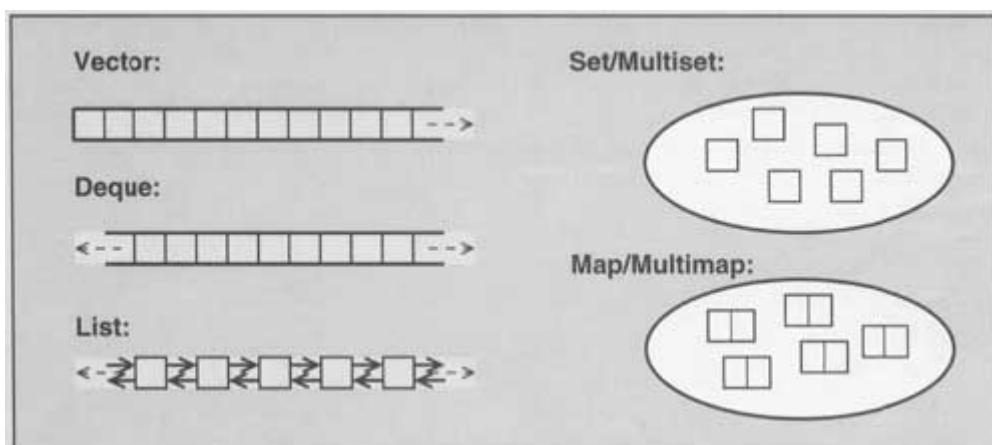


مكتبة القوالب المعيارية Standard Template Library

[الدرس الثاني :](#)

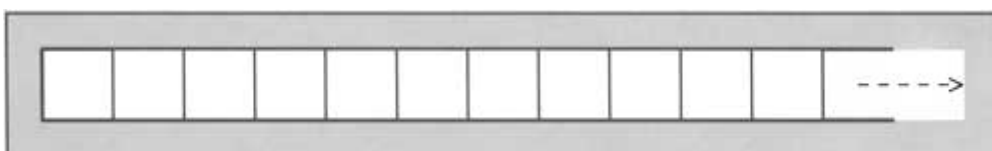
[الوعاء \(Container\) :](#)



عرفناه مسبقاً بـ " وهو يحتوي على عدد من الكائنات التي لها نفس النوع ويقوم بإدارتها بواسطة الطرق "method" الموجودة به " ويوجد عدة أمثلة على الوعاء مثل " vector , list , stack , set , map , multiset " طبعاً لو نظرت إلى هذه الأوعية تجد أنها تقسم إلى ثلاث فئات وسندرس بعضاً من خصائصها :

١ - الأوعية المتسلسلة (Sequence Containers) :

وهي الـ Container التي تكون البيانات المخزنة بها متسلسلة بدون أية علاقة تحكم بينها أي أنك أنت الذي تحدد بالضبط أين يخزن العنصر ولا توجد دالة تكون مسئولة عن ترتيب العناصر . والـ Containers هي vector و list و deque وسنأخذ نبذة سريعة عن كل واحد مع مثال تفصيلي عن الـ [vector](#)



يقوم مقام المصفوفة العادية " التقليدية " في C++ ومما يميزه أنه كتب بطريقة ذكية ليوفر أداء عالي مع الاستخدام الفعال للذاكرة ووجود طرق جاهزة فيه تساعد على إدارته ولو تلاحظ العلاقة بين الحجم الحالي للـ vector عن طريق الطريقة size والسعة الحالية له عن طريق الطريقة capacity تدرك معنا هذا الكلام ونأتي إلى الخصائص :

(١) [مكان تعريفه :](#) في الملف الرئيسي vector

(٢) [تعريفه في المكتبة :](#) على هذا النحو

```
template <class T,  
         class Allocator = allocator<Type> >  
class vector;
```

(٣) [الوصول إلى العناصر :](#) يتم الوصول إلى العناصر بشكل مباشر لأي عنصر متواجد فيه أي نفس خاصية المصفوفات مثلاً تستطيع تقول vector[10] على اعتبار v هو vector

(٤) [الإضافة والحذف منه :](#) يتميز بالسرعة في الإضافة الخلفي للبيانات حيث أنه سيكون الخيار الأفضل في ذلك . وتستطيع تطبيق هذه الطريقة باستخدام push_back حيث إنها تحجز خانة جديدة للعنصر إذا لم يوجد مكان في الذاكرة . ولا يوفر طريقة للإضافة من الأمام مثل الـ Containers الأخرى التي توفر push_front وهذا يرجع إلى طريقة كتابة implementation لها وله أيضاً ولكن vector يوفر طريقة تقليدية وهي insert يمكن من خلالها إدخال البيانات من الأمام . وأما الإضافة من المنتصف فهو مصفوفة " يعني يحتاج shift لكن لا تقلق هو من سيقوم بهذا العمل " . وبالنسبة للحذف فهو بالضبط مثل الإضافة.

(٥) الخيارات المتوفرة : يوفر خيارات عديدة للمستخدم فمثلاً إذا أردت عند الوصول إلى أي عنصر أن تدقق هل تجاوز المدى أم لا فاستخدم الطريقة at حيث يوجد بها range check أما إذا أردت زيادة السرعة على حساب الأمان فيمكن أن تستخدم العامل [] التقليدي فلا يوجد به . range check أيضاً إذا كان vector من نوع Boolean فيوجد له Implementation خاص فيه يوفر الذاكرة مع السرعة في الأداء مع وجود طرق إضافية فيه.

(٦) أمثلة على استخدامه :

أولاً : كيفية تعريفه :

الآن سنعرف vector من نوع int .

```
vector<int> v1; // لم تحدد حجمه بعد
// OR
// vector<int> v1(15); // الآن حددت عدد العناصر التي سوف يبدأ بها
```

طيب افرض كانت عندك مصفوفة حجمها ١٠ عناصر وأردت نقلها إلى vector هل ستحتاج إلى for-loop لنقلها ؟ الجواب لا . فقط في جملة واحدة تستطيع نقلها كيف !!! هذا هو الجواب :

```
int a1[]={10,11,12,13,14,15,16,17,18,19}; // هذه المصفوفة التي نريد نقلها
vector<int> v1(a1 , a1+10); // فقط بجملة واحدة
```

الذي قمنا به هو أننا أرسلنا عنوان المصفوفة " بداية المصفوفة " ونهايتها . ونفس الكلام ينطبق لو كانت البيانات موجودة في list و أردت أن تنقلها إلى vector أو العكس.

ثانياً : المرور عليه بالكامل :

توجد طريقة ذكية وسريعة لكي تمر على جميع العناصر وهي باستخدام الـ iterator لكن بما أنه سيكون موضوعنا في الدرس القادم فلن أشرحه في هذا الدرس وسأذكر طريقة أخرى وهي عن طريق الدالة size ، لنفرض أن الـ vector السابق أردت المرور عليه وطباعة الأعداد اللي أكبر من ١٥ فسوف يكون الكود كالتالي :

```
for(int i=0; i < v1.size(); i++)
    if(v1[i] > 15 )
        print(v1[i]);
```

طبعاً توجد دالة في المكتبة إسمها for_each تقوم بهذا الغرض لكن لم يحن الوقت لشرحها فهي تحتاج إلى فهم الـ iterator .

شرح مفصل لدوال الـ vector :

١ – البناء (Constructor) :

ملاحظة :

n هو عدد ، c و c1 هما vector ، beg و pos و end هي مؤشرات على العناصر

vector<Type> c	يكون vector فاضي
vector<Type> c1(c2)	يكون نسخة من vector آخر ونسخ جميع عناصره
vector<Type> c(n)	يكون vector مع n عنصر من البداية
vector<Type> c(n,elem)	يكون vector مع n نسخة من العنصر elem
vector<Type> c(beg,end)	يكون vector مع نسخ عناصر وعاء آخر

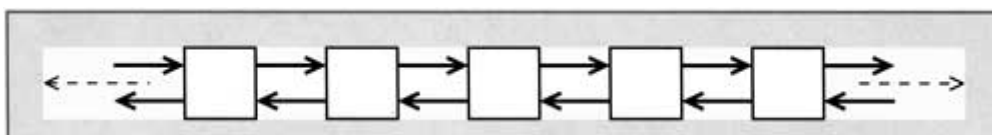
٢ – الطرق الغير معدلة :

c.size()	ترجع عدد العناصر الفعلية الموجودة في الـ vector
c.empty()	تعيد true إذا كان الـ vector فاضي و false غير ذلك
c.max_size()	يعيد لك عدد العناصر الممكن حجزها في الذاكرة
capacity()	يعيد لك عدد العناصر الممكن إستيعابها قبل عمل reallocation

٣ - الطرق المعدلة :

<code>reserve (n)</code>	تكبير سعة الـ vector بعدد n
<code>c1 = c2</code>	اسناد جميع عناصر c2 إلى c1
<code>c1.swap (c2)</code>	التبديل بين عناصر c1 و c2
<code>c.at (idx)</code>	يعيد العنصر صاحب الموقع idx ولكن يطلق إستثناء إذا idx أكبر من حجم الـ vector
<code>c[idx] </code>	يعيد العنصر صاحب الموقع idx بدون تشييك
<code>c.front ()</code>	يعيد أول عنصر
<code>c.back ()</code>	يعيد آخر عنصر
<code>c.insert (pos, elem)</code>	يضيف elem في المكان pos
<code>c.insert (pos, n, elem)</code>	يضيف n نسخة من elem في المكان pos
<code>c.push_back (elem)</code>	يضيف elem إلى نهاية الـ vector
<code>c.pop_back ()</code>	يزيل آخر عنصر
<code>c.erase (pos)</code>	يزيل عنصر من مكان pos
<code>c.resize (num)</code>	يعدل حجم الـ vector سواء بالتكبير او التصغير
<code>c.clear ()</code>	يزيل كل العناصر

List



هي عبارة عن قائمة مترابطة مضاعفة (doubly linked list) توفر دوال كثيرة تساعد في اختصار الوقت في كتابة الأكواد وهي أكثر من الدوال المتوفرة للـ vector ونمر على خصائصها :

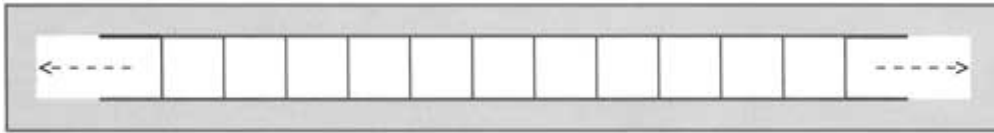
- (١) مكان تعريفه : في الملف الرئيسي list :
- (٢) تعريفه في المكتبة : على هذا النحو

```
template <class T,
        class Allocator = allocator<Type> >
class list;
```

(٣) الوصول إلى العناصر : إذا كنت قد درست linked list في مادة Data Structures فستعرف أن أشد عيوبها أنك لا تستطيع الوصول إلى العنصر مباشرة بل تحتاج إلى التحرك من بداية القائمة حتى العنصر المطلوب عبر جملة تكرار أو أكثر من جملة واحدة لكي تصل إلى العنصر المطلوب لكن توجد دالة تسمى advance في الملف الرئيسي algorithm ترسل لها المؤشر iterator " سيأتي شرحه بالتفصيل " مع عدد من العناصر التي تريد أن تتخطاها .

(٤) الإضافة والحذف منه : تعتبر أفضل وعاء بشكل عام في الإضافة والحذف من حيث السرعة من أي مكان فهي لا تحتاج إلى shift للعناصر ، فقط تكوين Node جديدة و من ثم ربطها ببقية القائمة .

(٥) الخيارات المتوفرة : توفر طرق أكثر من المتوفر في الوعاء vector فمثلاً توجد الطريقة merge التي تدمج قائمتين و sort و reverse وكلها يتضح عملها من اسمها .



هو اختصار لـ double-ended queue ومن اسمه يتضح أن له نهايتين و يجمع ما بين سرعة وفعالية الـ vector في العمليات وبين سهولة ومرونة الـ list في الإضافة و الحذف ويمكن الإضافة فيه من الأمام أو الخلف . ونمر على الخصائص :

(١) مكان تعريفه : في الملف الرئيسي deque :

(٢) تعريفه في المكتبة : على هذا النحو

```
template <class T,  
        class Allocator = allocator<Type> >  
class deque;
```

(٣) الوصول إلى العناصر : يعتبر مثل الـ vector ويدعم جميع ما يدعم الـ vector من الوصول المباشر والعشوائي للعناصر " لكن في الحقيقة فالوصول يتغرق أكثر من خطوة لكنها لا تذكر " .

(٤) الإضافة والحذف منه : يدعم ما يدعمه الـ list و الـ vector حيث يمكن الإضافة من الأمام عن طريق push_front أو الحذف أيضاً عن طريق pop_front ويمكن أيضاً الإضافة من الخلف عن طريق push_back أو الحذف عن pop_back أما الإضافة من الوسط أو الحذف فإنه يقوم بعملية shift للعناصر نظراً لك implementation المكتوب فيه "طبعاً الـ implementation معقد جداً"

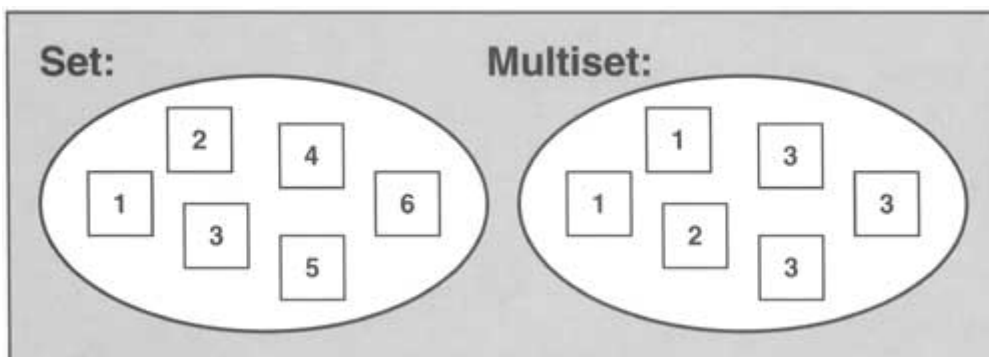
(٥) الخيارات المتوفرة : لا يوفر خيارات إضافة أكثر من أنه يمثل حل وسط ما بين الـ list و vector في الاستخدام حيث يوفر ما عجز عنه الـ vector وهي الإضافة من الأمام ويوفر الوصول العشوائي والمباشر للعناصر الذي عجزت عنه الـ list

٢ - الأوعية الترابطية (Associative Containers) :

وهي الأوعية التي تكون بياناتها مرتبة على أساس المفتاح للكائن وقد يكون المفتاح هو نفسه قيمة الكائن مثل ما هو موجود في set و multiset أو يكون منفصل عن القيمة مثل ما هو موجود في map و multimap ولكن الترتيب الافتراضي فيها هو تصاعدي حيث تتولي ترتيب العناصر الدالة less الموجودة في الملف الرئيسي functional ويمكنك تغييرها واستخدام بدل منها greater أو greater_equal أو تكتب الدالة الخاصة بك بس ملاحظة هذه لو أكون دقيق ليست دالة هي " functor راجع الدرس السابق "

والآن سنأخذ نبذة سريعة عنهما :

Set & multiset



هي عبارة عن وعاء مرتبة عناصره على حسب القيمة من الأصغر إلى الأكبر " هذا الترتيب الافتراضي " و implementation الداخلي هو عبارة عن red black tree ويعتبر مفيد لمن يبحث عن وعاء يقوم بترتيب عناصره بشكل تلقائي . وللمعلومية الفرق الوحيد في set و multiset هو أن set لا تقبل تكرار العناصر فيها أما الـ multiset فهي ترحب بالعناصر المتكررة . وهذه خصائصه :

- (١) مكان تعريفه : في الملف الرئيسي set :
- (٢) تعريفه في المكتبة : على هذا النحو

```
template <class T,
          class Compare = less<Type>,
          class Allocator = allocator<Type> >
class set;

template <class T,
          class Compare = less<Type>,
          class Allocator = allocator<Type> >
class multiset;
```

- (٣) الوصول إلى العناصر : وفر ما وفرته الـ list فقط من الوصول الغير مباشر إلى العناصر
- (٤) الإضافة والحذف منه : تعطي مرونة عالية في الإضافة فهي عبارة عن tree لكن في الحذف قد يستغرق وقت أطول
- (٥) الخيارات المتوفرة : توفر خيارات عديدة منها مثلاً إمكانية كتابتك دالة المقارنة الخاصة فيك ولا تحتاج إلى استخدام الدالة الافتراضية less ترك للمبرمج حرية الاختيار إما بدعم التكرار أو بمنعه .
- (٦) أمثلة على استخدامه :

أولاً : كيفية تعريفه :

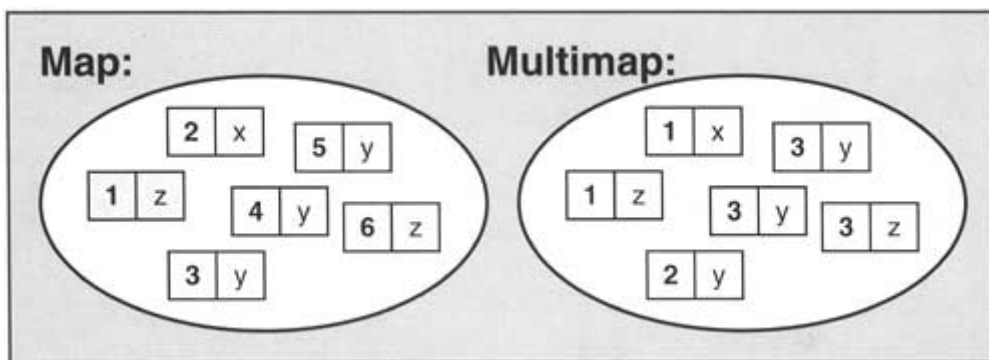
الآن سنعرف set من نوع int

```
set <int> s1; // أيضاً ترتيبه من الأصغر إلى الأكبر لم تحدد حجمه بعد
// OR
// set<int> s1(15); // الآن حددت عدد العناصر التي سوف يبدأ بها
```

ماذا الآن لو أردت أن أجعل ترتيب العناصر من الأكبر إلى الأصغر فالحل سهل وهو أستخدم الـ greater functor و هذا الكود يوضح الفكرة

```
set <int , greater<int> > s1;
```

[map & multimap](#)



هي عبارة عن وعاء مرتبة عناصره على حسب المفتاح من الأصغر إلى الأكبر " هذا الترتيب الافتراضي " فكل عنصر تضيفه إلى المجموعة يجب أن يحتوي على key و value أي مفتاح وقيمة وأيضاً الفرق بينهما أن multimap يقبل تكرار المفاتيح أما map فلا يقبل تكرارها . و بالنسبة للخصائص التالية :

- (١) مكان تعريفه : في الملف الرئيسي map :
- (٢) تعريفه في المكتبة : على هذا النحو

```
template <class Key, class T,
          class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,Type> > >
class map;

template <class Key, class T,
          class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,Type> > >
class multimap;
```

(٢) الوصول إلى العناصر: نفس ما قلنا عن الـ set.

(٤) الإضافة والحذف منه: نفس ما قلنا عن الـ set.

(٥) الخيارات المتوفرة: تعطي خيارات عديدة أفضلها إمكانية البحث عن العنصر عبر مفتاحه بطريقه سهله وبسيطة عن طريق العامل [] فمثل إذا كان المفتاح من نوع string وكانت m من نوع map وأردت البحث عن مفتاح "ali" فما عليك إلا كتابة m["ali"] فقط وهذه تذكرني بالمصفوفة المترابطة الموجودة في php ولغات السكربت .

(٦) أمثلة على استخدامه:

أولاً: كيفية تعريفه :

الآن سنعرف map مفاتيحه من نوع int وقيمة من نوع string

وأيضاً ترتيبه من الأصغر إلى الأكبر لم تحدد حجمه بعد //

```
map <int, string> s1;
```

ماذا الآن لو أردت أن أجعل ترتيب العناصر من الأكبر إلى الأصغر فالحل سهل وهو أستخدم الـ greater functor وهذا الكود يوضح الفكرة

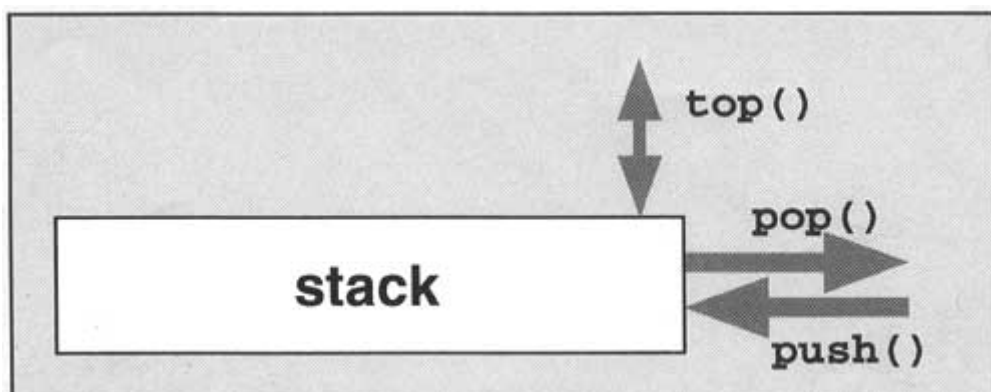
```
map <int, string , greater<int> > s1;
```

لاحظ إن الـ greater يطلب وسيط بنفس نوع الـ key وليس الـ value
ملاحظة: تسمى الـ Associative Containers و Sequence Containers بـ "first-class Containers الأوعية الأولية" وهذا لأنها تدعم الـ Iterator من جهة ومن جهة أخرى فهي تعتبر framework لمكتبة STL.

٢ - Container Adapters (لم أعرف ترجمتها إلى كلمة مستساغة):

وهي الأوعية التي تقوم بأداء مهم معينة ولها شروط وقيود لاستخدامها وهي في الأساس تعتبر تغليف لأحد الـ Sequence Containers و الوعاء الافتراضي هو deque لهم كلهم إلا واحد منهم يستخدم vector وهذه الأوعية هي stack و queue و priority_queue طبعاً جميعها لا توفر وصول مباشر ولا غير مباشر أو عشوائي عن طريق الـ Iterator لأنها ببساطة لا تدعمه ولا يمكنك الوصول إلى العناصر إلا عن طريق الدوال المتوفرة لذلك سأشرح كل دالة باختصار ونبدأ على بركة الله :

: stack



هو وعاء يطبق قاعدة الذي يدخل أخيراً يخرج أولاً (Last In First Out (LIFO) وخصائصه هي كالتالي :

(١) مكان تعريفه: في الملف الرئيسي stack

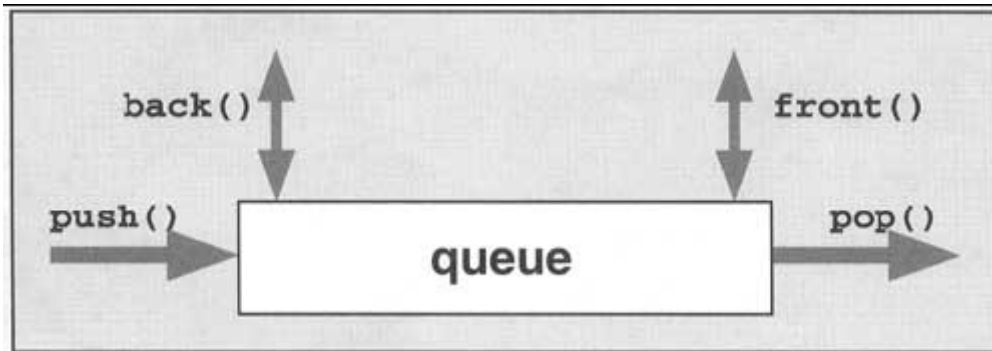
(٢) تعريفه في المكتبة: على هذا النحو :

```
template <class Type,
        class Container = deque<Type> >
class stack;
```

(٣) الطرق التي يدعمها :

push : وهذه تستقبل العنصر وتضيفه إلى stack
 pop : وهذه يحذف آخر عنصر تمت إضافته إلى stack
 top : وهذه يعيد آخر عنصر تمت إضافته إلى stack دون حذفه

queue :



هو وعاء يطبق قاعدة الذي يدخل أولاً يخرج أولاً (FIFO) First In First Out وخصائصه هي كالتالي :

(١) مكان تعريفه : في الملف الرئيسي queue

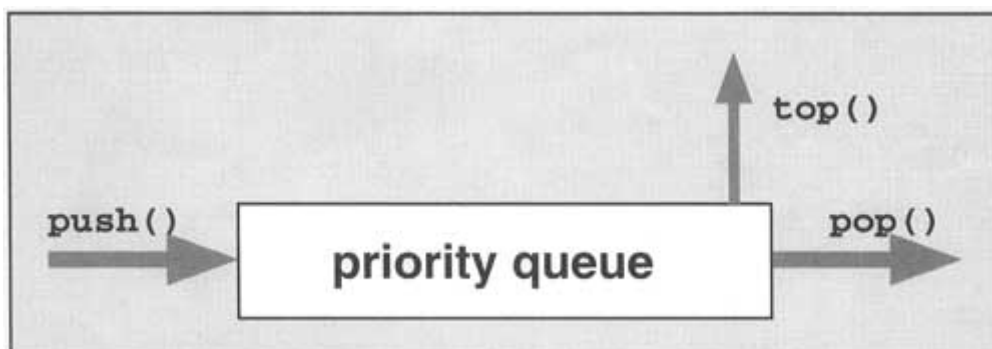
(٢) تعريفه في المكتبة :

```
template <class T,
        class Container = deque<Type> >
class queue;
```

(٣) الطرق التي يدعمها :

push : وهذه تستقبل العنصر وتضيفه إلى queue
 pop : وهذه يحذف أول عنصر تمت إضافته إلى queue
 front : وهذه يعيد أول عنصر تمت إضافته إلى queue دون حذفه.
 back : وهذه يعيد آخر عنصر تمت إضافته إلى queue دون حذفه.

priority queue :



هو وعاء يطبق قاعدة أن العنصر الأعلى أولوية هو من سيكون في الأمام وخصائصه هي كالتالي :

(١) مكان تعريفه : في الملف الرئيسي queue

(٢) تعريفه في المكتبة :

```
template <class T,  
        class Container = vector<Type>,  
        class Compare = less<typename Container::value_type> >  
class priority_queue;
```

(٣) الطرق التي تدعمها :

push : وهذه تستقبل العنصر وتضيفه إلى priority_queue
pop : وهذه يحذف أعلى عنصر أولوية تمت إضافته إلى priority_queue
top : وهذه يعيد أعلى عنصر أولوية تمت إضافته إلى priority_queue دون حذفه.

وعموماً stack و queue و priority_queue كلهم يشتركون في دالتين هما size و empty وهي معروفة ولا تحتاج إلى شرح.
وأخيراً : توجد في المكتبة شبة أوعية " near-Containers " وهي string و bitset و valarray لكن ربما سأشرح فقط string لكن ليس الآن.

Lesson written by (B.M.S)