Advanced Software Development

Benutzerdefinierte Speicherverwaltung

Übersicht

- Varianten von new
- Überladen von new/delete
- Memory Pools
- STL Allokatoren

Varianten von new

- Standard-Variante
  - falls nicht erfolgreich: Exception std::bad_alloc
- Zusätzliche Varianten (in Header <new>)
  - nothrow Variante (keine Exception)

```
size_t n = …;
int *p = new(nothrow) int[n];
if (p == nullptr) {
  cout << "Allocation failed\n";
}
…
```

  - „Placement new“

Placement new

- Objekt an vorgegebener Adresse erzeugen

```
char buf[sizeof(MyObject)];
Object *obj = new(buf) MyObject;
```

- Korrektes Alignment sicherstellen

```
aligned_storage<sizeof(MyObject), alignment_of<MyObject>::value>::type buf;
Object *obj = new(&buf) MyObject;
```

Placement new: Freigabe

- Getrennte Freigabe von Speicher und Objekt
- *delete* kann nicht zur Freigabe des Objekts verwendet werden
- Expliziter Aufruf des Destruktors

```
obj->~Object();
```

Placement new – Beispiel (1)

```
template <typename T>
class List {
public:
  ~List();

  void Prepend(T const &value);
  void RemoveFirst();

private:
  struct Node {
    T value;
    Node *next;
    Node(T const &val, Node *n = nullptr) : value(val), next(n) {}
  };
  Node *head = nullptr;
  std::vector<void*> nodeMem;  // reusable memory for nodes
};
```

Placement new – Beispiel (2)

```
template <typename T>
void List<T>::Prepend(T const &value) {
  if (nodeMem.empty()) {
    head = new Node(value, head);
  }
  else { // reuse memory
    void *mem = nodeMem.back();
    nodeMem.pop_back();
    head = new(mem) Node(value, head);
  }
}
```

Placement new – Beispiel (3)

```
template <typename T>
void List<T>::RemoveFirst() {
  Node *n = head;
  head = n->next;
  n->~Node();
  nodeMem.push_back(n);
}
```

Placement new – Beispiel (4)

```
template <typename T>
List<T>::~List() {
  while (head != nullptr) {
    Node *n = head;
    head = n->next;
    delete n;
  }
  for (auto mem : nodeMem) {
    operator delete(mem);
  }
}
```

Placement new – Beispiel (5)

- Laufzeitmessung
  - Testsystem: Xeon 2.5 GHz
  - List<int> mit insgesamt 10 Mio. Operationen
  - ohne Wiederverwendung: ca. 0,6s
  - mit Wiederverwendung: ca. 0,06s

Überladen von new/delete (1)

- Klasse kann Operatoren *new* und *delete* überladen
- Beispiel:

```
class Object {
public:
  …
  void *operator new(size_t size);
  void operator delete(void *ptr, size_t size);
};
```

- Gilt auch für alle Unterklassen

Überladen von new/delete (2)

- Auch Array-Operatoren *new[]* und *delete[]* können überladen werden
- Beispiel:

```
class Object {
public:
  …
  void *operator new[](size_t size);
  void operator delete[](void *ptr);
};
```

Überladen – Beispiel (1)

```
struct MemDeleter {
  void operator()(void *mem) { operator delete(mem); }
};

template <typename T>
class List {
public:
  ~List();

  void Prepend(T const &value);
  void RemoveFirst();
```

Überladen – Beispiel (2)

```
private:
  struct Node {
    T value;
    Node *next;
    Node(T const &val, Node *n = nullptr)
      : value(val), next(n) {}
    void *operator new(size_t size);
    void operator delete(void *ptr, size_t size);
  };
  Node *head = nullptr;
  static vector<unique_ptr<void, MemDeleter>> nodeMem;
};
```

Überladen – Beispiel (3)

```
template <typename T>
void List<T>::Prepend(T const &value) {
  head = new Node(value, head);
}

template <typename T>
void List<T>::RemoveFirst() {
  Node *n = head;
  head = n->next;
  delete n;
}

template <typename T>
List<T>::~List() {
  while (head != nullptr) {
    Node *n = head;
    head = n->next;
    delete n;
  }
}
```

Überladen – Beispiel (4)

```
template <typename T>
void *List<T>::Node::operator new(size_t size) {
  assert(size == sizeof(Node));
  if (nodeMem.empty()) {
    return ::operator new(size);
  }
  auto mem = nodeMem.back().release();
  nodeMem.pop_back();
  return mem;
}

template <typename T>
void List<T>::Node::operator delete(void *ptr, size_t size) {
  assert(size == sizeof(Node));
  nodeMem.emplace_back(ptr);
}
```

Memory Pool

- in Blöcke fixer Größe unterteilter Speicherbereich
- beschränkt auf Objekte fixer Größe
- sehr effiziente Reservierung und Freigabe eines Blocks
- effektive Ausnutzung des Speichers
- Freigabe des gesamten Speichers auf einmal möglich

Boost Memory Pool

- Fertig verwendbare Implementierung eines Memory Pools
- Verschiedene Varianten
  - pool (Objekte bestimmter Größe)
  - object_pool (Objekte eines bestimmten Typs)
  - singleton_pool (Globaler Pool für Objekte bestimmter Größe)

boost::pool

- Pool für Objekte einer bestimmten Größe
- Beispiel:

```
#include <boost/pool/pool.hpp>
…
void doSometing(int const n) {
  pool<> intPool(sizeof(int));
  for (int i = 1; i <= n; i++) {
    int *pi = reinterpret_cast<int*>(intPool.malloc());
    // oder: int *pi = new(intPool.malloc()) int;
    …
  }
} // automatische Freigabe aller allokierten Objekte
```

- Destruktoren werden nicht aufgerufen

boost::pool – Funktionalität

- Allokation eines Speicherblocks: *malloc*
- Freigabe eines Speicherblocks: *free*
- Freigabe aller Blöcke: *purge_memory*
- Allokation/Freigabe aufeinanderfolgender Blöcke:

```
int *arr = reinterpret_cast<int*>(intPool.ordered_malloc(n));
…
intPool.ordered_free(arr, n);
```

boost::object_pool

- Pool für Objekte eines bestimmten Typs
- Beispiel:

```
#include <boost/pool/object_pool.hpp>
…
void doSomething(int const n) {
  object_pool<Object> objPool;
  for (int i = 1; i <= n; i++) {
    Object *pObj = new(objPool.malloc()) Object;
    …
  }
} // automatische Freigabe aller allokierten Objekte
```

- Destruktoren werden aufgerufen
- Freigabe: *objPool.destroy(pObj);*

boost::object_pool – Beispiel (1)

```
template <typename T>
class List {
public:

  // Kein expliziter Destruktor nötig

  void Prepend(T const &value);
  void RemoveFirst();

private:
  struct Node {
    T value;
    Node *next;
    Node(T const &val, Node *n = nullptr) : value(val), next(n) {}
  };
  Node *head = nullptr;
  boost::object_pool<Node> nodePool;
};
```

boost::object_pool – Beispiel (2)

```
template <typename T>
void List<T>::Prepend(T const &value) {
  head = new(nodePool.malloc()) Node(value, head);
}

template <typename T>
void List<T>::RemoveFirst() {
  Node *n = head;
  head = n->next;
  nodePool.destroy(n);
}
```

Laufzeitmessung (10 Mio. Operationen):

- Wiederverwendung mit *vector*: 0,06 s
- Verwendung von *object_pool*: 0,03s

boost::singleton_pool

- Globaler Pool für Objekte einer fixen Größe (basierend auf boost::pool)
- Beispiel:

```
#include <boost/pool/singleton_pool.hpp>
…
struct MyPoolTag {};
typedef singleton_pool<MyPoolTag, sizeof(int)> MyIntPool;

int main() {
  int *pi = new(MyIntPool::malloc()) int;
  …
  MyIntPool::free(pi);
  …
```

boost::singleton_pool – Beispiel (1)

```
template <typename T>
class List {
public:
  ~List();
  void Prepend(T const &value);
  void RemoveFirst();

private:
  struct Node {
    T value;
    Node *next;
    Node(T const &val, Node *n = nullptr) : value(val), next(n) {}
    void *operator new(size_t size);
    void operator delete(void *ptr, size_t size);
  };
  typedef boost::singleton_pool<Node, sizeof(Node)> NodePool;
  Node *head = nullptr;
};
```

boost::singleton_pool – Beispiel (2)

```
template <typename T>
void List<T>::Prepend(T const &value) {
  head = new Node(value, head);
}

template <typename T>
void List<T>::RemoveFirst() {
  Node *n = head;
  head = n->next;
  delete n;
}

template <typename T>
List<T>::~List() {
  while (head != nullptr) {
    Node *n = head;
    head = n->next;
    delete n;
  }
}
```

boost::singleton_pool – Beispiel (3)

```
template <typename T>
void *List<T>::Node::operator new(size_t size) {
  assert(size == sizeof(Node));
  return NodePool::malloc();
}

template <typename T>
void List<T>::Node::operator delete(void *ptr, size_t size) {
  assert(size == sizeof(Node));
  NodePool::free(ptr);
}
```

Laufzeitvergleich

Listen-Implementierung	**10 Mio. Op.**
Standard new/delete	0,6s
Wiederverwendung mit vector	0,06s
Wiederverwendung mit statischem ptr_vector	0,1s
boost::object_pool	0,03s
boost::singleton_pool	
statischer boost::pool	

STL Allokatoren

- STL-Container fordern Speicher mittels sog. Allokatoren an
- Default-Allokator: std::allocator<T> im Header <memory>
- Angabe eines alternativen Allokators bei Deklaration

```
vector<int, MyAllocator<int>> v;
list<int, MyAllocator<int>> l;
map<string, int, less<string>, MyAllocator<pair<string const, int>> m;
```

std::allocator (1)

```
template <typename T> class allocator {
public:
  using size_type = size_t;
  using difference_type = ptrdiff_t;
  using pointer = T*;
  using const_pointer = const T*;
  using reference = T&;
  using const_reference = const T&;
  using value_type = T;

  template <typename U> struct rebind { using other = allocator<U>; };

  allocator();
  allocator(const allocator&);
  template <typename U> allocator(const allocator<U>&);

  ~allocator();
```

std::allocator (2)

```
  pointer address(reference x) const { return std::addressof(x); }
  const_pointer address(const_reference x) const { return std::addressof(x); }

  // request memory for n Ts, do not initialize
  pointer allocate(size_type n, allocator<void>::const_pointer hint = nullptr);

  // deallocate memory for n Ts, do not destroy
  void deallocate(pointer p, size_type n);

  size_type max_size() const;

  // initialize *p
  template <typename U, typename... Args>
  void construct(U *p, Args &&...args) { new(p) U(std::forward<Args>(args)...); }

  // destroy p (do not release memory)
  template <typename U> void destroy(U *p) { p->~U(); }
}
```

boost::pool_allocator

- STL-kompatibler Allokator basierend auf singleton_pool
- 2 Varianten
  - pool_allocator
  - fast_pool_allocator
- Beispiel:

```
#include <boost/pool/pool_alloc.hpp>
…
vector<int, pool_allocator<int>> vec;
list<int, fast_pool_allocator<int>> lst;
```

Laufzeitvergleich

10 Mio. push_back Operationen, Elementtyp int

Allokator	**vector**	**list**
std::allocator	0,11s	1,06s
boost::pool_allocator	0,42s	1,00s
boost::fast_pool_allocator	0,44s	0,72s