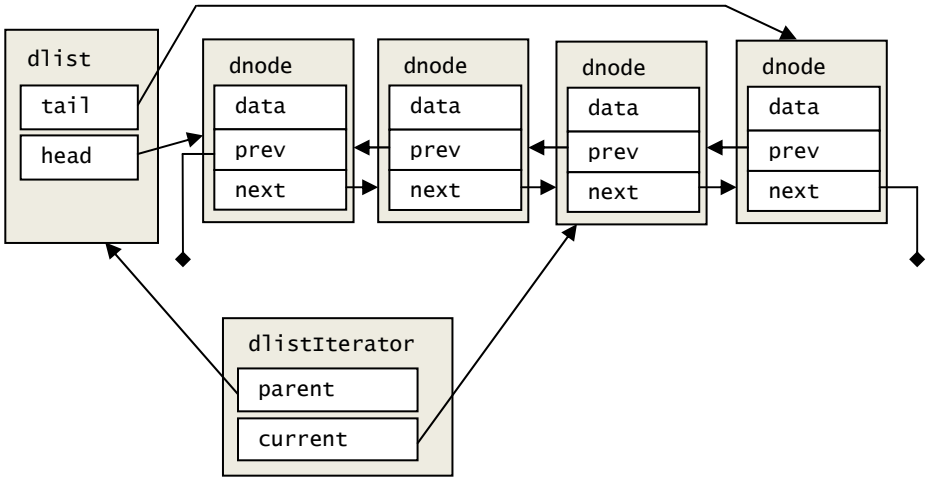
Overloading the -> and * operators in iterators.

The overloaded -> and * operators can be a bit tricky to understand. The aim of this document is to go through them step by step using the `dlist` code from the lecture notes.

Before starting, the following diagram shows a doubly linked list and an iterator pointing at it.



- The head value in the `dlist` points to the first dnode in the list. `tail` points to the last dnode in the list.
- Each dnode has a variable `prev` which points to the previous dnode and a variable `next` which points to the next dnode in the list. By pointing we mean it contains the address of the particular dnode.
- Each dnode also contains the `data` we are storing in the list. Since this is template code it is of type `dataType`. When we compile the program, the compiler will figure out if it is an `int`, or `double` or some other `class` we have defined.
- `parent` in the `dlistIterator` points to the `dlist` it is iterating through. `current` points to a particular dnode within the linked list.

Overloading the -> operator

The following code overloads the arrow operator.

```
dataType* operator -> () const {
   if (current == NULL) {
     throw std::invalid_argument(
        "Attempting to dereference NULL in dlistIterator");
   }
   return &(current->data);
}
```

If we examine the return type (`dataType*`) this says we are going to return an address of a `dataType`. `dataType` is what we are storing in out list.
We can now examine the line `return &(current->data);`

`current` points to a dnode. `current->data` accesses the `data` component in the dnode. `&(current->data)` gets the address of the `data` component in the dnode that `current` is pointing to.

Since `data` is of type `dataType`, and we are returning the address of it, this matches the return type of the overloaded operator. The following diagram illustrates

All this allows us to do the following.

```
class myData
{
    private:
       // some data
    public:
       void printData() { /* print the data in myData */ }
};

int main()
{
   dlist<myData> myList;

   // fill myList with myData

   dlistIterator<myData> itr;
   for (itr = myList.begin(); itr != myList.end(); itr++)
   {
      itr->printData();
   }

}
```

`itr` is pointing to a dnode in `myList`. However, we are using the arrow operator to call the `printData` function of the `myData` stored in the dnode.

Overloading the * operator

The following code overloads the dereference operator

```
dataType& operator * () const {
   if (current == NULL) {
     throw std::invalid_argument(
        "Attempting to dereference NULL in dlistIterator");
   }
   return current->data;
}
```

The overloaded operator is returning a reference to a `dataType`. This is not the same as returning an address. This is accomplished with the line `return current->data;`

The following code is an example of how it would be used.

```
class myData
{
    private:
       // some data
    public:
       void printData() { /* print the data in myData */ }
};

int calculateData(myData &theData)
{
   // do something with the data and return an int result
}

int main()
{
   dlist<myData> myList;

   // fill myList with myData

   dlistIterator<myData> itr;
   for (itr = myList.begin(); itr != myList.end(); itr++)
   {
      cout <<  calculateData(*itr) << "\n";
   }

}
```

The `calculateData` function requires a `myData`. The overloaded arrow operator returns an address of a `dataType`, not a reference, so we can't use that. This is where the overloaded dereference operator is useful.

The following diagrams illustrates the difference between the two overloaded operators