# Lecture 10

- Various variable-length storage
  - Doubling-Array
  - R-value reference
  - Return-Value Optimization

# Various Variable-Length Data Storage

- Doubling-Array: Understanding std::vector.
- R-Value reference: Moving is faster than copying.
- Return-value optimization
- Hash Set
  - Simple case : Hash Code == Hash Key
  - General case : Hash Code == A function of hash key

# Doubling-Array

- Background technique of std::vector
- The idea: Want an array class that grows its length on demand.

# Minimum variable-length array class

```
#ifndef VLARRAY_IS_INCLUDED
#define VLARRAY_IS_INCLUDED
/* { */

template <class T>
class VLArray
{
private:
    long long int len;
    T *dat;
public:
    VLArray();
    ~VLArray();
    void CleanUp(void);
    long long int GetN(void) const;

    VLArray(const VLArray <T> &incoming);
    VLArray <T>&operator=(const VLArray <T> &incoming);
    void CopyFrom(const VLArray <T> &incoming);
};
template <class T>
VLArray<T>::VLArray()
{
    len=0;
    dat=nullptr;
}
template <class T>
VLArray<T>::~VLArray()
{
    CleanUp();
}
template <class T>
void VLArray<T>::CleanUp(void)
{
    if(nullptr!=dat)
    {
        delete [] dat;
        dat=nullptr;
    }
    len=0;
}

template <class T>
long long int VLArray<T>::GetN(void) const
{
    return len;
}

template <class T>
VLArray<T>::VLArray(const VLArray <T> &incoming)
{
    dat=nullptr;
    CopyFrom(incoming);
}
template <class T>
VLArray <T> &VLArray<T>::operator=(
    const VLArray <T> &incoming)
{
    CopyFrom(incoming);
    return *this;
}
template <class T>
void VLArray<T>::CopyFrom(const VLArray <T> &incoming)
{
    if(this!=&incoming)
    {
        CleanUp();
        if(0<incoming.len)
        {
            this->len=incoming.len;
            this->dat=new T [incoming.len];
            for(decltype(len) i=0; i<incoming.len; ++i)
            {
                this->dat[i]=incoming.dat[i];
            }
        }
    }
}

/* } */
#endif
```

- This VLArray class is not useful.
- Want to be able to resize it.

```
template <class T>
void VLArray<T>::Resize(long long int newSize)
{
   if(0>=newSize)
   {
      CleanUp();
   }
   else
   {
      T *newDat=new T [newSize];
      for(decltype(len) i=0; i<len && i<newSize; ++i)
      {
         newDat[i]=dat[i];
      }
      if(nullptr!=dat)
      {
         delete [] dat;
      }
      dat=newDat;
      len=newSize;
   }
}
```

- Then, can Add an element to the array.

```
template <class T>
void VLArray<T>::Add(const T &newElem)
{
   Resize(len+1);
   dat[len-1]=newElem;
}
```

```
template <class T>
T &VLArray<T>::operator[](long long int idx)
{
   return dat[idx];
}
template <class T>
const T &VLArray<T>::operator[](long long int idx) const
{
   return dat[idx];
}
```

# Problem

- Adding N elements costs $O(N^2)$ time.
- $O(N^2)$ comes from number of elements added times number of elements that need to be copied every time the array is resized.

- How about making it $O(N\log N)$?
- Instead of growing the array every time a new element is added, grow the array when the number of element crosses the boundary of $2^n$.

- Add a new variable:

```
long long int nAvailable;
```

- In the constructor,

```
nAvailable=0;
```

- In CleanUp,

```
nAvailable=0;
```

- Modify CopyFrom:

```
this->len=incoming.len;
this->dat=new T [incoming.len];
```

```
this->len=incoming.len;
this->nAvailable=incoming.nAvailable;
this->dat=new T [incoming.nAvailable];
```

- Modify Resize as:

```
template <class T>
void VLArray<T>::Resize(long long int newSize)
{
   if(0>=newSize)
   {
      CleanUp();
   }
   else
   {
      long long int newNAvailable=1;
      while(newNAvailable<newSize)
      {
         newNAvailable*=2;
      }

      T *newDat=new T [newNAvailable];
      for(decltype(len) i=0; i<len && i<newSize; ++i)
      {
         newDat[i]=dat[i];
      }
      if(nullptr!=dat)
      {
         delete [] dat;
      }
      dat=newDat;
      len=newSize;
      nAvailable=newNAvailable;
   }
}
```

- Add Grow function, and modify Add function as:

```
template <class T>
void VLArray<T>::Grow(void)
{
   if(0==nAvailable) // Mean empty
   {
      dat=new T [1];
      nAvailable=1;
   }
   else
   {
      nAvailable*=2;
      T *newDat=new T[nAvailable];
      for(decltype(len) i=0; i<len; ++i)
      {
         newDat[i]=dat[i];
      }
      if(nullptr!=dat)
      {
         delete [] dat;
      }
      dat=newDat;
   }
}

template <class T>
void VLArray<T>::Add(const T &newElem)
{
   if(nAvailable<=len)
   {
      Grow();
   }
   dat[len]=newElem;
   ++len;
}
```

Grow function doubles the number of available elements, but does not change the length.

- Let’s see how many times the copy operator is called if a function returns an instance of VLArray.
- A local VLArray instance is copied to the temporary instance, which is then copied to the final destination.

- It can be made efficient by moving instead of copying the data.

- Moving: Transferring the ownership of the resource from one instance to another.

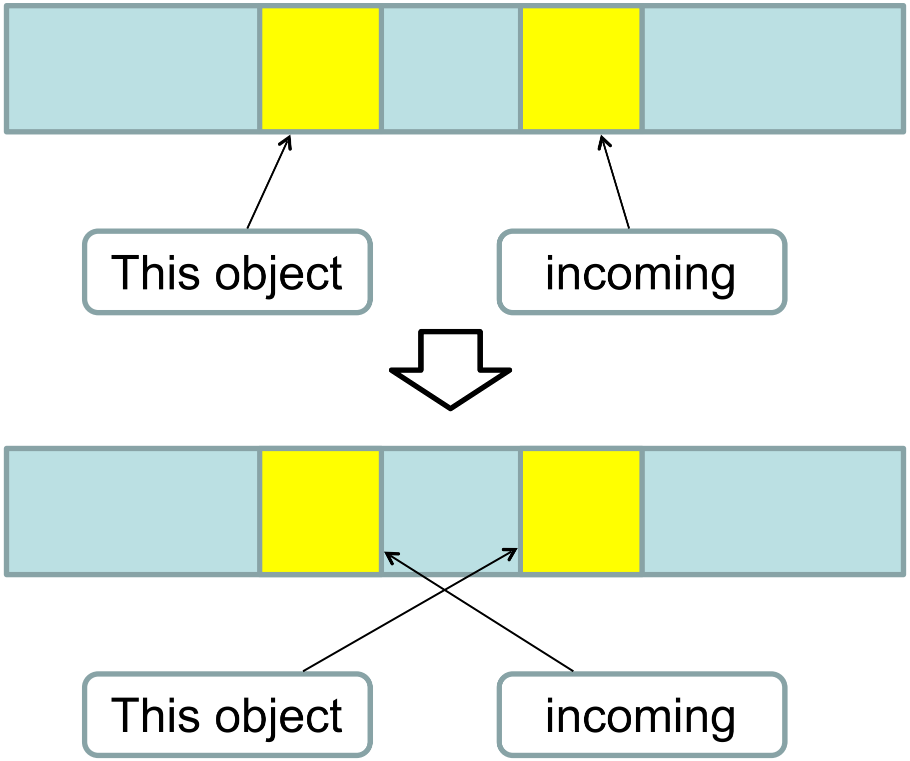
- After releasing the ownership, the object must be in the state that the destructor can safely destroy the object.

- Adding Swap function.
- Both this object and the incoming object are in the state that the destructor can safely destroy.
- Ownership can be swapped.

```
template <class T>
void VLArray<T>::Swap(VLArray <T> &incoming)
{
    if(&incoming!=this)
    {
        auto dat=incoming.dat;
        auto len=incoming.len;
        auto nAvailable=incoming.nAvailable;

        incoming.dat=this->dat;
        incoming.len=this->len;
        incoming.nAvailable=this->nAvailable;

        this->dat=dat;
        this->len=len;
        this->nAvailable=nAvailable;
    }
}
```



# R-value reference

- Next question: When C++ can automatically take advantage of the moving?
- When the incoming object is an R-value.
- L-value and R-value:
  - L-value: If you say a=123;, a is on the left. Therefore, a is an L-value.
  - R-value: Something that cannot be an L-value. For example, a return value of a function is an R-value. You cannot substitute something to the return value.
- Use && for R-value reference.

- C++ will use the following moving constructor / operator when the incoming object is a R-value.

```
template <class T>
VLArray<T>::VLArray(VLArray <T> &&incoming)
{
   dat=nullptr;
   Swap(incoming);
}
template <class T>
VLArray <T> &VLArray<T>::operator=(VLArray <T> &&incoming)
{
   Swap(incoming);
   return *this;
}
```

# Return-Value Optimization

- If you write as:
  ```
  auto ary=MakeArray();
  ```
  the moving/copying functions may not even be called at all.
- Compile with and without optimization option to see the behavior.

- The following function:

```
VLArray <int> MakeArray(void)
{
   VLArray <int> ary;
   for(int i=0; i<100; ++i)
   {
      ary.Add(i);
   }
   return ary;
}
```

can be re-written as two separate functions:

```
VLArray <int> MakeArray(void)
{
   VLArray <int> ary;
   MakeArray(ary)
   return ary;
}

void MakeArray(VLArray <int> &ary)
{
   for(int i=0; i<100; ++i)
   {
      ary.Add(i);
   }
}
```

- When the optimizer sees a pattern:

```
auto ary=MakeArray();
```

- It automatically re-interpret as:

```
decltype(MakeArray()) ary;
MakeArray(ary);
```

- Small chance of getting a strange bug, if you assume that the variable is moved by a moving constructor or operator, and if a moving constructor or operator is doing something special.