C++
Raymond Klefstad, Ph.D.
Dynamic Allocation of Arrays

Pointers and References

- they contain the address of some object
- allow access to that object
- can have multiple references to an object

```
int i = 10;
```

- a pointer contains the address of some object

```
int * p = & i; // & gives address of object i
```

- references do too, but are set only at construction time

```
int & j = i;
i = 50; // changes i directly
 *p = 60; // changes i indirectly
j = 70; // changes i indirectly
```

- pointers can be changed to point to other objects

```
int k = 20;
p = & k; // p now points to k
 *p = 60; // changes k indirectly
```

- references cannot be changed (after construction)

```
j = k; // i now gets value of k
```

- zero is used for null address (means pointing to nothing)

```
p = 0;
```

- indirection through zero is an error

```
 *p = 100; // should cause run-time error
```

- references can't be null
- "this" is actually a pointer to an object

Arrays and Pointers

- In C++, arrays are implemented as pointers to first element

```
int a[4];
int b[2];
int * p = a;
 *p = 10;
p[0] = 20;
p = b;
 *p = 30;
```

```
p[0] = 40;
```

- pointer arithmetic

```
p[1] = 70;
 *(p+1) = 70; // does the same thing
```

- character strings are arrays of characters

```
char s1[] = "Hello";
char * s2 = "Hello"; // not the same thing as s1
char * s3 = s1;
s3[0] = 'M'; // changes s1 to "Mello"
s2[0] = 'M'; // will give "segmentation violation"
```

Limitations of Fixed-Size Arrays

- size must be known at compile-time
- once it is allocated, array cannot grow
- size may depend on use
- dynamic allocation of an array gives us flexibility

Dynamic Allocation of Arrays

- until now, we allocated arrays on the stack
- operator new[] allocates array from heap (free store)
- dynamic array is deallocated (returned to heap) with operator delete[]
- new[] and delete[] go together

```
int main()
{
  int size = getFromUser( "How Many?" );
  int * a = new int[size];
  for ( int i = 0; i < size; ++i )
    a[i] = getFromUser( "Next Integer" );
  for ( int i = 0; i < size; ++i )
    cout << a[i] << endl;
  delete[] a;
  return 0;
}
```

Arrays within Classes

- low-level arrays are only for representation of a class
- constructor and destructor keep track of array sub-storage

```
class Vector
{
private:
  int maxLength;
  int * buf;
public:
  Vector( int newLength )
```

```
    : maxLength( newLength ), buf( new int[newLength] )
  {
  }
  ~Vector()
  {
    delete[] buf;
  }
};
int main()
{
  Vector v(10);  // v gets constructed here
  return 0; // v gets destructed here
}
```

The Copy Constructor

- allows construction from an existing object
- this object should be a duplicate (clone) of the existing object
- copy constructor takes one parameter
  - a reference to object of this class type
- EG copy constructor for Vector

```
class Vector
{
public:
  Vector( Vector & v )
    : maxLength( v.maxLength ), buf( new int[v.maxLength]
)
  {
    for ( int i = 0; i < v.maxLength; ++i )
      buf[i] = v.buf[i];
  }
};
```

---

operator index

- the index operator allows us to treat our class objects as arrays
- reference return value allow user to modify the object returned

```
class Vector
{
private:
  int maxLength;
  int * buf;
  bool inBounds( int i )
  {
    return i >= 0 && i < maxLength;
  }
public:
  int & operator [] ( int index )
  {
```

```
    assert( inBounds( index ) );
    return buf[index]; /// returns ref to buf[index]
  }
};
```

- ---

EG use of operator []

```
int main()
{
  Vector v(10);
  v[0] = 1;  /// used as L-value
  for ( int i = 1; i < v.length(); ++i )
    v[i] = v[i-1] * 2;
  cout << v[9] << endl; /// used as R-value
  return 0;
}
```

---

operator =

- the assignment operator allows us to assign instances of our class

```
class Vector
{
private:
  int maxLength;
  int * buf;
  void resizeTo( int newSize )
  {
    delete[] buf;
    buf = new int[newSize];
    maxLength = newSize;
  }
public:
  Vector & operator = ( Vector & v )
  {
    if ( maxLength != v.maxLength )
      resizeTo( v.maxLength );
    for ( int i = 0; i < maxLength; ++i )
      buf[i] = v.buf[i];
    return *this;
  }
};
```

---

operator ==

- the equality operator allows us to compare instances of our class

```
class Vector
{
private:
  int maxLength;
  int * buf;
```

```
public:
  bool operator == ( Vector & v )
  {
    if ( maxLength != v.maxLength )
      return false;
    for ( int i = 0; i < maxLength; ++i )
      if ( buf[i] != v.buf[i] )
        return false;
    return true;
  }
};
```

Pointer Caveats

- uninitialized pointer error
  - will lead to serious run-time errors
- garbage
  - memory associated with an object whose life has ended
- dangling reference error
  - a pointer to a dead object (garbage)
  - referencing such an object is an error (usually difficult to debug)
- memory leak error
  - storage lost (forever) due to missing ‘delete’
- redundant delete error
  - delete same storage more than once
- Wisdom: use of pointers should be restricted to class implementations

---

Extended Example

- a flexible list of integers

```
class IntList
{
private:
  int maxLength; // the capacity of this IntList
  int curLength; // the actual number of elements in this IntList
  int * buf; // base of the array of integers in this IntList
  const int DEFAULT_SIZE = 10;
  bool indexInBounds( int i )
  {
    return i >= 0 && i < curLength;
  }
  void increaseArrayTo( int newSize )
  {
    int * oldBuf = buf;
    buf = new int[newSize];
    copy( buf, oldBuf, curLength );
    maxLength = newSize;
    /// curLength stays the same
```

```
    delete[] oldBuf;
  }
  static void copy( int * to, int * from, int n )
  {
    for ( int i = 0; i < n; ++i )
      to[i] = from[i];
  }
```

- ---

continued

```
public:
  IntList( int newLength = DEFAULT_SIZE )
    : maxLength( newLength ),
      curLength(0),
      buf( new int[newLength] )
  {
    assert( newLength > 0 );
  }
  IntList( const IntList & l )
    : maxLength( l.maxLength ),
      curLength( l.curLength ),
      buf( new int[l.maxLength] )
  {
    copy( buf, l.buf, l.curLength );
  }
  int length()
  {
    return curLength;
  }
  void append( int value )
  {
    if ( curLength == maxLength )
      increaseArrayTo( maxLength + DEFAULT_SIZE );
    buf[curLength++] = value;
  }
  void print( ostream & out )
  {
    for ( int i = 0; i < curLength; ++i )
      out << buf[i] << ' ';
  }
```

- ---

continued

```
  int & operator [] ( int index )
  {
    assert( indexInBounds( index ) );
    return buf[index];
  }
  IntList & operator = ( IntList l )
```

```
  {
    if ( maxLength < l.curLength )
      increaseArrayTo( l.curLength );
    copy( buf, l.buf, l.curLength );
    curLength = l.curLength;
    return *this;
  }
  bool operator == ( IntList l )
  {
    if ( curLength != l.curLength )
      return false;
    for ( int i = 0; i < curLength; ++i )
      if ( buf[i] != l.buf[i] )
        return false;
    return true;
  }
  bool operator != ( IntList l )
  {
    return ! operator == ( l );
  }
```

- ---

continued

```
  int indexOf( int value )
  {
    for ( int i = 0; i < curLength; ++i )
      if ( buf[i] == value )
        return i;
    return -1;
  }
  void remove( int value )
  {
    int i = indexOf( value );
    if ( i != -1 )
      buf[i] = buf[--curLength];
  }
  IntList reverse()
  {
    IntList result(curLength);
    for ( int i = curLength - 1; i >= 0; --i )
      result.append( buf[i] );
    return result;
  }
  ~IntList()
  {
    delete[] buf;
  }
};
```

```
ostream & operator << ( ostream & out, IntList l )
{
  l.print( out );
  return out;
}
```

- ---

using IntList

```
int main()
{
  IntList myInts( 10 );
  for ( int i = 0; i < 7; ++i )
    myInts.append( i + 1 );
  for ( int i = 1; i < myInts.length(); ++i )
    myInts[i] *= myInts[i-1];
  IntList yourInts = myInts;
  cout << "My ints are " << myInts << endl;
  cout << "Your ints are " << yourInts << endl;
  if ( myInts == yourInts )
    cout << "and they're equal\n";
  IntList otherInts;
  if ( otherInts.length() == 0 )
    cout << "otherInts is empty\n";
  otherInts = yourInts;
  cout << "Other ints are " << otherInts << endl;
  otherInts = otherInts.reverse();
  cout << "Other ints are " << otherInts << endl;
  otherInts.remove( 720 );
  cout << "Other ints are " << otherInts << endl;
  if ( myInts != yourInts )
    cout << "and they're not equal\n";
  return 0;
}
```

- ---

The output

```
My ints are 1 2 6 24 120 720 5040
Your ints are 1 2 6 24 120 720 5040
and they're equal
otherInts is empty
Other ints are 1 2 6 24 120 720 5040
Other ints are 5040 720 120 24 6 2 1
Other ints are 5040 1 120 24 6 2
```

---

typedef

- allows you to name a new type

```
typedef int * IntPointer; /// IntPointer synonym for int *
IntPointer p = 0;  /// p is a pointer to an int
typedef char Buffer[20];
```

```
Buffer buf;  /// buf is an array of 20 char
```

---

const Parameters

- allows you to protect a parameter from accidental modification

```
void print( const int j )
{
  for ( int i = j; i >= 0; --j ) /// error is caught!
    cout << i;
}
```

- const with & is common for efficiency improvement

```
void print( const Coins & c )
{
  cout << c.total() << endl;
}
```

---

static member functions

- just like regular functions - no this parameter
- still defined within a class
- have class scope
- can access private parts of class members

Dynamic Allocation of Objects

- single objects are allocated from heap with operator new
- delete frees storage allocated with new
- new returns the address of the storage
- delete takes that same address and returns the storage to the heap

```
int main()
{
  Coins & cr = * new Coins(5,6,7,8);
  cout << cr.total();
  delete & cr;
}
```

- ---

  pointers are used more often than references with dynamic allocation
- for pointers, operator -> is used instead of operator . to select class members
  - p->member is same as (*p).member

```
int main()
{
     Coins * cp = new Coins(1,2,3,4);
    cout << cp->total();
    delete cp;
 }
```