POINTERS TO OBJECTS

Objects are stored in memory, so pointers can point to objects just as they can to variables of basic types.

The new *Operator*:

The new operator allocates memory of a specific size from the operating system and returns a pointer to its starting point.

If it is unable to find space, in returns a 0 pointer.

When you use new with objects, it does not only allocate memory for the object, it also *creates* the object in the sense of invoking the object's constructor.

This guarantees that the object is correctly initialized, which is vital for avoiding programming errors.

The delete *Operator*:

To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that releases the memory back to the operating system.

If you create an array with new Type[ ];, you need the brackets when you delete it:

```
int * ptr = new int[10];
:
delete [ ] ptr;
```

Don't forget the brackets when deleting arrays of objects.
Using them ensures that all the members of the array are deleted and that the destructor is called for each one.
If you forget the brackets, only the first element of the array will be deleted.

Example:

```
class String{
   int size;
   char *contents;
 public:
   String();                                    // Default constructor
   String(const char *);                        // Constructor
   String(const String &);                      // Copy constructor
   const String& operator=(const String &);     // Assignment operator
   void print() const ;
   ~String();                                   // Destructor
};
int main()
{
    String *sptr = new String[3];      // Creats 3 objects
    String s1("String_1");             // A String object
    String s2("String_2");             // Another String object
    *sptr = s1;                        // Assignment to the first elelement of the array
    *(sptr + 1) = s2;                  // Assignment to the second element of the array
    sptr->print();                     // Prints the first element
    (sptr+1)->print();                 // Prints the second element
    sptr[1].print();                   // Prints the second element
    delete[] sptr;                     // Objects pointed by sptr are deleted
    return 0;
}
```

See Example: e71.cpp

Linked List of Objects

A class may contain a pointer to objects of its type. This pointer can be used to build a chain of objects, a linked list.

```
class Teacher{
   friend class TeacherList;
   string name;
   int age, numOfStudents;
   Teacher * next;                    // Pointer to next object of teacher
 public:
   Teacher(const string &, int, int);  // Constructor
   void print() const;
   const string& getName() const {return name;}
  ~Teacher()                  // Destructor
};
```

See Example: e72.cpp

```
// linked list for teachers
class TeacherList{
   Teacher *head;
  public:
   TeacherList(){head=0;}
   bool append(const string &,int,int);
   bool del(const string &);
   void print() const ;
   ~TeacherList();
};
```


Problem and Solutions: In the previous example the Teacher class must have a pointer to the next object and the list class must be declared as a friend.

If this class is written by the same group then it is possible to put such a pointer in the class.

But usually programmers use ready classes, written by other groups, for example classes from libraries. And these classes may not have a next pointer.

To build linked lists of such ready classes there are two techniques.

a) Using the is-a relation:

Programmer can derive a new class (TeacherForList) with a next pointer.

```
class TeacherForList : public Teacher{      // TeacherForList is a Teacher with  a pointer
   friend class Teacher_list;
   TeacherForList * next;                        // next Teacher
   TeacherForList(const string &, int, int);     // constructor
};

TeacherForList :: TeacherForList (const string & n, int a, int nos)
                                    :Teacher (n, a, nos)
{
    next = 0;
}
```

b) Using the has-a relation:

Another way to build linked lists of such ready classes is to define a **node class** that **has a** teacher object.

The node objects are conneted to build the list.
Each object of the node class will hold the addresses of an element (Teacher)

```
class TeacherNode{                          // TeacherNode has a Teacher
   friend class TeacherList;
   Teacher * element;                       // The element of the list
   TeacherNode * next;                      // next node
   TeacherNode(const string &, int, int);   // constructor
   ~TeacherNode();                          // destructor
};

TeacherNode::TeacherNode(const string & n, int a, int nos){
   element = new Teacher(n, a, nos);
   next = 0;
}

TeacherNode::~TeacherNode(){
  delete element;
}
```

See Example: e73.cpp

Pointers and Inheritance

If a class Derived has a public base class Base, then a pointer to Derived can be assigned to a variable of type pointer to Base without use of explicit type conversion.

In other words, a pointer to Base can carry the address of an object of Derived.

A pointer to Base can point to objects of Derived.

For example, a pointer to Teacher can point to objects of Teacher and to objects of Principal.

A principal **is a** teacher, but a teacher is not always a principal.

The opposite conversion, for pointer to Base to pointer to Derived, must be explicit.

```
class Base{
.....
};
class Derived : public Base {
.....
};
Derived d;
Base *bp = &d;                  // implicit conversion
Derived *dp = bp;               // ERROR! Base is not Derived
dp = static_cast<Derived *>(bp); // explicit conversion
```

Accessing members of the Derived class via a pointer to the Base class:

When a pointer to Base class points to objects of the Derived class, only the members inherited from the Base can be accessed via this pointer.

In other words, members just defined in the Derived class, can not be accessed via a pointer to the Base class.

For example, a pointer to Teacher can hold the address of an object of the Principal type.

Using this pointer (Teacher type) it is possible to access only teacher properties of the principal, i.e. only the members that the Principal inherits from the Teacher class.

Using a pointer to the derived type (Principal) it is possible to access, as expected, all (public) members of the Principal (both inherited from the Teacher and defined in the Principal).

See the example in the next slide.

We will investigate some additional issues about pointers with inheritance (such as accessing overridden functions) in the next chapter (Polymorphism).

 Example:

```
class Teacher{                          // Base class
  protected:
    string name;
  public:
    void teachClass ();                 // Teachers  behavior (responsibility)
};
class Principal : public Teacher{       // Derived class
     string school_name;
   public:
      void directSchool ();             // Principals behavior (responsibility)
};
//---------------- Test Code --------------------
  Principal  objPrincipal;                      // Object of Principal type
  Teacher   *ptrTeacher = & objPrincipal;       // Pointer to Teacher points to Principal
  ptrTeacher->teachClass();                     // OK. Teaching is a teacher- behavior
  ptrTeacher->directSchool();                   // ERROR! Directing a school is not a
                                                // teacher- behavior. It is not inherited
                                                // from the Teacher
  Principal   *ptrPrincipal = & objPrincipal;   // Pointer to Principal points to Principal
  ptrPrincipal ->teachClass();                  // OK. Principal is a Teacher
  ptrPrincipal ->directSchool();                // OK. Principals behavior
```

Pointers and Inheritance (con'd):

If the class Base is a **private** base of Derived , then the implicit conversion of a Derived* to Base* would not be done.

Because, in this case a public member of Base can be accessed through a pointer to Base but not through a pointer to Derived.

Example:

```
class Base{
    int m1;
  public:
    int m2;                          // m2 is a public member of Base
};

class Derived : private Base {     // m2 is not a public member of Derived
  :
};

Derived d;
d.m2 = 5;                            // ERROR!  m2 is private member of Derived
Base *bp = &d;                       // ERROR! private base

bp = reinterpret_cast<Base*>(&d); // ok: explicit conversion AVOID!
bp->m2 = 5;                          // ok but dangerous  AVOID!
```

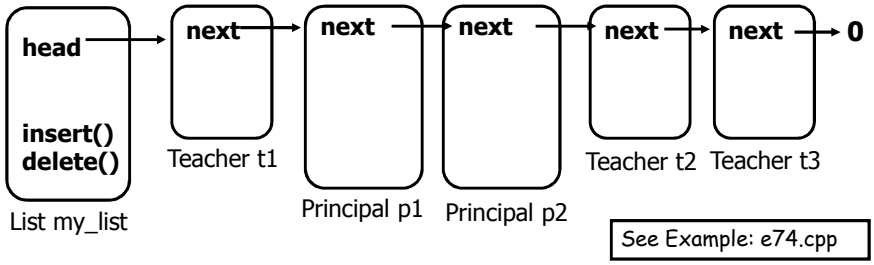

Heterogeneous Linked Lists

Using the inheritance and pointers, **heterogeneous** linked lists can be created.

A list specified in terms of pointers to a base class can hold objects of any class derived from this base class.

We will discuss heterogeneous lists again, after we have learnt **polymorphism**.

Example: A list of teachers and principals



See Example: e74.cpp

There is a **list** class in the Standard Template Library (STL) of the C++. You don't need to write classes to build linked lists. You can use the list class of the library.