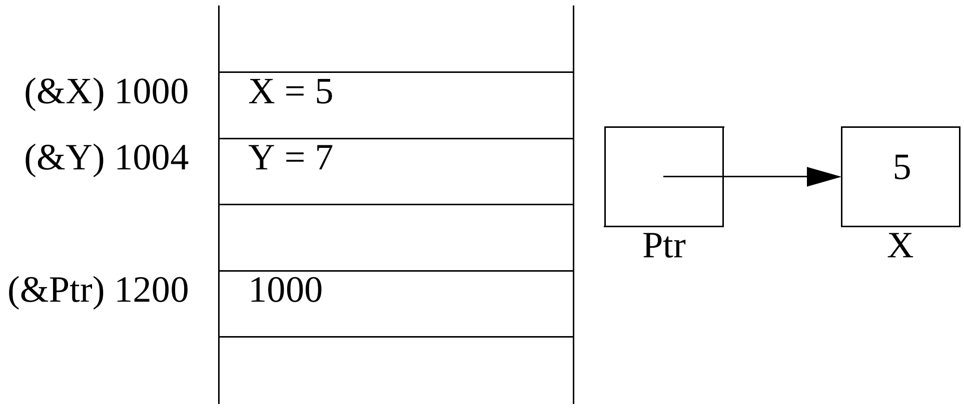
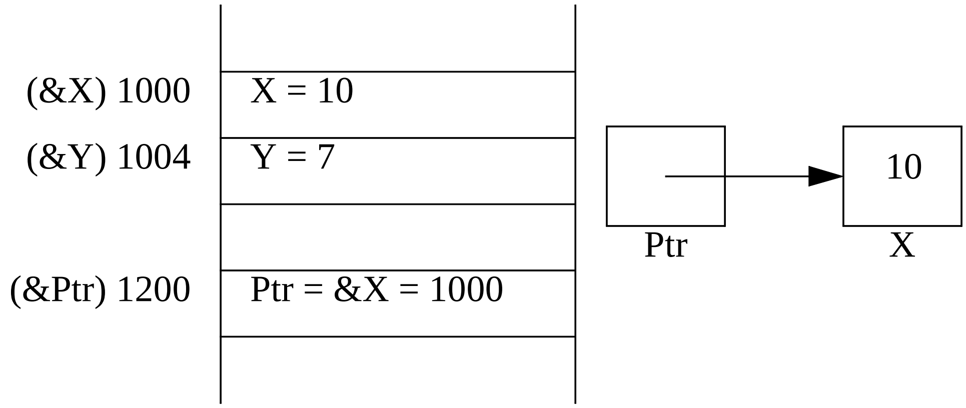


Chapter 1

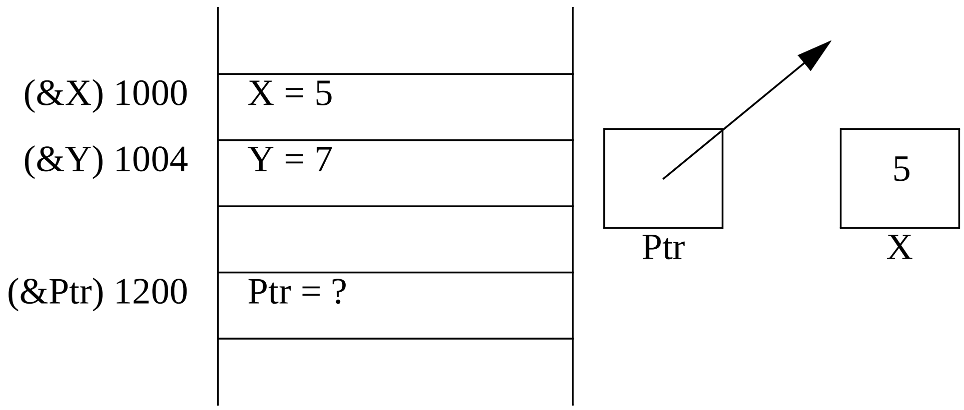
Pointers, Arrays, and Structures



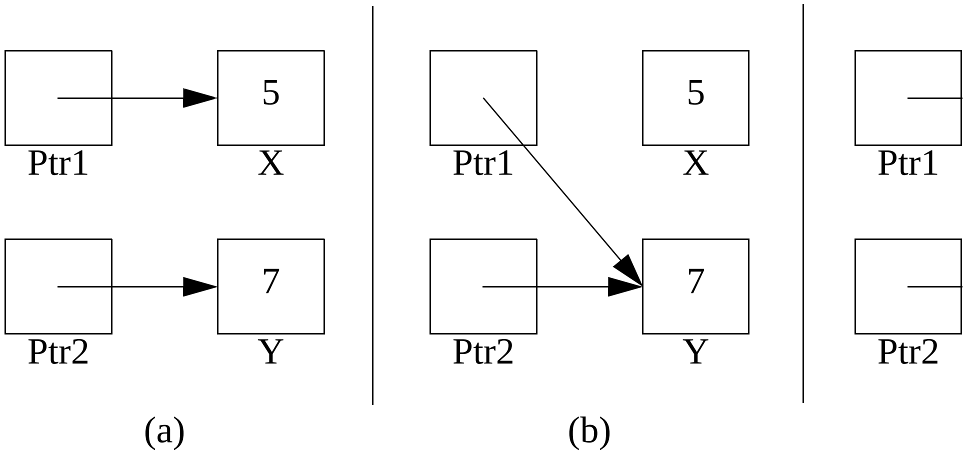
Pointer illustration



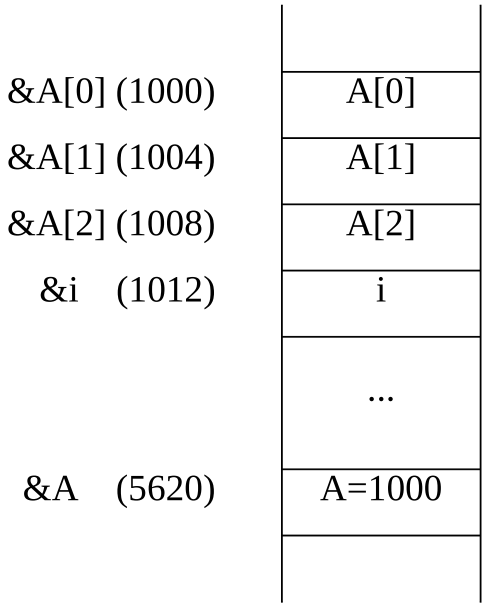
Result of `*Ptr=10`



Uninitialized pointer



(a) Initial state; (b) `Ptr1=Ptr2` starting from initial state;
(c) `*Ptr1=*Ptr2` starting from initial state



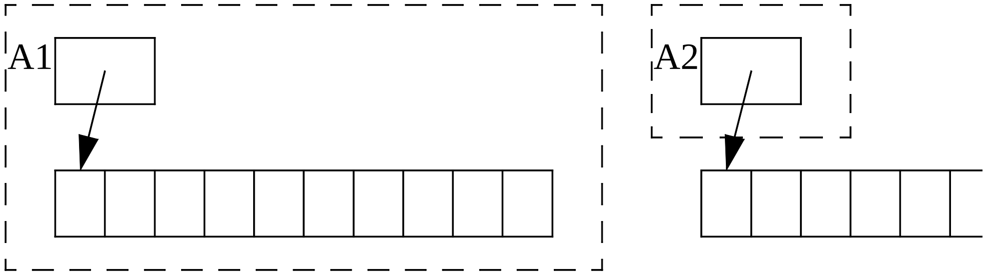
Memory model for arrays (assumes 4 byte `int`); declaration is `int A[3]; int i;`


```
size_t strlen( const char *Str );
char * strcpy(       char *Lhs, const char *Rhs );
char * strcat(       char *Lhs, const char *Rhs );
int    strcmp( const char *Lhs, const char *Rhs );
```

Some of the string routines in `<string.h>`

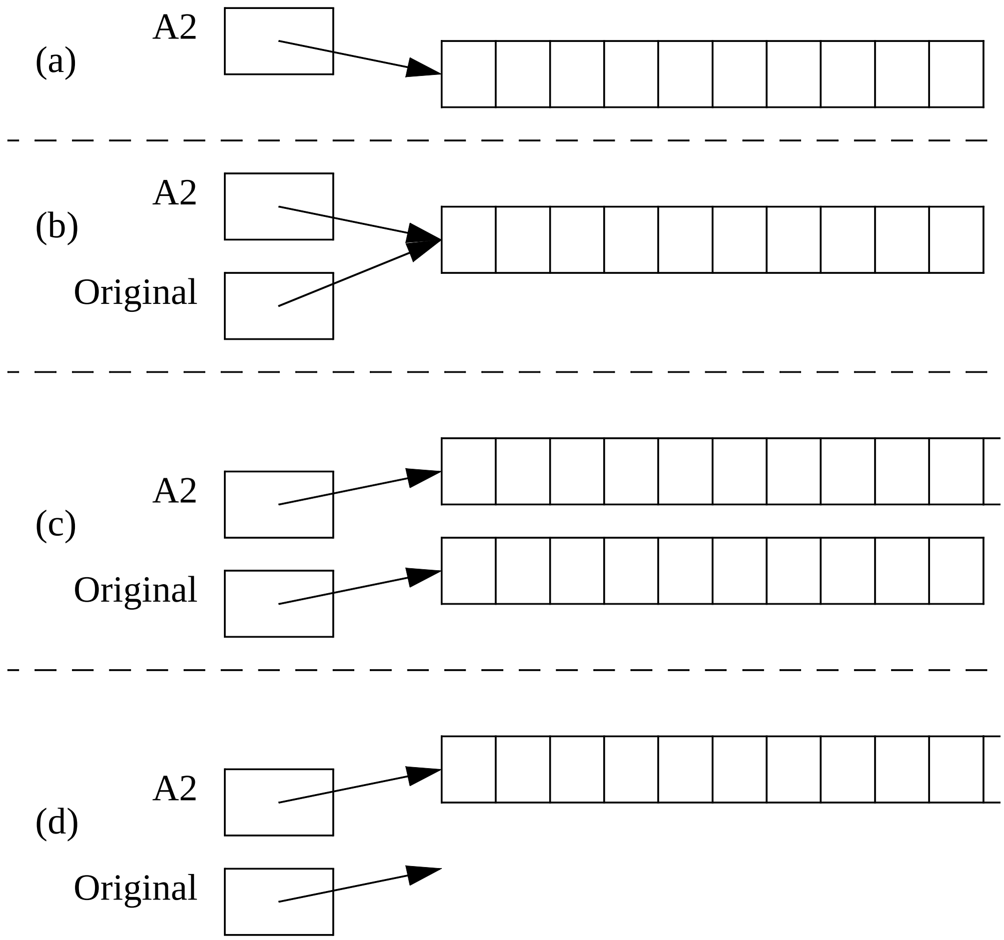

```
void
F( int i )
{
    int A1[ 10 ];
    int *A2 = new int [ 10 ];

    ...
    G( A1 );
    G( A2 );

    // On return, all memory associated with A1 is freed
    // On return, only the pointer A2 is freed;
    // 10 ints have leaked
    // delete [ ] A2;   // This would fix the leak
}
```

Two ways to allocate arrays; one leaks memory

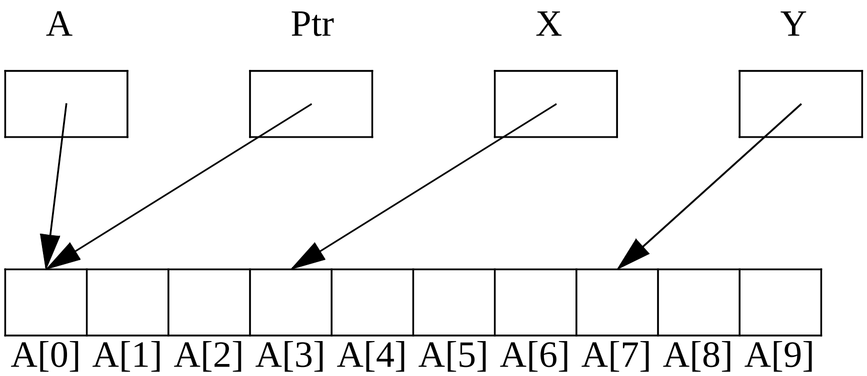



```
int *Original = A2;       // 1. Save pointer to the original
A2 = new int [ 12 ];      // 2. Have A2 point at more memory
for( int i = 0; i < 10; i++ ) // 3. Copy the old data over
    A2[ i ] = Original[ i ];
delete [ ] Original;      // 4. Recycle the original array
```

Memory reclamation



Array expansion: (a) starting point: `A2` points at 10 integers; (b) after step 1: `Original` points at the 10 integers; (c) after steps 2 and 3: `A2` points at 12 integers, the first 10 of which are copied from `Original`; (d) after step 4: the 10 integers are freed



Pointer arithmetic: `X=&A[3]; Y=X+4`


```
// Test that Strlen1 and Strlen2 give same answer
// Source file is ShowProf.cpp

#include <iostream.h>

main( )
{
    char Str[ 512 ];

    while( cin >> Str )
    {
        if( Strlen1( Str ) != Strlen2( Str ) )
            cerr << "Oops!!!!" << endl;
    }

    return 0;
}
```

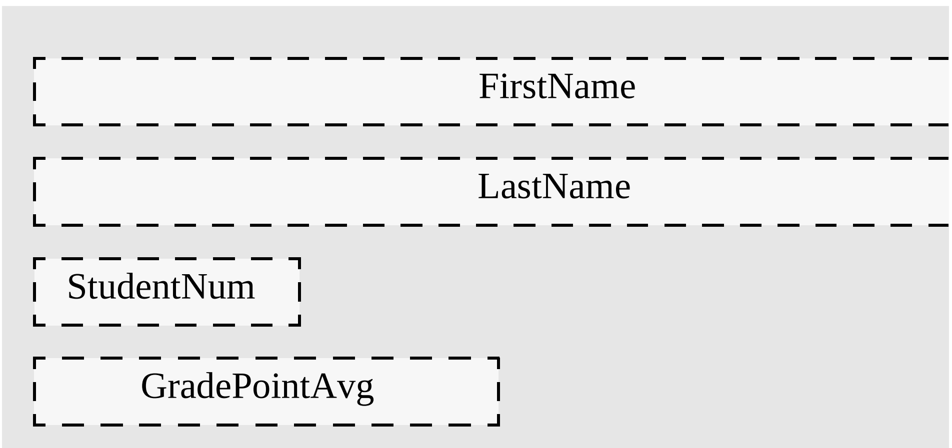
%time	cumsecs	#call	ms/call	name
26.6	0.34	25145	0.01	___rs__7istreamFPc
22.7	0.63	25144	0.01	_Strlen2__FPCc
14.8	0.82			mcount
12.5	0.98	25144	0.01	_Strlen1__FPCc
8.6	1.09	25145	0.00	_do_ipfx__7istreamFi
6.2	1.17	25145	0.00	_eatwhite__7istreamFv
4.7	1.23	204	0.29	_read
3.1	1.27	1	40.00	_main

First eight lines from `prof` for program

%time	cumsecs	#call	ms/call	name
34.4	0.31			mcount
26.7	0.55	25145	0.01	___rs__7istreamFPc
8.9	0.63	25145	0.00	_do_ipfx__7istreamFi
6.7	0.69	25144	0.00	_Strlen1__FPCc
6.7	0.75	25144	0.00	_Strlen2__FPCc
6.7	0.81	25145	0.00	_eatwhite__7istreamFv
6.7	0.87	204	0.29	_read
3.3	0.90	1	30.00	_main

First eight lines from `prof` with highest optimization

```
struct Student
{
    char FirstName[ 40 ];
    char LastName[ 40 ];
    int StudentNum;
    double GradePointAvg;
};
```



`Student` structure

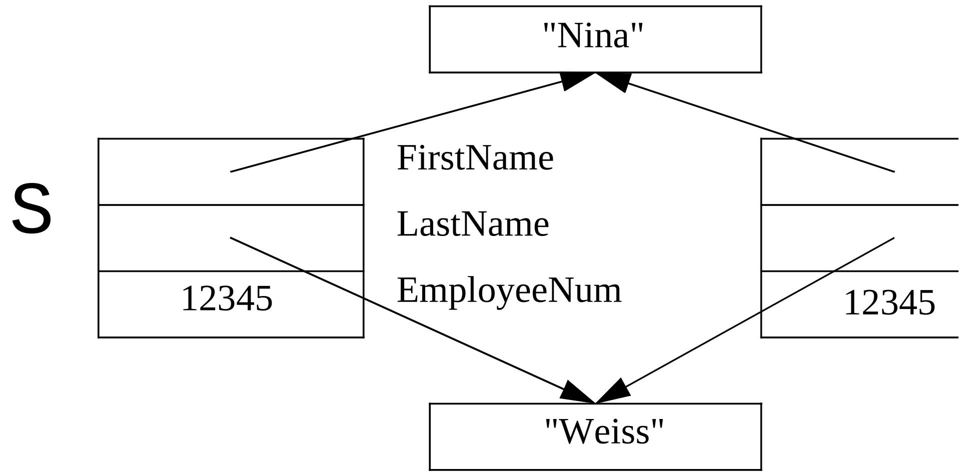


Illustration of a shallow copy in which only pointers are copied

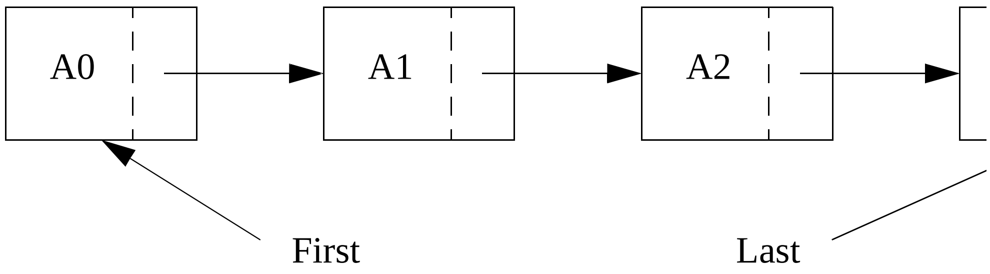


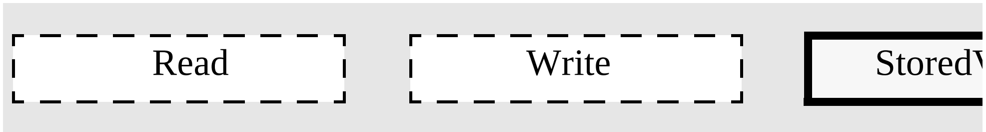
Illustration of a simple linked list

Chapter 2

Objects and Classes


```
// MemoryCell class
//  int Read( )         -->  Returns the stored value
//  void Write( int X ) -->  X is stored

class MemoryCell
{
  public:
        // Public member functions
    int Read( )           { return StoredValue; }
    void Write( int X )   { StoredValue = X; }
  private:
        // Private internal data representation
    int StoredValue;
};
```

A complete declaration of a `MemoryCell` class



`MemoryCell` members: `Read` and `Write` are accessible, but `StoredValue` is hidden

```
// Exercise the MemoryCell class

main( )
{
    MemoryCell M;

    M.Write( 5 );
    cout << "Cell contents are " << M.Read( ) << '\n';
        // The next line would be illegal if uncommented
//  cout << "Cell contents are " << M.StoredValue << '\n';
    return 0;
}
```

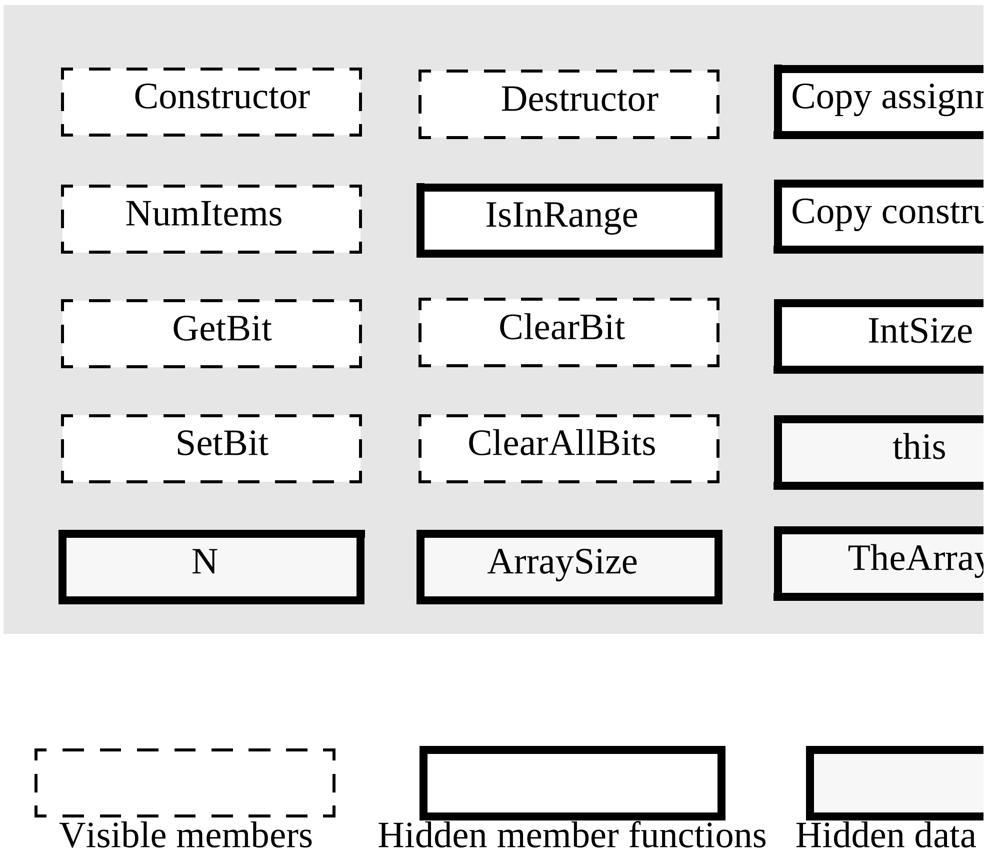
A simple test routine to show how `MemoryCell` objects are accessed

```
// MemoryCell interface
//  int Read( )         -->  Returns the stored value
//  void Write( int X ) -->  X is stored

class MemoryCell
{
  public:
    int Read( );
    void Write( int X );
  private:
    int StoredValue;
};



// Implementation of the MemoryCell class members

int
MemoryCell::Read( )
{
    return StoredValue;
}

void
MemoryCell::Write( int X )
{
    StoredValue = X;
}
```

A more typical `MemoryCell` declaration in which interface and implementation are separated


```
// BitArray class: support access to an array of bits
//
// CONSTRUCTION: with (a) no initializer or (b) an integer
//     that specifies the number of bits
// All copying of BitArray objects is DISALLOWED
//
// ******************PUBLIC OPERATIONS**********************
// void ClearAllBits( )   --> Set all bits to zero
// void SetBit( int i )   --> Turn bit i on
// void ClearBit( int i ) --> Turn bit i off
// int GetBit( int i )    --> Return status of bit i
// int NumItems( )        --> Return capacity of bit array

#include <iostream.h>

class BitArray
{
  public:
    // Constructor
    BitArray( int Size = 320 );          // Basic constructor

    // Destructor
    ~BitArray( ) { delete [ ] TheArray; }

    // Member Functions
    void ClearAllBits( );
    void SetBit( int i );
    void ClearBit( int i );
    int  GetBit( int i ) const;
    int  NumItems( ) const { return N; }
  private:
        // 3 data members
    int *TheArray;                       // The bit array
    int N;                               // Number of bits
    int ArraySize;                       // Size of the array

    enum { IntSz = sizeof( int ) * 8 };
    int IsInRange( int i ) const;// Check range with error msg

        // Disable operator= and copy constructor
    const BitArray & operator=( const BitArray & Rhs );
    BitArray( const BitArray & Rhs );
};
```

Interface for `BitArray` class



`BitArray` members

```
BitArray A;             // Call with Size = 320
BitArray B( 50 );       // Call with Size = 50
BitArray C = 50;        // Same as above
BitArray D[ 50 ];       // Calls 50 constructors, with Size 320
BitArray *E = new BitArray; // Allocates BitArray of Size 320
E = new BitArray( 20 );// Allocates BitArray of size 20; leaks
BitArray F = "wrong";  // Does not match basic constructor
BitArray G( );         // This is wrong!
```

Construction examples

Chapter 3

Templates

Array position	0	1	2	3	4	5
Initial State:	8	5	9	2	6	3
After A[0..1] is sorted:	5	8	9	2	6	3
After A[0..2] is sorted:	5	8	9	2	6	3
After A[0..3] is sorted:	2	5	8	9	6	3
After A[0..4] is sorted:	2	5	6	8	9	3
After A[0..5] is sorted:	2	3	5	6	8	9

Basic action of insertion sort (shaded part is sorted)

Array position	**0**	**1**	**2**	**3**	**4**	**5**
Initial State:	8	5				
After A[0..1] is sorted:	5	8	9			
After A[0..2] is sorted:	5	8	9	2		
After A[0..3] is sorted:	2	5	8	9	6	
After A[0..4] is sorted:	2	5	6	8	9	3
After A[0..5] is sorted:	2	3	5	6	8	9

Closer look at action of insertion sort (dark shading indicates sorted area; light shading is where new element was placed)


```
// Typical template interface
template <class Etype>
class ClassName
{
  public:
    // Public members
  private:
    // Private members
};


// Typical member implementation
template <class Etype>
ReturnType
ClassName<Etype>::MemberName( Parameter List  ) /* const */
{
    // Member body
}
```

Typical layout for template interface and member functions

Chapter 4

Inheritance

```
class Derived : public Base
{
    // Any members that are not listed are inherited unchanged
    // except for constructor, destructor,
    // copy constructor, and operator=
  public:
    // Constructors, and destructors if defaults are not good
    // Base members whose definitions are to change in Derived
    // Additional public member functions
  private:
    // Additional data members (generally private)
    // Additional private member functions
    // Base members that should be disabled in Derived
};
```

General layout of public inheritance

Public inheritance situation	**Public**	**Protected**	**Private**
Base class member function accessing *M*	Yes	Yes	Yes
Derived class member function accessing *M*	Yes	Yes	No
`main`, accessing *B.M*	Yes	No	No
`main`, accessing *D.M*	Yes	No	No
Derived class member function accessing	Yes	No	No
B is an object of the base class; *D* is an object of the publicly derived class; *M* is a member of the base class.			

Access rules that depend on what *M*'s visibility is in the base class

Public inheritance situation	Public	Protected	Private
F accessing *B.MB*	Yes	Yes	Yes
F accessing *D.MD*	Yes	No	No
F accessing *D.MB*	Yes	Yes	Yes
B is an object of the base class; *D* is an object of the publicly derived class; *MB* is a member of the base class. *MD* is a member of the derived class. *F* is a friend of the base class (but not the derived class)			

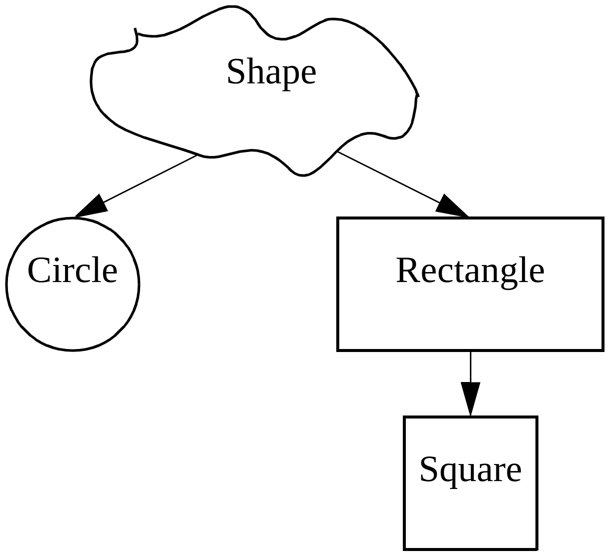
Friendship is not inherited

```
const VectorSize = 20;
Vector<int> V( VectorSize );
BoundedVector<int> BV( VectorSize, 2 * VectorSize - 1 );
    ...
BV[ VectorSize ] = V[ 0 ];
```

`Vector` and `BoundedVector` classes with calls to `operator[]` that are done automatically and correctly


```
Vector<int> *Vptr;
const int Size = 20;
cin >> Low;
if( Low )
    Vptr = new BoundedVector<int>( Low, Low + Size - 1 );
else
    Vptr = new Vector<int>( Size )

    ...
(*Vptr)[ Low ] = 0;       // What does this mean?
```

`Vector` and `BoundedVector` classes



The hierarchy of shapes used in an inheritance example

1. *Nonvirtual functions*: Overloading is resolved at compile time. To ensure consistency when pointers to objects are used, we generally use a nonvirtual function only when the function is invariant over the inheritance hierarchy (that is, when the function is never redefined). The exception to this rule is that constructors are always nonvirtual, as mentioned in Section 4.5.
2. *Virtual functions*: Overloading is resolved at run time. The base class provides a default implementation that may be overridden by the derived classes. Destructors should be virtual functions, as mentioned in Section 4.5.
3. *Pure virtual functions*: Overloading is resolved at run time. The base class provides no implementation. The absence of a default requires that the derived classes provide an implementation.

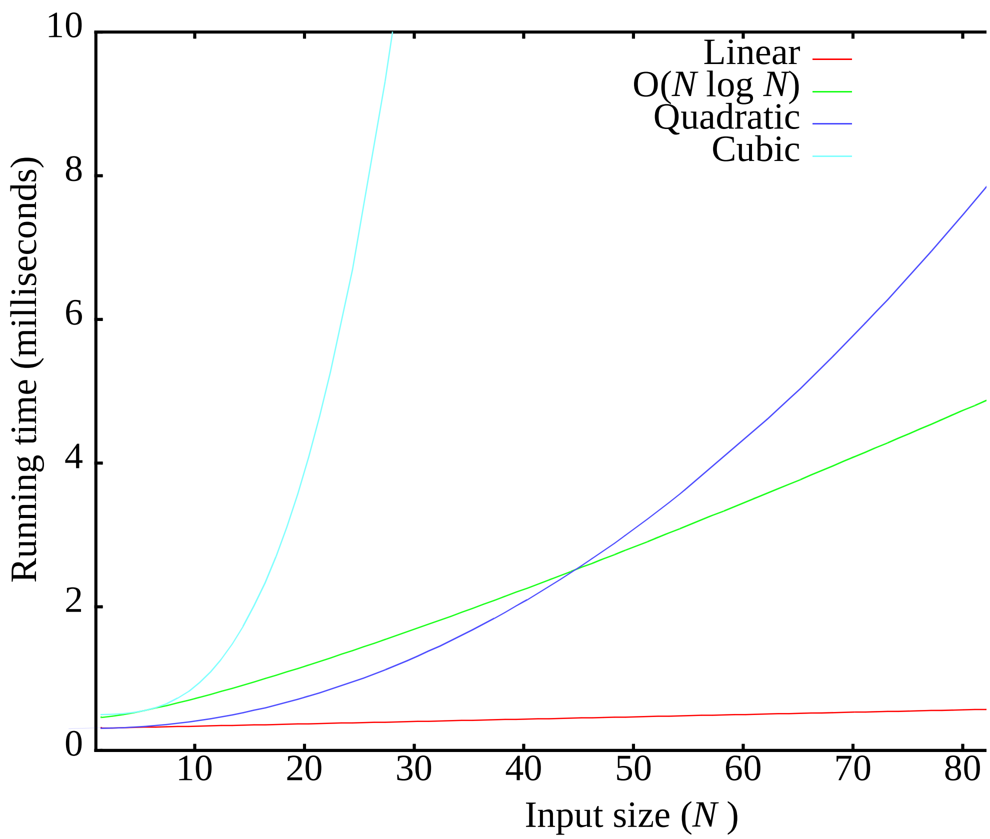
Summary of nonvirtual, virtual, and pure virtual functions

1. Provide a new constructor.
2. Examine each virtual function to decide if we are willing to accept its defaults; for each virtual function whose defaults we do not like, we must write a new definition.
3. Write a definition for each pure virtual function.
4. Write additional member functions if appropriate.

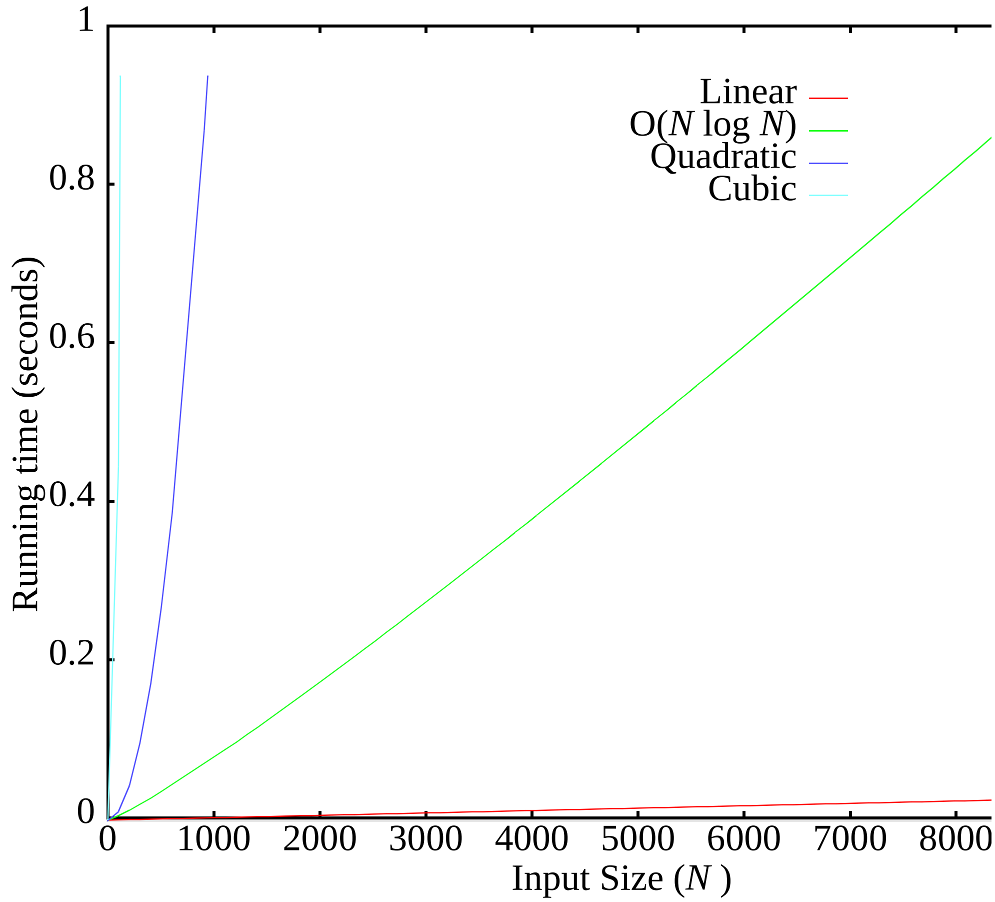
Programmer responsibilities for derived class

Chapter 5

Algorithm Analysis



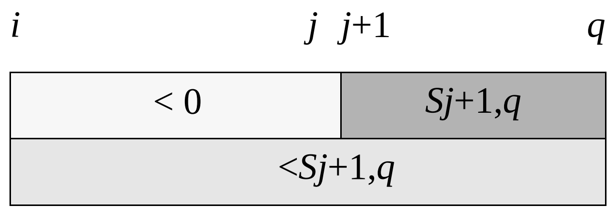
Running times for small inputs



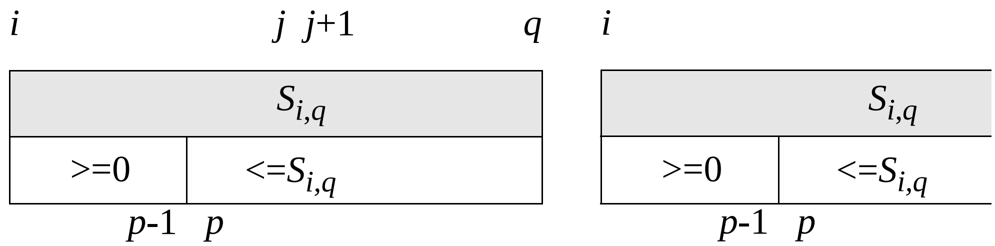
Running time for moderate inputs

Function	**Name**
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate



The subsequences used in Theorem 5.2



The subsequences used in Theorem 5.3. The sequence from *p* to *q* has sum at most that of the subsequence from *i* to *q*. On the left, the sequence from *i* to *q* is itself not the maximum (by Theorem 5.2). On the right, the sequence from *i* to *q* has already been seen.

DEFINITION: (Big-Oh) $T(\quad) = O(\qquad)$ if there are positive constants c and N_0 such that $T(\quad) \le cF(\quad)$ when $N \ge N_0$.

DEFINITION: (Big-Omega) $T(\quad) = \Omega(\qquad)$ if there are positive constants c and N_0 such that $T(\quad) \ge cF(\quad)$ when $N \ge N_0$.

DEFINITION: (Big-Theta) $T(\quad) = \Theta(\qquad)$ if and only if $T(\quad) = O(\qquad)$ and $T(\quad) = \Omega(\qquad)$.

DEFINITION: (Little-Oh) $T(\quad) = o(\qquad)$ if there are positive constants c and N_0 such that $T(\quad) < cF(\quad)$ when $N \ge N_0$.

Mathematical expression	**Relative rates of growth**
$T(N) = O(F(N))$	Growth of $T(N)$ is $\leq$ growth of $F(N)$
$T(N) = \Omega(F(N))$	Growth of $T(N)$ is $\geq$ growth of $F(N)$
$T(N) = \Theta(F(N))$	Growth of $T(N)$ is = growth of $F(N)$
$T(N) = o(F(N))$	Growth of $T(N)$ is $<$ growth of $F(N)$

Meanings of the various growth functions

N	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0.00103	0.00045	0.00066	0.00034
100	0.47015	0.01112	0.00486	0.00063
1,000	448.77	1.1233	0.05843	0.00333
10,000	NA	111.13	0.68631	0.03042
100,000	NA	NA	8.01130	0.29832

Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms

N	**CPU time T (milliseconds)**	T/N	T/N^2	$T/(N \log N)$
10,000	100	0.01000000	0.00000100	0.00075257
20,000	200	0.01000000	0.00000050	0.00069990
40,000	440	0.01100000	0.00000027	0.00071953
80,000	930	0.01162500	0.00000015	0.00071373
160,000	1960	0.01225000	0.00000008	0.00070860
320,000	4170	0.01303125	0.00000004	0.00071257
640,000	8770	0.01370313	0.00000002	0.00071046

Empirical running time for N binary searches in an N-item array

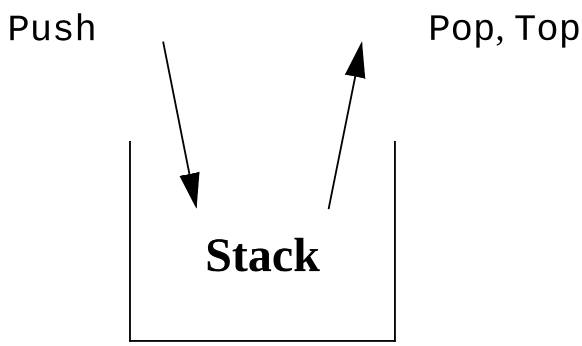
Chapter 6

Data Structures


```
#include <iostream.h>
#include "Stack.h"

// Simple test program for stacks

main( )
{
    Stack<int> S;

    for( int i = 0; i < 5; i++ )
        S.Push( i );

    cout << "Contents:";
    do
    {
        cout << ' ' << S.Top( );
        S.Pop( );
    } while( !S.IsEmpty( ) );
    cout << '\n';

    return 0;
}
```

Sample stack program; output is

```
Contents: 4 3 2 1 0
```

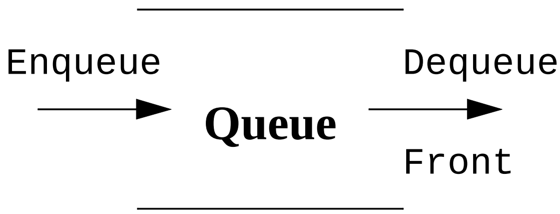


Stack model: input to a stack is by `Push`, output is by `Top`, deletion is by `Pop`


```cpp
#include <iostream.h>
#include "Queue.h"

// Simple test program for queues

main( )
{
    Queue<int> Q;

    for( int i = 0; i < 5; i++ )
        Q.Enqueue( i );

    cout << "Contents:";
    do
    {
        cout << ' ' << Q.Front( );
        Q.Dequeue( );
    } while( !Q.IsEmpty( ) );
    cout << '\n';

    return 0;
}
```

Sample queue program; output is

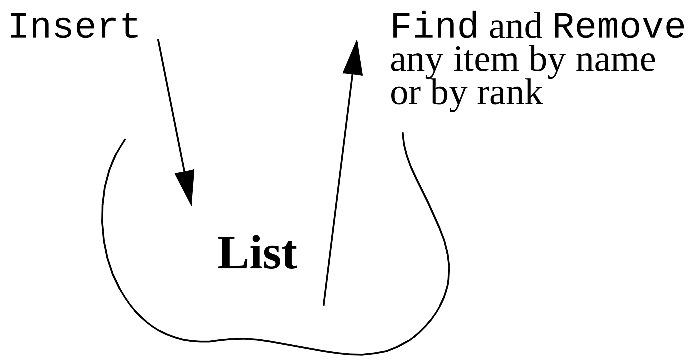
```
Contents:0 1 2 3 4
```



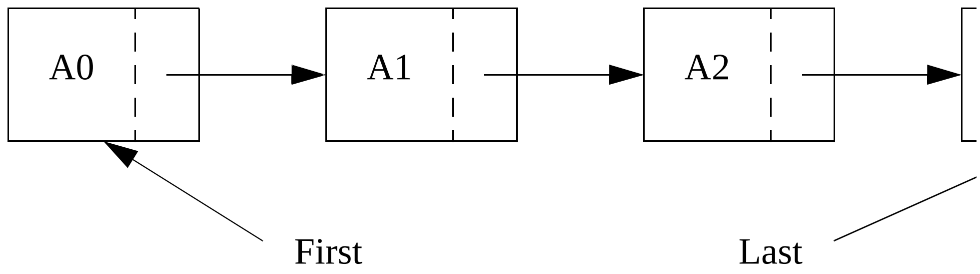
Queue model: input is by `Enqueue`, output is by `Front`, deletion is by `Dequeue`

```
#include <iostream.h>
#include "List.h"

// Simple test program for lists

main( )
{
    List<int> L;
    ListItr<int> P = L;

        // Repeatedly insert new items as first elements
    for( int i = 0; i < 5; i++ )
    {
        P.Insert( i );
        P.Zeroth( ); // Reset P to the start
    }

    cout << "Contents:";
    for( P.First( ); +P; ++P )
        cout << ' ' << P( );
    cout << "end\n";

    return 0;
}
```

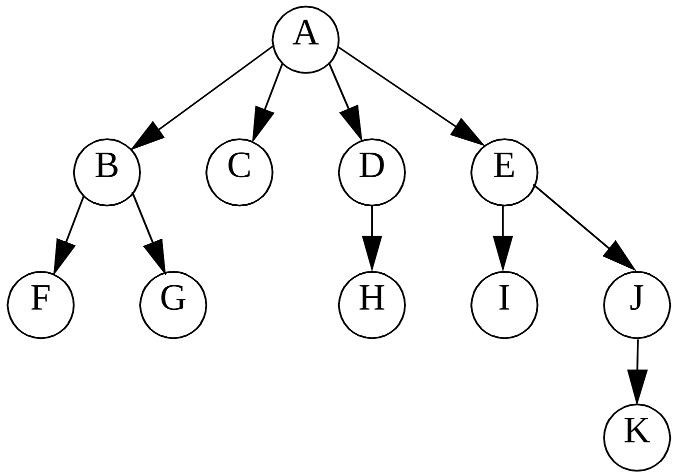
Sample list program; output is `Contents: 4 3 2 1 0 end`



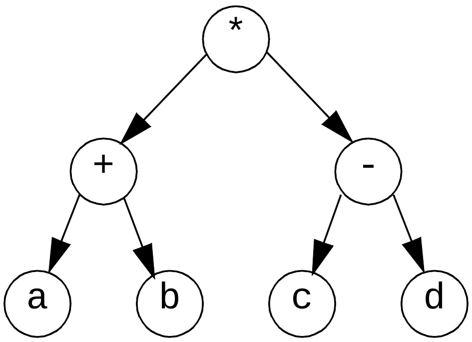
Link list model: inputs are arbitrary and ordered, any item may be output, and iteration is supported, but this data structure is not time-efficient



A simple linked list



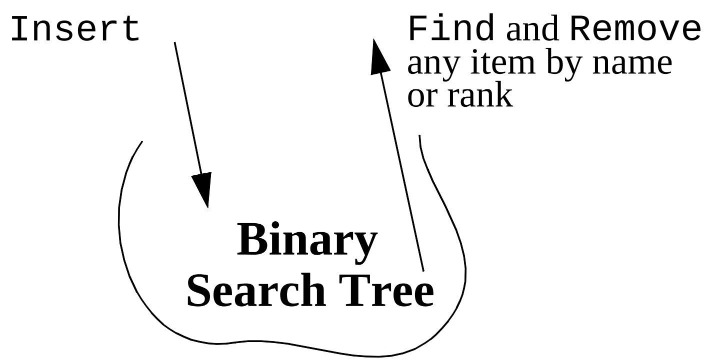
A tree



Expression tree for `(a+b)*(c-d)`


```
#include <iostream.h>
#include "Bst.h"

// Simple test program for binary search trees

main( )
{
    SearchTree<String> T;

    T.Insert( "Becky" );

        // Simple use of Find/WasFound
        // Appropriate if we need a copy
    String Result1 = T.Find( "Becky" );
    if( T.WasFound( ) )
        cout << "Found " << Result1 << ';';
    else
        cout << "Becky not found;";

        // More efficient use of Find/WasFound
        // Appropriate if we only need to examine
    const String & Result2 = T.Find( "Mark" );
    if( T.WasFound( ) )
        cout << " Found " << Result2 << ';';
    else
        cout << " Mark not found; ";

    cout << '\n';

    return 0;
}
```

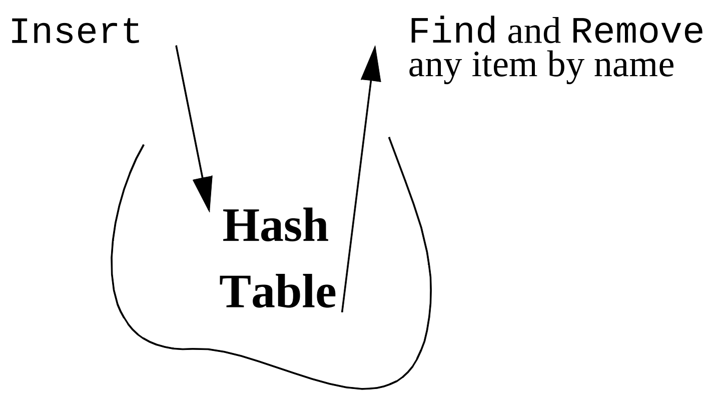
Sample search tree program;
output is `Found Becky; Mark not found;`



Binary search tree model; the binary search is extended to allow insertions and deletions


```
#include <iostream.h>
#include "Hash.h"

// A good hash function is given in Chapter 19
unsigned int Hash( const String & Element, int TableSize );

// Simple test program for hash tables

main( )
{
    HashTable<String> H;

    H.Insert( "Becky" );

    const String & Result2 = H.Find( "Mark" );
    if( H.WasFound( ) )
        cout << " Found " << Result2 << ';';
    else
        cout << " Mark not found; ";

    cout << '\n';

    return 0;
}
```

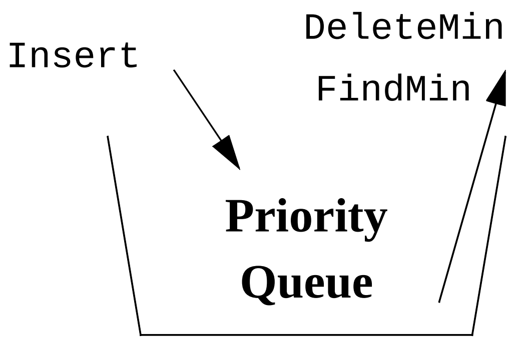
Sample hash table program;
output is `Found Becky; Mark not found;`



The hash table model: any named item can be accessed or deleted in essentially constant time


```
#include <iostream.h>
#include "BinaryHeap.h"

// Simple test program for priority queues

main( )
{
    BinaryHeap<int> PQ;

    PQ.Insert( 4 ); PQ.Insert( 2 ); PQ.Insert( 1 );
    PQ.Insert( 5 ); PQ.Insert( 0 );

    cout << "Contents:";
    do
    {
        cout << ' ' << PQ.FindMin( );
        PQ.DeleteMin( );
    } while( !PQ.IsEmpty( ) );
    cout << '\n';

    return 0;
}
```

Sample program for priority queues;
output is `Contents: 0 1 2 3 4`



Priority queue model: only the minimum element is accessible

Data Structure	Access	Comments
Stack	Most recent only, `Pop`, $O(1)$	Very very fast
Queue	Least recent only, `Dequeue`, $O(1)$	Very very fast
Linked list	Any item	$O(N)$
Search Tree	Any item by name or rank, $O(\log N)$	Average case, can be made worst case
Hash Table	Any named item, $O(1)$	Almost certain
Priority Queue	`FindMin`, $O(1)$, `DeleteMin`, $O(\log N)$	`Insert` is $O(1)$ on average $O(\log N)$ worst case

Summary of some data structures