C++ Memory management

- Storage types
- Allocation and deallocation
- *malloc()/free()* vs. *new/delete*
- Memory management internals

C++ Memory Management – Introduction

- C++ added necessary memory management support for object-oriented programming
- C++ fixed some loopholes and enhanced the memory management compared to plain C compilers

- This chapter shows:
  - The memory management model in C++
  - The three types of data storage
  - The various versions of operators (*new* and *delete)*
  - Some technique and guidelines for effective and bug-free storage usage

C++ Memory Management – Types of storage

- C++ has three fundamental types of storage:
  - Automatic storage
    - stack memory
  - Static storage
    - static data
  - Free storage
    - dynamically allocated data
- Each of them has different semantics of object initialization and lifetime

C++ Memory Management – Types of storage – Automatic storage

- The automatic storage (also called *stack storage)* is used for
  - Local objects that are not explicitly declared as *static* or *extern*
  - Local objects that are declared *auto* (default) or *register*
  - Function arguments
- *Created automatically* on the stack upon entering a function or a block
- *Destroyed automatically* when the function or the block exits
- At entrance, a *new copy* is always created
- **The default** value of automatic variables and non-class objects is **indeterminate**

C++ Memory Management – Types of storage – Static storage (1)

- The static storage is used for
  - Global objects
  - Static data members of a class
  - Namespace variables
  - Static variables in functions
- The address of a static object **remains the same** throughout the program's execution cycle
  - Constructed only once during the lifetime of the program
- **By default**, initialized to **zero**
  - Initialized by its constructor, if needed

C++ Memory Management – Types of storage – Static storage (2)

```
int num; // global variables have static storage

int func( void ){
  static int calls; // initialized to 0 by default
  return ++calls;
}

class C {
private:
  static bool b; // b has static storage
};

namespace NS{
  std::string str; // str has static storage
}
```

C++ Memory Management – Types of storage – Free storage

- The free storage (also called *heap memory* or *dynamic memory*) contains
  - Objects created by the *new* operator
  - Variables created by the *new* operator

- They persist until they are destroyed with the *delete* operator

  - Unreleased memory is not automatically returned to the operating system!
  - This produce ***memory leaks***

- The address of a *free store* object is determined **at runtime only**
- The initial value of raw storage that is allocated with the *new* operator is **unspecified**

C++ Memory Management – Allocation and Deallocation Functions (1)

- C++ offers the following global functions for allocating and deallocating
  - *new* and *new[]*
  - *delete* and *delete[]*

- These functions are accessible from the header <new>
  - The inclusion of this header is not necessary, done implicitly

- The declarations look like:

```
void* operator new( std::size_t ) throw( std::bad_alloc );
void* operator new[]( std::size_t ) throw( std::bad_alloc );
void operator delete( void* ) throw();
void operator delete[]( void* ) throw();
```

C++ Memory Management – Allocation and Deallocation Functions (2)

- The implicitly inclusion of *new/delete* does not implicitly include *std, std::bad_alloc* and *std::size_t*
  - The usage of these names needs an explicit inclusion of <new> header file

```
#include <new>
using namespace std;

char * allocate (size_t bytes);

int main
{
  char * buff = allocate( sizeof(char) );
  return 0;
}
```

C++ Memory Management – Semantics of Allocation and Deallocation (1)

- The allocation function returns an pointer to ***`void*`***

- The first argument of an allocation function is of type ***`std::size_t`***
  - This corresponds to the requested memory size

- The allocation tries to allocate the requested memory block from the *free store* memory

- If the allocation was successful then it returns a pointer to the start of the reserved block
  - Failure will be discussed later

```
void* operator new( std::size_t ) throw( std::bad_alloc );
```

C++ Memory Management – Semantics of Allocation and Deallocation (2)

- The deallocation function returns nothing (*`void`*)
- The first argument is of type ***`void*`***.
  - Additional arguments are possible
  - If the first argument is a pointer to *NULL*, then nothing is deallocated
  - The first argument must be a pointer that has been returned by an *new* operator
- Allocation and deallocation must be used in pairs

```
void operator delete( void* ) throw();
```

C++ Memory Management – *malloc() / free()* vs. *new / delete*

- C++ still supports *malloc()* and *free()* to ensure backward-compatibility
  - Combining legacy C code with C++ code
  - Write C++ code that is meant to be supported in C environment
  - Making *new* and *delete* implementable by unsing *malloc()* and *free()*

- Try to avoid using *malloc()* and *free()* in C++ code
  - They do not support object semantics

- *New* and *delete* are also **significantly safer**

C++ Memory Management – *malloc() / free()* vs. *new / delete* – Object Semantic Support

- *new* and *delete* automatically constructs respectively destructs objects
- *malloc()* and *free()* **only** allocate respectively deallocate memory space from the *heap*
  - The constructor and destructor **won't be invoked**

```
class C {...}

C* myNew( void ) {          // very bad
  C* pC = static_cast<C*>(malloc(sizeof(C)));
  return pC;
}

C* cPt;
cPt = myNew();
```

C++ Memory Management – *malloc() / free()* vs. *new / delete* – Safety

- The *new* operator calculates automatically the size of the object that it constructs
  - With *malloc()* the programmer has to give this size
  - *malloc()* returns a pointer to *void**
    - needs an explicit *type casting*

- The *new* operator bypasses these two problems

```
int* p = static_cast<int*> malloc(sizeof(int));
int* p2 = new int;
```

C++ Memory Management – *malloc() / free()* vs. *new / delete* – Extensibility

- The *new* operator can be overloaded
  - Special classes can thus implement its on *new* operator
  - This is not possible with *malloc()*

- Do not intermix *new* with *free()*, respectively *malloc()* with *delete*
  - **The result is undefined**
  - It is even possible, that *new* and *malloc()* use different heaps

C++ Memory Management – *new / delete* – Arrays (1)

- **`new []`** allocates an array of objects of a specific type
  - The returned value is a pointer to the first element of the array

```
int *p = new int[10];
bool equal = (p == &p[0]); // true
delete[] p;
```

- Object arrays must be delete with **`delete []`**
  - Using a plain *delete* ends in a unspecified behavior
  - *new []* stores the number of elements in the allocated array in a special way
  - The *delete []* retrives this number of elements and frees the correct number of elements
    - The correct number of destructors can be invoked
- Missmatch in *new []* and *delete []* causes memory leaks, heap corruptions or program crashes

C++ Memory Management – *new / delete* – Arrays (2)

- These rules apply also to arrays of fundamental types!
  - *delete []* doesn‘t invoke a destructor, but it still has to retrieve the number of elements in the array

```
void f()
{
  char* pc = new char[100];
  string* ps = new std::string[100];

  //...
  delete[] pc; // no destructors invoked

  delete[] ps  // each member’s destructor
               // is called
}
```

C++ Memory Management – Exceptions and *operator new* (1)

- Early C++ standards returned *NULL* if a *new* operation failed (e.g. not enough memory)
  - Same behaviour as *malloc()*
  - Applied also for *new []*
  - Check needed against *NULL*
    - Tedious and error prone
    - Long and time gourmand especially for arrays

```
char* p = new char [size];
if( p == NULL ) // this was fine until 1994
```

- Failures in dynamic memory allocation are quite rare
  - Indicate an unstable system

- The *NULL* returning in case of problem was replaced with a thrown exception ***std::bad_alloc***

C++ Memory Management – Exceptions and *operator new* (2)

- Programs calling *new* directly or indirectly (e.g. STL) must have an exception handler for *std::bad_alloc*
  - Otherwise, the program will terminate with an unhandled exception
    - Testing the returned pointer against *NULL* makes thus no sense at all, because the exception arrives before a possible test
    - The pointer check only uses system resources in case of normal behavior

```
void f(int size) {
  char* p = new char [size];
  // ...use p safely
  delete [] p;
  return;
}
```

C++ Memory Management – Exceptions and *operator new* (3)

- The usage of new with exception handling

```
#include <stdexcept>
#include <iostream>

using namespace std;
const int BUF_SIZE = 1048576L;
int main(){
  try{
    f(BUF_SIZE);
  }
  catch( bad_alloc& ex ){
    cout << ex.what() << endl;
    // ...other diagnostics and remedies
    return -1;
  }
  return 0;
}
```

C++ Memory Management – Exception-Free *operator new* (1)

- Sometimes it is undesirable to throw exception during the *new* operator
  - Exceptions are turned off for performance enhancement
  - Platform doesn't support exceptions
- Thus, the C++ standard has also a *new* operator, which **does not** throw an exception in case of failure
  - It returns a pointer to *NULL* in case of failure
  - This version of *new* operator takes an additional argument of type *const std::nothrow_t*
  - It exists for normal and for array *new* operators

C++ Memory Management – Exception-Free *operator new* (2)

- The usage will be

```
void f( int size ){
  char* p = new(nothrow) char [size];
  if( p == 0 )  // test against NULL
  {
    // ...use p
    delete [] p;
  }

  string* pstr = new(nothrow) string;
  if( pstr == 0 ) // test against NULL
  {
    // ...use pstr
    delete pstr;
  }
  return;
}
```

C++ Memory Management – Exception-Free *operator new* (3)

- The argument *nothrow* is defined and created in header <*new*>
  - `extern const nothrow_t nothrow;`

- The structur of *nothrow_t* looks like this
  - `struct nothrow_t {};`

- The empty structure won’t be used to transport any information about the failure

C++ Memory Management – Placement *operator new* (1)

- Another version of the *new* operator exist, which allows to construct an object at a predetermined memory position
  - Building custom-made memory pools
  - Garbage collection
  - Mission critical applications
    - The memory is already allocated
- Construction of object with the placement *new* operator is faster
  - The construction doesn’t need to allocate memory (only constructors are called)
  - **The memory must have been allocated in advance**

C++ Memory Management – Placement *operator new* (2)

- Destructors of objects constructed with the *placement new operator* has to be destructed **explicitly**
  - The `delete [] p` will delete only the char array but won't implicitly invoke the C++ classe's destructor

```
class C{
public:
  C() { cout<< "constructed" <<endl; };
  ~C(){ cout<< "destroyed" <<endl; };
};
int main(){
  char* p = new char [ sizeof(C) ];
  C* pc = new(p) C; // placement new
  //... used pc
  pc->C::~C();
  delete [] p;
}
```

C++ Memory Management – Exceptions during Object Construction (1)

- The *new operator* performs two operations:
  - Allocation of memory from the free storage
    - Calls an allocation function
  - Construct the object on the just allocated memory

- What happens if the object construction fails (exception during the construction)?
  - Does the allocated memory consist?

- The allocated memory will be directly freed before propagating the exception
  - No memory leak is produced

C++ Memory Management – Exceptions during Object Construction (2)

```
#include <new>
using namespace std;

class C{/*...*/};

void __new() throw (bad_alloc){
  C* p = reinterpret_cast<C*> (new char [sizeof(C)]); // step
   1: allocate raw memory
  try
  {
    new(p) C; // step 2: construct the objects on previously
   allocated buffer
  }
  catch(...) // catch any exception thrown from C's
   constructor
  {
    delete[] p; // free the allocated buffer
    throw; // re-throw the exception of C's constructor
  }
}
```

C++ Memory Management – Alignment Considerations

- Pointers returned by the *new operator* have the correct alignment to be converted into another pointer

```
char* pc = new char[ sizeof( Employee ) ];
Employee* pemp = new(pc) Employee;
//...use pemp
pemp->Employee::~Employee();
delete [] pc;
```

- This doesn’t work with buffers that are allocated on the stack (automatic storage)

```
char pbuff [ sizeof( Employee ) ];
Employee* p = new(pbuff ) Employee; // undefined
  behavior
```

- Neither it works on the previously *static* objects

C++ Memory Management – Alignment Considerations – Member Alignment

- The size of a *struct* or *class* might be bigger than the result of adding all internal data members
  - The compiler can add *padding* to realign members whose size does not fit exactly into a machine word

- To get the real size, use *sizeof()*

C++ Memory Management – Size of a Complete Object

- The size of a complete object can **never be** zero

```
class Empty{};
Empty e;          // e occupies at least 1 byte
```

- The object *e* occupies at least 1 byte
- The compiler does not allow objects with zero bytes
  - With zero bytes, addresses of different empty objects could overlap
  - The compiler guarantees that each empty object has also an unique address
- Incomplete objects can have a size of zero byte
  - Subobject in a derived class

C++ Memory Management – *new* and *delete* declared in a namespace

- The operators *new* and *delete* can be declared in a **class scope**
- It is **illegal** to declare them in a *namespace*

```
char* pc;

namespace A{
  void* operator new ( size_t );
  void operator delete ( void * );
  void func (){
    pc = new char ( ’a’);
  }
}

void f() { delete pc; } // A::delete or ::delete?
```

C++ Memory Management – Overloading *new* and *delete* in a class (1)

- Overloading the operators *new* and *delete* for a given class *C* is possible
  - The following statement would invoke the class' defined operators

```
C* p = new C;
//....
delete p;
```

- Class-specified versions of *new* and *delete* is useful when the default memory management is unsuitable
  - Different behavior in case of an allocation failure (see next slide)
  - Custom memory pool

C++ Memory Management – Overloading *new* and *delete* in a class (2)

```
class C{
  C()  { cout << "constructed" << endl; };
  ~C() { cout << "destroyed“ << endl; };
  void* operator new( size_t size ); // implicitly static
  void operator delete( void *p );   // implicitly static
};

void* C::operator new( size_t size ) throw( const char* ){
  void* p = malloc( size );
  if( p == 0 ) throw "allocation failure"; // instead of
                                           // std::bad_alloc
  return p;
}
void C::operator delete( void* p ){ free(p); }

int main(){
  try{ C* p = new C; delete p;}
  catch( const char* err ){ cout << err << endl; }
  return 0;
}
```

C++ Memory Management – Guidelines for Effective Memory Usage (1)

- Choosing the right type of memory storage is an important programmer's task
  - Impact of performance
  - Impact of security and reliability
  - Maintenance

- Prefer, if possible, **Automatic storage** to Free storage
  - Runtime overhead is little
    - Free storage needs interaction with the operating system
    - Free storage can be fragmented, thus finding a free spot can take some time
    - Exception handling also adds runtime overhead
  - Maintenance
    - Dynamic allocation might fail. Additional code must check this
  - Safety
    - Memory leaks can occur if dynamic objects are deleted more than one time or not deleted at all

C++ Memory Management – Guidelines for Effective Memory Usage (2)

```
void f( void )
{
  string* p = new string;
  // ...use p
  if( p->empty() != false )
  {
    // ...do something
    return; // OOPS! memory leak: p was not deleted
  }
  else // string is empty
  {
    delete p;
    // ..do other stuff
  }
  delete p; // OOPS! p is deleted twice if
            // isEmpty==false
}
```

C++ Memory Management – Guidelines for Effective Memory Usage (3)

```
void f( void )
{
  string s;
  // ...use s
  if( s.empty() != false )
  {
    // ...do something
    return;
  }
  else
  {
    // ..do other stuff
  }
}
```

C++ Memory Management – Local Object Instantiation

- The correct syntax for a local object instantiation by using the **default constructor** is:

```
string str;  // no parenthesis
```

- Attention, using parenthesis has a completly different meaning

```
string str(); // declaration of a function
```

C++ Memory Management – Zero as an Universal Initializer

- The literal 0 (*integer)* can be used as an universal initializer for any fundamental data type

```
void *p = 0; // zero is implicitly converted to void*
float salary = 0; // 0 is cast to a float
char name[10] = { 0 }; // 0 cast to a '\0'
bool b = 0; // 0 cast to false
void(*pf)(int) = 0; // pointer to a function
int (C::*pm)() = 0; // pointer to a class member
```

C++ Memory Management – Always initialize pointers

- An uninitialized pointer has an **undetermined value**
  - This is a *wild pointer*
  - It is quasi impossible to test if a *wild pointer* is valid
    - Especially when passed as an argument
    - Only *NULL* test is possible
- Do not trust to the *NULL-initialization* provided by the compiler!

```
void func( char* p );
int main(){
  char* p; // dangerous: uninitialized
  //...many lines of code; p left uninitialized
  if( p ) // erroneously assuming that a non-null
  {
    func( p ); // func has no way of knowing
  }
  return 0;
}
```

C++ Memory Management – Deleting a Pointer More than Once

- Applying an *delete* to an already deleted pointer has a undefined behavior
  - The pointer is undefined after the first delete

- Assigning a delete pointer to *NULL* avoids this danger
  - Applying the delete operator to a *NULL* pointer is harmless

```
if( ps->empty() )
{
  delete ps;
  ps = NULL; // safety-guard:
  // further deletions of ps will be harmless
}
```

C++ Memory Management – Storage reallocation

- In the C language, the function *realloc()* exists for resizing existing buffers
- This functionality **does not** exist in the C++ language

- Two possibilities exist in C++ to do something similar:
  - Allocating a new buffer with the new size, then copy the old buffer into the new one and finally delete the old buffer
    - Not very elegant and error prone
    - Inefficient and tedious
  - The better way for object / arrays that change their size often is to use containers of the STL
    - They change the size dynamically

C++ Memory Management – Local Static Variables (1)

- Local Static Variables (not to be confused with static class members) are initialized to *zero*
  - They are created before program's outset
  - They are destructed after program's termination

- They are accessible only within the declaration scope
  - Useful for function state storage

- Using classes with static data member is a better choice for keeping *states*
  - Better flexibility
  - Class member replace the static variables
  - Member functions replace the global functions

C++ Memory Management – Local Static Variables (2)

- Every derived object that inherits such a member function also refers to the same instance of the local static variables

- Static variables are problematic in a multithreaded environment
  - They are shared
  - They need to use locks to access them

C++ Memory Management – Conclusion (1)

- C++ offers tremendous diversity for dynamic memory allocation
- Three types memory storage exist (automatic, static, free)
- The *new* and *delete* operators have a lot of different versions
  - Exceptions, Exception-free, placement
- Garbage collector does not exist in C++
  - Additional runtime overhead
  - Destructors are not invoked immediately
- Use automatic memory allocation
- Use STL for object that grows and shrinks dynamically

C++ Memory Management – Conclusion (2)

- The source of most of the bugs in a C++ program must be searched in the

memory management

- Avoid these bugs by using if possible *automatic* memory allocation
- Avoid these bugs with a proper memory allocation and deallocation