Dynamic Memory Allocation

`new` and `delete` operators

- Memory is allocated by using the `new` operator (it is not a function)
  - `new` is followed by a type name and causes a block of memory of the size of the type to be allocated on the free store
  - It returns the address of the allocated block

    ```
    vec3 * v;
    v = new vec3;
    ```

  - The above replaces the C statement

    ```
    v = ( vec3 * ) malloc ( sizeof ( vec3 ) );
    ```

  - We can also allocate an array using `new` as

    ```
    v = new vec3[ n ];
    ```

  - If an array of a type-with-constructor is called, the constructor without arguments is called on all elements

    ```
    complex *az = new complex[ 10 ];    // calls complex() on all az[i]
    ```

  - The operator `new` makes calls to two separate functions:
    1. free store allocator
    2. appropriate constructor (additional feature to `malloc`)
- The space allocated by `new` is recycled by using the `delete` operator
  - The syntax is:

    ```
    delete v;
    delete[] az;   // Syntax to reclaim storage for an array
    ```

  - `delete` makes a call to a destructor function, if one exists; otherwise, it is exactly like the function `free` in C
- Example with the class `scalar_matrix`

```
class scalar_matrix
{
    double *sm;    // Matrix elements
    int col, row;  // Dimensions of matrix

    public:
        scalar_matrix( int, int );
        ~scalar_matrix() { delete[] sm; }
        double& element( int, int );
        // ...
}

scalar_matrix::scalar_matrix( int x, int y )      // Constructor function
{
    col = x;
    row = y;
    sm = new double[ col * row ];
}

double& scalar_matrix::element ( int x, int y )
```

```
{
    if ( 0 <= x && x < col && 0 <= y && y < row )
        return ( sm[y*row + x] );
}
```

- The declaration

```
scalar_matrix m ( 5, 6 );
```

causes a call to the constructor and initializes `m.sm` with a `new double[5 * 6]`

- Example – Linked list

```
class cell
{
    int info;
    cell * next;
    friend class list;
};

class list
{
    cell *head;
    cell *cur;       // Current position
    cell * pre;      // Predecessor of current position

    public:
        list();
        ~list();
        void insert ( int n );    // Insert before current position
        void remove();            // Remove current position
        int info();               // Information in the current element
        void advance();           // Advance current pointer
        void reset();             // Change current pointer to point to head
        bool atend() { return ( cur == NULL ); };
};

list::list()    // Constructor function
{
    head = cur = pre = NULL;
}

list::~list()   // Destructor function -- Removes all cells in the list
{
    cell *p = head;

    while ( p )
    {
        cell *q = p;
        p = p->next;
        delete q;
    }
}

void list::insert( int n )
```

```
{
    cell *p = new cell;
    p->info = n;
    p->next = cur;
    if ( pre )
        pre->next = p;
    else
        head = p;
    pre = p;
}

void list::remove()
{
    if ( !cur )
        return;
    if ( pre )
        pre->next = cur->next;
    else
        head = cur->next;
    cell *p = cur;
    cur = cur->next;
    delete p;
}

int list::info()
{
    return ( cur ? cur->info : 0 );
}

void list::advance();
{
    if ( cur )
    {
        pre = cur;
        cur = cur->next;
    }
}

void list::reset()
{
    cur = head;
    pre = NULL;
}
```

– `cell` has no public access but all member functions of `list` have been declared friends; you can walk through the list with the following loop:

```
for ( l.reset(); !l.atend(); l.advance() )
{
    // Body of the loop
}
```

Construction and destruction

- Constructor is a member function with the same name as the class and no return type

- Destructor is a member function with the name `~class-name`, no arguments, and no return type
  - Just as a constructor guarantees proper initialization of an object, the destructor guarantees proper cleanup after the object is no longer needed
- A class may have many constructors but not more than one destructor
- Both constructors and destructors are not actual functions but only get caused to be called
- If a class does not have a constructor, an instance of the class gets initialized with random data (garbage) or an exact copy of another object in the class
- Destructors are necessary only for those classes that need cleanup, most commonly recycling the allocated storage
- You may also decrement a reference counter, or close a file in the destructor
- A constructor for class `X` is called in the following circumstances:
  - A `static` local or a global variable of class `X` is declared. The constructor is called before `main` starts, or when the program enters its scope for the first time
  - An automatic variable of class `X` is declared within a block and the location of its declaration is reached
  - A function is called with argument `X`. A function call causes all its argument variables to be allocated and initialized
  - An unnamed temporary variable is created to hold the return value of a function returning `X`
  - An instance is obtained from the free store with `new X`
  - A variable is being initialized that has a member of type `X`
  - A variable is being initialized that is derived from `X`
- A destructor for class `X` is called in the following circumstances:
  - After the end of `main()` to destroy all `static` local or global instances of `X`
  - At the end of each block containing an automatic variable of type `X`
  - At the end of each function containing an argument of type `X`
  - To destroy any unnamed temporaries of type `X` after their use
  - When an instance of `X` is `delete`d
  - When a variable with a member of type `X` is destroyed
  - When a variable derived from `X` is destroyed
- Important to have destructor (or explicitly return the memory to the free store) as otherwise, the allocated memory stays after the end of a function and creates *garbage*
- A class may have many constructors with different argument types, with two of them being special:
  - The constructor with no arguments, or `X()`

    Also known as the default constructor

    Important: When an array is initialized using either of the following

    ```
    complex c[n];
    complex *c = new complex[n];
    ```

    the constructor with no arguments is called on each array element; if a class has constructors but none of them without arguments, it is an error to allocate an array
  - The *copy constructor*, `X( const X& )`
    - The copy constructor is called whenever a copy of an object needs to be made, such as the following instances

    · When a function argument is initialized with the value in the call
    · When a return value is copied out of the function into an unnamed temporary
    · When a variable is declared with an initializer of the same type

```
list l = tasklist;
```

      Initialization (above) is different from assignment (below)

```
list l;
l = tasklist;
```

      Initialization is accomplished by a call to the copy constructor (if it exists) which will be like

```
list::list ( const list& )
```

      Assignment is initialized with the constructor

```
list::list ( void )
```

      and `tasklist` is copied by using the operator `list::operator= ( list& )` (if that exists)

  * In the absence of a copy constructor, C++ makes a bitwise copy of all data members which may lead to problems with dynamically allocated objects (or object members)
  * Parameter to the copy constructor
    · Passed as a reference parameter, to prevent making a copy of the parameter
    · Qualified by a `const` to ensure that it does not get modified
- You can prevent a user from creating an object with no parameters by making the default constructor `private` to the class

- Static data
  - Consider the examples

```
complex i ( 0, 1 );       // global variable
fraction a[10];           // global array
```

  - The objects are allocated in the static data area
  - Both of them are initialized through the constructors before the start of `main`
  - If there is no destructor (`~complex` or `~fraction`), no cleanup occurs

- Local variables
  - Allocated in the stack area
  - Initialized and destroyed just as automatic variables

- Function arguments
  - Consider the function call

```
i = count ( tasklist, 0 );
```

  - When the function starts, two local variables (let us call them `l1` and `l2`) are allocated in the stack area
  - `l1` is initialized with a copy of `tasklist` (through copy constructor, if it exists, or through memberwise copy)
  - `l2` is initialized with 0
  - They are destroyed by the destructor calls at the end of the function

- Unnamed temporaries
  - Consider the following segment

```
fraction x, y;
// ...
cout << x * y;

fraction fraction::operator* ( fraction b )
{
    fraction r;
    // ...
    return ( r );
}
```

  - In the call to `operator* ( x, y )`, the result is computed in `r`
  - An unnamed temporary variable is created in the scope of the caller before the call is made to the function, and initialized with a copy of `r` using copy constructor or memberwise copy
  - The temporary variable is given to `operator<<` and destroyed after that
  - An explicit call to the constructor also results in unnamed temporary

```
cout << fraction ( 1, 2 );
```

- Free-store data
  - Consider the example

```
file* f = new file ( "input.dat" );
// ...
delete f;
```

  - The structure is allocated on free store
  - When the call to `delete` is made, the destructor is called first and then, the storage is returned to free store
  - When an array is deleted, the destructor is called on each element of the array before the storage space is returned

Copying dynamically allocated objects

- Default copy of one structure to another makes an exact copy of all members
- This behavior can lead to problems if one of the members is a pointer into the free store

Memberwise copying

- If no special copy semantics are specified through the copy constructors or assignment operator, a default method is applied to perform memberwise copy of structures
- Some structure members might be objects with special copy semantics
- Recursive copy rule:
  - If the object has a copy initializer or assignment operator, it is called
  - If the object is a built-in type or a pointer, a bitwise copy is made
  - Otherwise, memberwise assignment is applied recursively to each subobject

Reference counting

- Technique to avoid the problem of garbage in free store