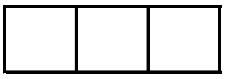
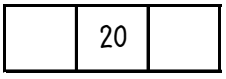
動的配列クラス

使いにくいC++配列を、使いやすくしてくれる便利なコンテナクラス。
ここでは、よく使用される、vector、list、mapを使用していく。

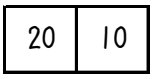
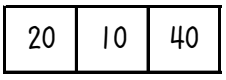
■ 固定配列の不便なところ

・リサイズできない
・関数の返り値、引数として使えない

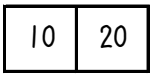
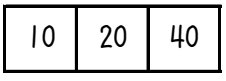
⇒ 動的配列クラスであれば、この不便さを解消できます
（ しかし、速度面からすると、固定配列の方が分がある場合が多い ）

int mData[3];	⇒		3枠確保。
mData[1] = 20;	⇒		

■ vector
配列の最後尾へのデータを追加や、削除が高速で処理できる。
一方、途中挿入などは処理が遅い。

std::vector<int> mVec;	⇒		確保無し
mVec.push_back(20);	⇒		枠追加
mVec.push_back(10);	⇒		枠追加
mVec.push_back(40);	⇒		枠追加

■ list
どこに挿入・削除するにしても、それなりの速度で処理できる。
vectorと使い勝手は似ているので、用途によって使い分けると良いです。

std::list<int> mList;	⇒		確保無し
mList.push_back(20);	⇒		枠追加
mList.push_front(10);	⇒		枠追加
mList.push_back(40);	⇒		枠追加

■ map
vector、listのように、動的に配列の要素数を変更することができますが、
mapは連想配列になりますので、配列の構造が大きくことなります。
※連想配列　…　キーに対応する値の組(ペア)とする配列

```
std::map<int, Unit*> mMap                    ⇒  確保無し

mMap.emplace(10, new Unit("すえ"));          ⇒
mMap.emplace(55, new Unit("かね"));          ⇒
mMap.emplace(30, new Unit("勇者"));          ⇒
auto findUnit = mMap.find(55);               ⇒
(*findUnit).first;                           ⇒  55
(*findUnit).second.GetName();                ⇒  かね
```

key	10
value	Unit

key	10	55
value	Unit	Unit

key	10	55	30
value	Unit	Unit	Unit

key	10	55	30
value	Unit	Unit	Unit

■ ループのやり方

```
std::vector<Unit*> mUnitVct;
for (Unit* unit : mUnitVct) {
   unit->Update();
}

std::list<Unit*> mUnitList;
for (Unit* unit : mUnitList) {
   unit->Update();
}

std::map<int, Unit*> mUnitMap;
for (std::pair<int, Unit*> p : mUnitMap) {
   p.second->Update();
}
```

※mapは、常にkeyとvalueのpair(ペア)で操作する
　firstがkey、secondがvalue

■ 削除のやり方

```
int mData[3];
```
⇒ 固定配列の場合は、削除ができない。。。
配列の要素数は、3のまま。

『動的』配列であれば、簡易的に削除可能となる。

```
for (veUnit = unitVct->begin(); veUnit != unitVct->end();) {
   if (IsCollision(*veUnit))
   {
      (*veUnit)->Damage(DAMAGE);
      if ((*veUnit)->IsDead())
      {
         veUnit = unitVct->erase(veUnit);      ※削除
         continue;
      }
   }
   ++veUnit;
}
```

注意！！　以下、エラーになる削除のやり方

```
①size = unitVct->size();
②for (int i = 0; i < size; i++)
   {
      // vectorは要素番号の指定が可能
      tmpUnit = (*unitVct)[i];
      if (IsCollision(tmpUnit))
      {
         tmpUnit->Damage(DAMAGE);
         if (tmpUnit->IsDead())
         {
            // i番目を削除
      ③    unitVct->erase(unitVct->begin() + i);
         }
      }
   }
```

① 要素数を取得
② 要素の数分、ループ
③ ユニットが死亡したら、削除

⇒ ③で要素の削除を行うと、当然、動的配列の要素数、構成が変わる。
ループ中の要素数が変わると、ループ数や、参照している要素が変わるため、
参照エラーとなる。
なので、ループ中に削除を行わず、
１回目のループで削除対象を記録しておき、その記録を元に、
２回目のループで削除を行う方法が安全ではあるが、
２回ループする分、処理速度が落ちるので、
ループ中で、上手いこと参照箇所を調整しつつ、削除を行うのが、
最初のコードになります。

これで、動的配列の基本動作である、追加や削除ができるようになると思います。