записване) на стойности на компонентите на контейнер.
  - ✓ Ако **p** е изходен итератор за даден контейнер, то на текущата компонента на контейнера *** p** може да се присвои стойност чрез оператор за присвояване *** p =x;.**
- Придвижването на итератора по компонентите на контейнера става само в едната посока и се извършва чрез операцията „**++**”.

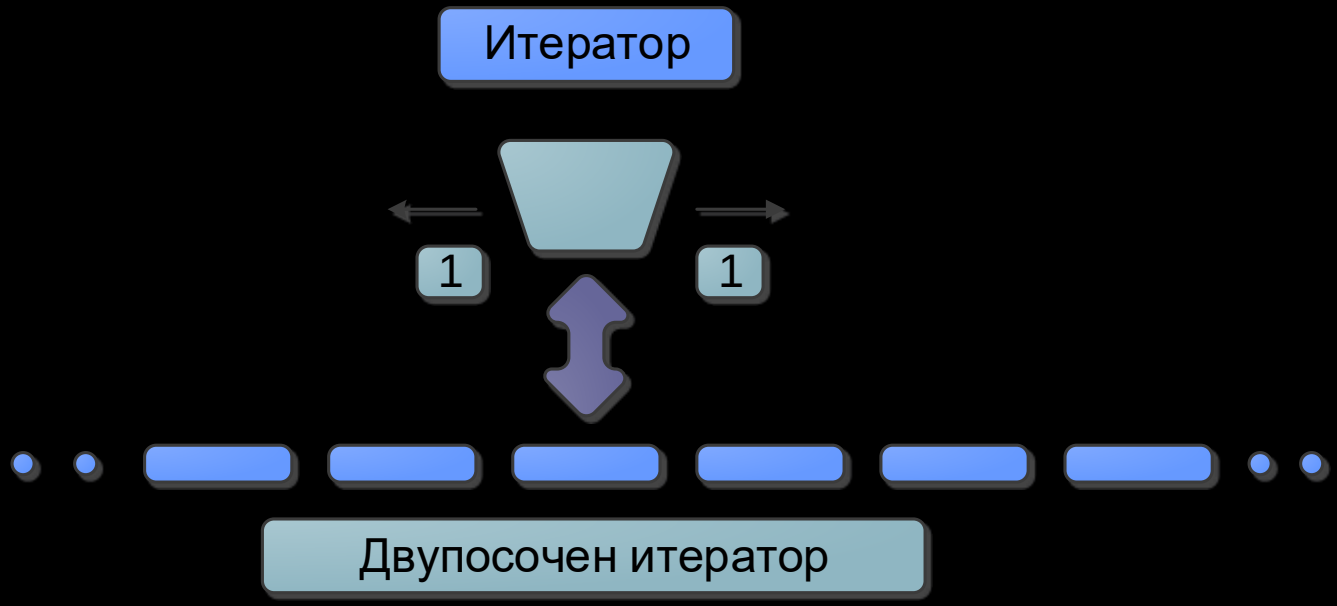
ЕДНОПОСОЧЕН ИТЕРАТОР

- Еднопосочният итератор (foreward) комбинира възможностите на входния и изходния итератор.
- Той се използва за изчитане или записване на елементите на контейнера елемент по елемент, придвижвайки се с една позиция само в една посока.
- Два двупосочни итератора **p и q** , дефинирани в рамките на един и същи контейнер, могат да се сравняват за равенство **p==q** и неравенство **p !=q**.
- Те са равни само, ако са позиционирани върху един и същи елемент на контейнера или/и двата сочат към елемента непосредствено след последния елемент на контейнера.
- Итератори от различни контейнери не би следвало да се сравняват.

ЕДНОПОСОЧЕН ИТЕРАТОР



ДВУПОСОЧЕН ИТЕРАТОР



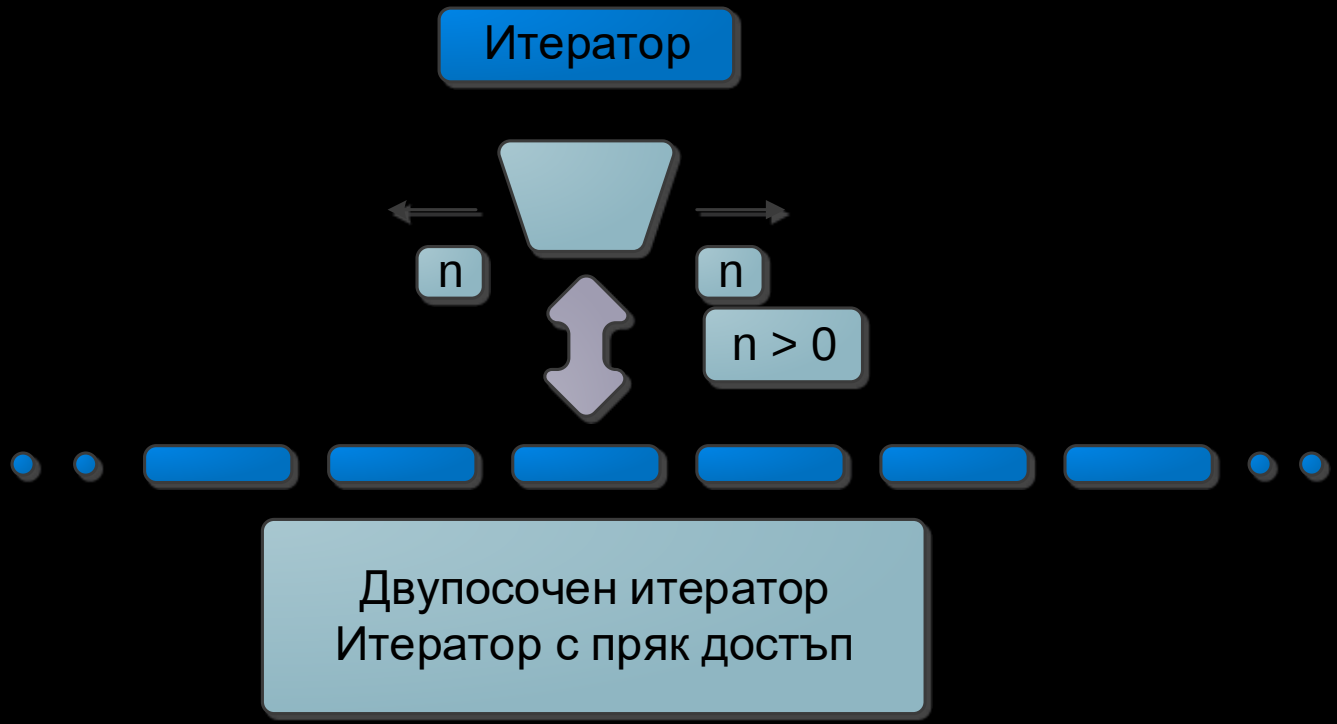
ДВУПОСОЧЕН ИТЕРАТОР

- Двупосочният итератор (bidirectional) се използва за четене на елементи в контейнера така, както това се извършва и с еднопосочните итератори.
- За разлика от тях двупосочният итератор може да извършва тези операции като се придвижва свободно и в двете посоки.
- За целта се използват операциите „++” при придвижване в права посока и „- -” в обратна посока.

ИТЕРАТОР С ПРЯК ДОСТЪП

- Итераторът с пряк достъп има всички възможности на двупосочния итератор, но в допълнение на това може да извърши достъп до произволен елемент и в двете посоки.

ИТЕРАТОР С ПРЯК ДОСТЪП



ПРЕДЕФИНИРАНИ ИТЕРАТОРИ

- В STL има четири вида предефинирани итератори и те са: iterator, const_iterator, reverse_iterator и const_reverse_iterator.
- Тези итератори са дефинирани във всеки контейнер от първи ред. Деклариране на итератор с име it за контейнера vector<int>

Vector<int>::iterator it;

- Итераторът const_iterator е подобен на iterator, но с тази разлика че той не допуска промяна на елемент от контейнера. С други думи това е итератор, с който могат само да се четат елементи от контейнера, но те не могат да се променят чрез него.
- Такива например са функциите print и printRev, които служат само за отпечатване на елемент на контейнера, без да се извършва някаква промяна в стойността им, то естествено е те да се декларират като константни итератори.
- Итераторите reverse_iterator и const_reverse_iterator се използват за достъп до елементите на контейнер, като обхождането им става в обратен ред. Чрез итератора от първия вид елементите на контейнера могат да се променят, докато с втория това не се допуска.

ИТЕРАТОРИ ЗА ВЪВЕЖДАНЕ И ИЗВЕЖДАНЕ НА ДАННИ

- Входните и изходните потоци от данни са сходни с последователностите, които се съхраняват и обработват от последователните контейнери.
- По тази причина , като аналогия на итераторите за последователните контейнери в STL са създадени и два итератора за вход и изход на еднотипни потоци от данни. Итераторът за вход е istream_iterator, а за изход – ostram_iterator.

АЛГОРИТМИ В STL

Кратък преглед на алгоритмите в STL

- Всеки контейнер има свои специфични операции, но има и такива, които са общи за всички контейнери, като например търсене на елемент, сортиране на елементите на контейнер, намиране на максимален или минимален елемент, заместване на елемент с друг, отстраняване на елемент, копиране на контейнер или на части от него и още много други.
- Авторите на STL са решили всички такива операции да бъдат изнесени в самостоятелна библиотека и реализирани по начин, независещ от конкретен контейнер.
- Така се формира философията и създава библиотека от STL алгоритми. Тя съдържа над 70 функции.

ФУНКЦИЯТА SORT

- Функцията sort подрежда елементите на контейнера в интервала [first…last], като използва операцията за сравнение „<”.
- За да се сортира **v**, обръщението към функцията трябва да е:

```
sort (v.begin(), v.end());
```

ФУНКЦИЯ FIND

- Функция **Find** извършва търсене на елемент в контейнер.

```
Iter find(Iter first, Iter last, const T& value);
```

- Тук **Iter** е входен итератор за контейнера, в който се извършва търсенето.
- Функцията find извършва последователно търсене в интервала [first…last] и връща стойност от тип итератор, който е позиция на елемент в контейнера.

WINDOWS TEMPLATE LIBRARY

- **Windows Template Library** (**WTL**) е безплатен софтуер, обектно-ориентирана C++ библиотека с шаблони за разработка на Win32 приложения.
- Той е разработен предимно като лека алтернатива на класовете на Microsoft Foundation и се основава на ATL (Active Template Library) на Microsoft , друг лек API, широко използван за създаване на COM и ActiveX библиотеки.

- WTL осигурява поддръжка за внедряване на различни елементи на потребителския интерфейс, от рамка и изскачащи прозорци, до MDI (multi-document interface) , стандартни и общи контроли, общи диалогови прозорци, листове със собственост и страници, GDI (Graphics Device Interface) обекти и други често срещани елементи на потребителския интерфейс, като прозорци с превъртане, прозорци за разделяне, ленти с инструменти и командни ленти.
- Основната цел на WTL е да достави малък и ефективен код, като същевременно осигурява обективен модел на по-високо ниво и по-гъвкав за разработчиците.

MICROSOFT FOUNDATION CLASS LIBRARY

ВЪВЕДЕНИЕ

- Може да се използва средата Microsoft Visual C++ за писане на всякакъв тип програми, които могат да бъдат написани на C или C++.
- Предоставя някои улеснения за управление на всеки един стадий от разработваната програма : Създаване на сорс кода , построяване (компилиране и свързване) на кода, тестване, дебъгване и оптимизиране на кода.

CARCHIVE CFILE

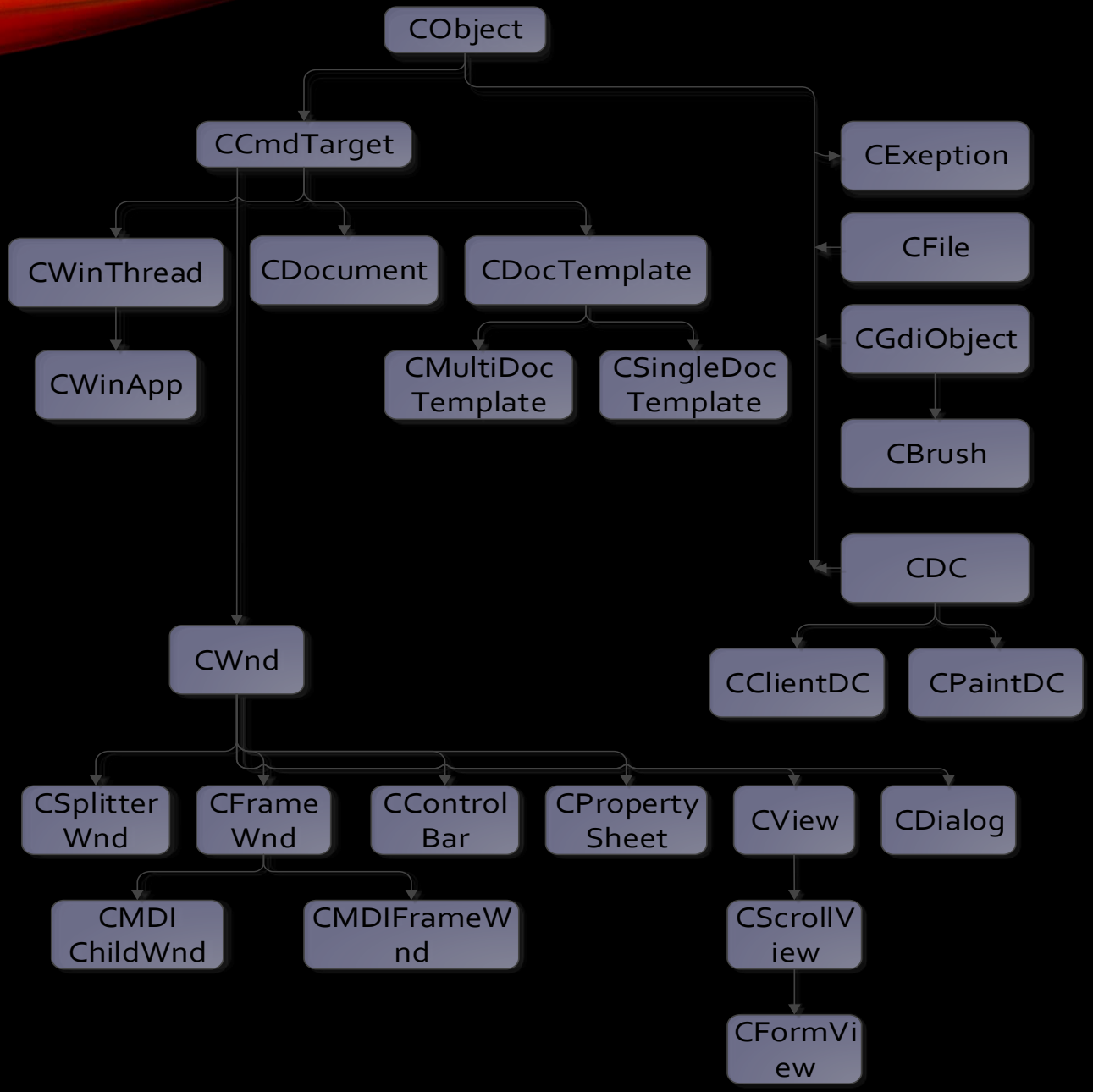
- В сърцето на механизма на **MFC** сериализацията е класът **CArchive**.
- Един обект на архив е абстракция за сериализиране на данните, на което и да е място, способно да съдържа серийни данни.
- Архивът може да се използва за сериализиране във файл, структура данни в оперативната памет и дори в клипборда на Windows.
- Когато за сериализиране на данни във файл се използва **CArchive** , архива се прикачва към обект от клас **CFile** , представляващ реалния файл на диска.

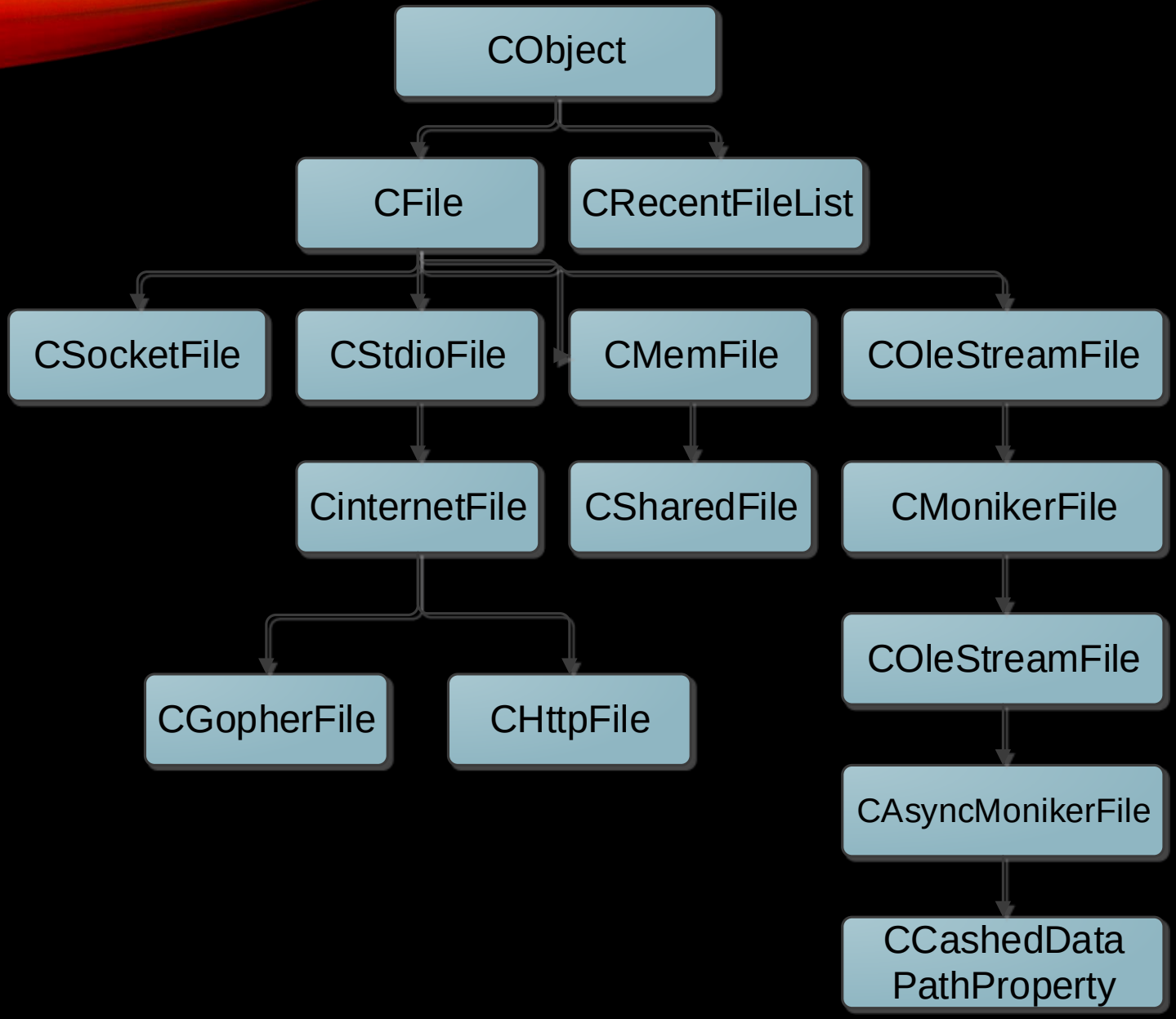
CARCHIVE CFILE

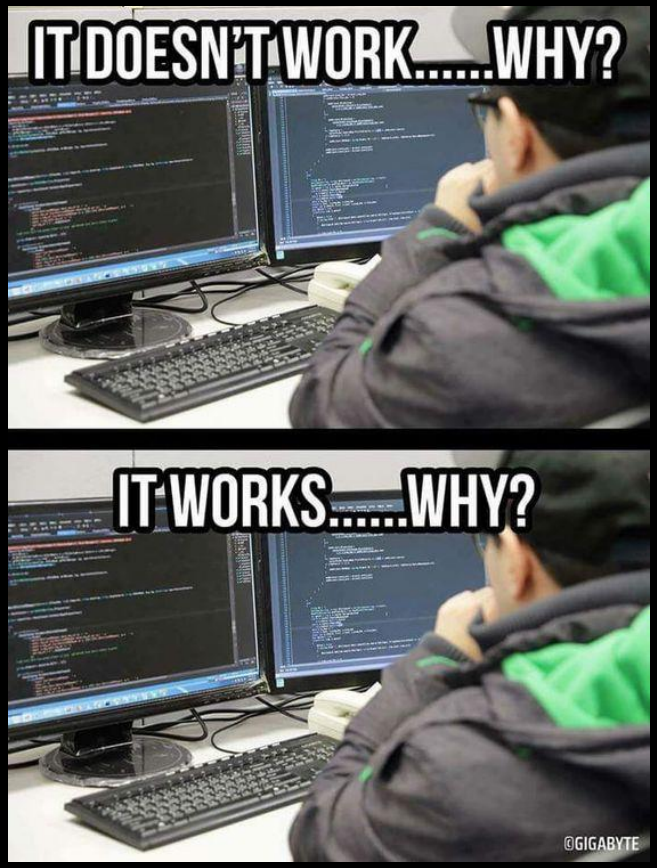
- **CFile** обхваща файлов идентификатор на операционната система, чрез който могат да се извършват всякакви входно/изходни операции.
- **CFile** директно поддържа небуфериран двоичен вход/изход към диска. Терминът небуфериран показва, че потребителят сам трябва да чете групи от байтове чрез директни извиквания на член функции от **CFile** – например **Read и Write.**
- Буфериранят вход/изход на файл е по-ефективен, защото четенето от диска е бавна операция. Идеята е много данни да се прочетат наведнъж в буфер-пространството в оперативната памет, към което сочат указатели. Една голяма операция за четене коства по-малко време отколкото много малки операции.

CARCHIVE CFILE

- След това могат да се направят много малки и много бързи прочитания от буфера.
- Терминът двоичен показва, че съхраняваните данни са в двоични байтове, а не текстови знакове; за съхранение и зареждане на текст, може да се използва производния на CFile клас **CStdioFile**.
- Комбинирането **на CArchive с CFile** е един от начините за получаване на буфериран вход/изход към файл.
- За да се създаде **CArchive** първо трябва да се създаде обект файл, производен на **CFile.** Подкласовете на **CFile** са показани на фигурата.


CObject
CCmdTarget
CExeption
CWinThread
CDocument
CDocTemplate
CFile
CGdiObject
CWinApp
CMultiDoc Template
CSingleDoc Template
CBrush
CDC
CWnd
CClientDC
CPaintDC
CSplitter Wnd
CFrame Wnd
CControl Bar
CProperty Sheet
CView
CDialog
CMDI ChildWnd
CMDIFrameW nd
CScrollV iew
CFormVi ew


CObject
CFile
CRecentFileList
CSocketFile
CStdioFile
CMemFile
COleStreamFile
CinternetFile
CSharedFile
CMonikerFile
CGopherFile
CHttpFile
COleStreamFile
CAsyncMonikerFile
CCashedData
PathProperty


IT DOESN'T WORK.....WHY?
IT WORKS.....WHY?
@GIGABYTE