Advanced memory handling

- Storage classes in C++
- The new and delete operators
- Overloading new and delete
- New and delete expressions
- Objects with restricted storage classes
- The RAII idiom

Storage classes in C++

- String literals are readonly objects
- Automatic (local) variables
- Global (namespace) static variables
- Local static variables
- Dynamic memory with new/delete or malloc/free
- Temporaries
- Arrays
- Subobjects (non-static class members)

Temporaries

- Created when evaluating an expression
- Guaranteed to live until the full expression is evaluated

```
void f( string &s1, string &s2, string &s3)
{
  const char *cs = (s1+s2).c_str();
  cout << cs;     // Bad!!

   if ( strlen(cs = (s2+s3).c_str()) < 8 && cs[0] == 'a' ) // Ok
        cout << cs;     // Bad!!
}
void f( string &s1, string &s2, string &s3)
{
    cout << s1 + s2;
    const string &s = s2 + s3; // binding to name keeps temporary
                               // alive until name goes out of scope
    if ( s.length() < 8 && s[0] == 'a' )
        cout << s;  // Ok
}
// s2+s3 destroyed here: when the const ref goes out of scope
```

New and delete

- New and delete expressions
- New and delete operators

```
void f( string &s1, string &s2, string &s3)
{
  try
  {
    int *ip = new int(42);  // new expression
    int *ap = new int[10];

    int *ptr = static_cast<int *>(::operator new(sizeof(int)));
  }
  catch(std::bad_alloc e) { ... }

  ::operator delete(ptr);

  delete ip;
  delete [] ap;
}
```

New expression

New expression do the following 3 steps when called as new X

- Allocate memory for X (usually calling operator new(sizeof(X)))
- Calls the constructor of X passing parameters if exists
- Converts pointer to the new object from void* to X* and returns

- But, what if
  - Operator new throws bad_alloc?
  - The constructor throws anything

```
X *ptr;
try
{
    ptr = new X(par1, par2);
}
catch( ... )
{
    // handle exceptions. Memory leak when constructor throws???
}
```

New and delete operators

- May throw std::bad_alloc exception
- Returns void*

```
namespace std
{
  class bad_alloc : public exception { /* ... */ };
}

void* operator new(size_t);     // new() may throw bad_alloc
void  operator delete(void *)  // delete() never throws
```

Nothrow version

- Returns nullptr if allocation is unsuccessful

```
// indicator for allocation that doesn't throw exceptions
struct nothrow_t {};
extern const nothrow_t nothrow;

// what to do, when error occurs on allocation
typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();

// nothrow version
void* operator new(size_t, const nothrow_t&);
void  operator delete(void*, const nothrow_t&);

void* operator new[](size_t, const nothrow_t&);
void  operator delete[](void*, const nothrow_t&);
```

Placement new

- Never allocate / deallocate memory

```
// placement new and delete
void* operator new(size_t, void* p)   { return p; }
void  operator delete(void* p, void*) { }

void* operator new[](size_t, void* p)   { return p; }
void  operator delete[](void* p, void*) { }

#include <new>
void f()
{
  char *cp = new char[sizeof(C)];
  for( long long i = 0; i < 10000000; ++i)
  {
     C *dp = new(cp) C(i);
     // ...
     dp->~C();
  }
  delete [] cp;
  return 0;
}
```

Overloading new and delete

- New and delete operators can be overloaded at two level
  - Class level (automatically static member functions)
  - Namespace level

```
struct node
{
          node( int v)  { val = v; left = rigth = 0; }
    void  print() const { cout << val << " "; }

    /* static */ void *operator new( size_t sz) throw (bad_alloc);
    /* static */ void  operator delete(void *p) throw();

    int   val;
    node *left;
    node *right;
};
```

Overloading new and delete

```
// member new and delete as static member
void *node::operator new( size_t sz) throw (bad_alloc)
{
    return ::operator new(sz);
}
void node::operator delete(void *p) throw()
{
    ::operator delete(p);
}
// global new and delete
void *operator new( size_t sz) throw (bad_alloc)
{
    return malloc(sz);
}
void operator delete( void *p) throw ()
{
    free(p);
}
```

Extra parameters for new

- One can define new and delete with extra parameters
  - Only new expression can pass extra parameters

```
struct node
{
          node( int v);
    void  print() const;

    static void *operator new( size_t sz, int i);
    static void  operator delete(void *p, int i);

    int   val;
    node *left;
    node *right;
};
void f()
{
    int i = 3;
    node *r = new(i+3) node(i);
}
```

Objects only in heap

- Sometimes restriction for storage location is useful

```
// Class should be allocated only in heap
class X
{
public:
    X() {}
    void destroy() const { delete this; }
protected:
    ~X() {}
};
class Y : public X { };
//  class Z { X xx; };  // use pointer!
void f()
{
    X* xp = new X;
    Y* yp = new Y;

    delete xp;        // syntax error
    xp->destroy();  // ok
}
```

Objects never in heap

- Sometimes restriction for storage location is useful

```
// Class should be allocated only not in heap
class X
{
private:
    static void *operator new( size_t);
    static void operator delete(void *);
    // Use: static void operator delete(void *) = delete; in C++11
};

class Y : public X { };
class Z { X xx; };  // ok!
void f()
{
    X* xp = new X;  // error
    X  x;           // ok
}
```

RAII

- Resource Acquisition Is Initialization
- The idea: keep a resource is expressed by object lifetime

```
// is this correct?
void f()
{
    char *cp = new char[1024];

    g(cp);
    h(cp);

    delete [] cp;
}
```

RAII

- Resource Acquisition Is Initialization
- The idea: keep a resource is expressed by object lifetime

```
// is this maintainable?
void f()
{
    char *cp = new char[1024];

    try
    {
      g(cp);
      h(cp);
      delete [] cp;
    }
    catch (...)
    {
      delete [] cp;
      throw;
    }
}
```

RAII

- Constructor allocates resource
- Destructor deallocates

```
// RAII
struct Res
{
    Res(int n) { cp = new char[n]; }
    ~Res() { delete [] cp; }
    char *getcp() const { return cp; }
};

void f()
{
    Res res(1024);

    g(res.getcp());
    h(res.getcp());
}
// resources will be free here
```

RAII

- Should be careful when implementing RAII
- Destructor calls only when living object goes out of scope
- Object lives only when constructor has successfully finished

```
// But be care:
struct BadRes
{
    Res(int n) { cp = new char[n]; ... init(); ... }
    ~Res()     { delete [] cp; }
    char *cp;
    void init()
    {
       ... throw XXX;
    }
};
```

Typical RAII solutions

- Smart pointers for memory handling
- Guards for locking
- ifstream, ofstream objects for file-i/o
- std::containers

```
class X
{
public:
    void *non_thread_safe();
private:
    Mutex lock_;
};

void *X::non_thread_safe();
{
    Guard<Mutex> guard(lock_);
    /* critical section */
}
```