

Creating a Simple Vector

- **“VecInt”**
 - A subset of “vector<int>”
 - See attached code listing
- **Use dynamic memory/pointers**
 - To implement varying size
- **A container ADT**
 - Similar to text, page 638
 - More complete implementation



Class Definition (1)

```
class VecInt
{
public:
  VecInt();
  explicit VecInt(unsigned int sz,
                 int val = 0);
  VecInt(const VecInt& v);
  ~VecInt();
  ...
  VecInt& operator=(const VecInt& right);
  ...
};
```

Class Definition (2)

```
class VecInt
{
public:
  ...
  void push_back(int val);
  unsigned int size() const;
  bool empty() const;
  void clear();
  int& operator[](unsigned int indx);
  const int& operator[]
    (unsigned int indx) const;
  ...
};
```

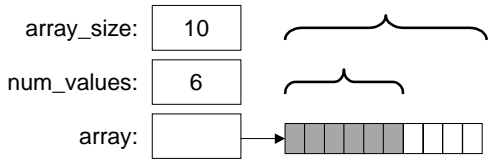
Class Definition (3)

```

class VecInt
{
...
private:
    unsigned int array_size;
    unsigned int num_values;
    int* array;
...
};

```

Managing the Array



array can be null, meaning no dynamic array; in this case, array_size and num_values must be zero.

No-Argument Constructor

```

VecInt::VecInt()
{ array_size(0), num_values(0), array(0) }
{
    // Initialization done in
    // initializer list.
}

```

array_size = 0;
num_values = 0;
array = 0;

Alternate Constructor

```

VecInt::VecInt(unsigned int sz, int val)
:array_size(sz),num_values(sz),array(0)
{
  if (array_size > 0)
  {
    array = new int [array_size];
    for (unsigned int indx = 0;
         indx < num_values; indx++)
    {
      array[indx] = val;
    }
  }
}

```

Copy Constructor (First Draft)

```

VecInt::VecInt(const VecInt& v)
{
  array = new int [v.num_values];
  array_size = v.num_values;
  num_values = v.num_values;

  for (unsigned int indx = 0;
       indx < num_values; indx++)
  {
    array[indx] = right.array[indx];
  }
}

```

Destructor

```

VecInt::~VecInt()
{
  // Destroy the dynamic array
  // if there is one.

  delete [] array;
}

```



Utility Functions

```

unsigned int VecInt::size() const
{
    return num_values;
}

bool VecInt::empty() const
{
    return (num_values == 0);
}

void VecInt::clear()
{
    num_values = 0;
}

```

Subscript Operators

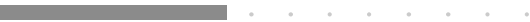
```

int& VecInt::operator[]
    (unsigned int indx)
{
    return array[indx];
}

const int& VecInt::operator[]
    (unsigned int indx) const
{
    return array[indx];
}

```

What about "const int&" return value?

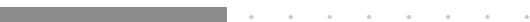


Adding a Vector Element

```

void VecInt::push_back(int val)
{
    ...
    // If there is room ...
    array[num_values++] = val;
}

```



Making More Room

```

...
if (array_size < (num_values + 1))
{
  int* old_array = array;
  if (array_size == 0) array_size = 1;
  array_size *= 2;
  array = new int [array_size];
  for (unsigned int indx = 0;
      indx < num_values; indx++)
  {
    array[indx] = old_array[indx];
  }
  delete [] old_array;
}

```

Why double the size?

Assignment Operator (First Draft)

```

...
VecInt& VecInt::operator=
    (const VecInt& right)
{
  delete [] array; // Destroy old
  array = new int [right.num_values];
  array_size = right.num_values;
  num_values = right.num_values;
  for (unsigned int indx = 0;
      indx < num_values; indx++)
    array[indx] = right.array[indx];
  return *this;
}

```

Self-Assignment Problem

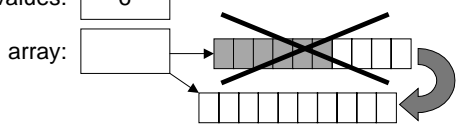
What if "x = x;"?

array_size:

num_values:

array:

- Delete old array
- Allocate new array
- Copy data



Would anyone really do this?

Checking Self-Assignment

```

VecInt& VecInt::operator=
    (const VecInt& right)
{
    {if (this != &right)}
    {
        ...
    }
    return *this;
}

```

What is this?

CopyData

- **Note similarity between**
 - Copy constructor
 - Assignment operator
- **Implement in one function**
- **Optimize**
 - Allocate new array only if:
 - Don't have one
 - Current one not big enough

