Basic C++

Dynamic array and List

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

http://gsd.web.elte.hu

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

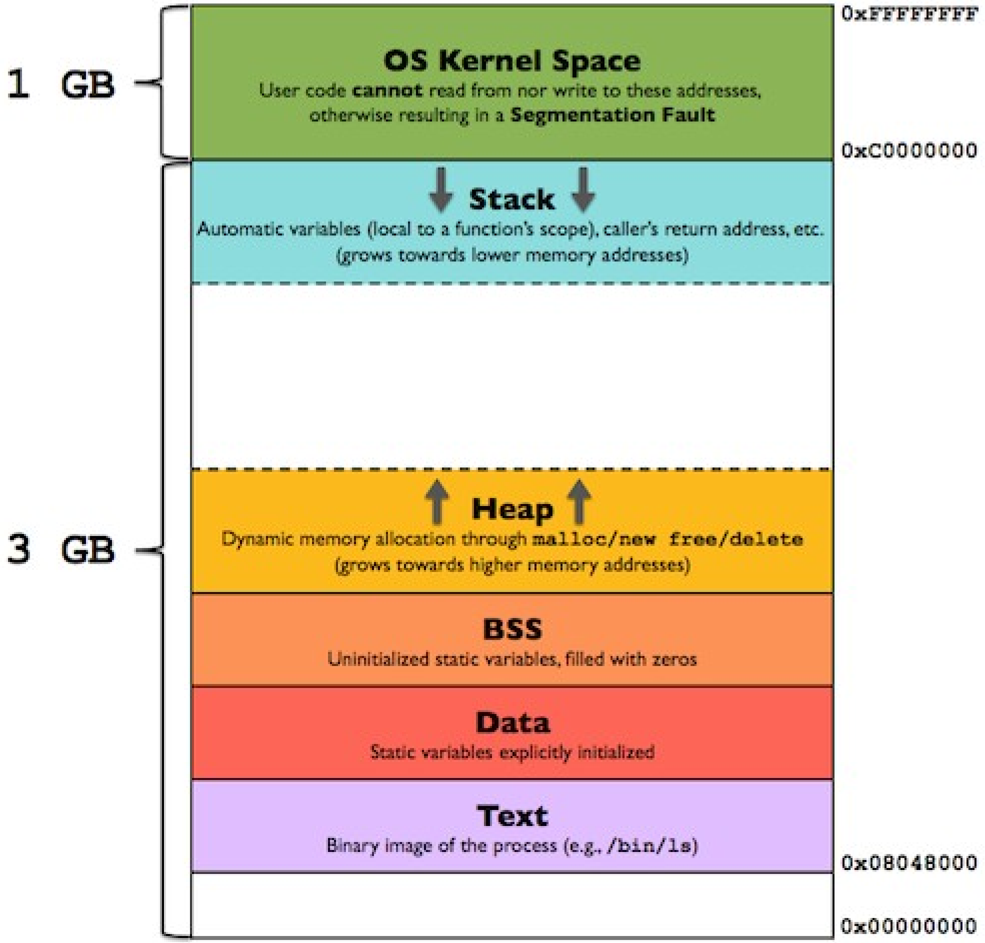
Advanced memory handling

- 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

```
namespace std
{
  using dangling_reference = string_view;
}
```

Memory model

- Different platform dependent memory models exist
- Most popular on UNIX: ELF (Executable and Linkable Format)



Storage classes in C++

- String literals are read-only objects
- Automatic (local) variables
- Global (namespace) variables with static lifetime
- 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;           // lifetime extension:
    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 "s" reference goes out of scope
```

Lifetime extension

- When a (const) reference is set to a temporary, the temporary will live until the reference goes out of scope

```
struct mystring : std::string {
  mystring(const std::string& s) : std::string(s) {};
  ~mystring() { std::cerr<<"lifetime end: "<< c_str() << '\n'; }
};
mystring operator+( const mystring& a,const mystring& b){
  std::string aa(a), bb(b);
  std::string res(aa+bb);
  return mystring(res);
}
int main() {
  {
    mystring first("first");
    mystring second("second");
    mystring third("third");
    mystring fourth("fourth");
    mystring fifth("fifth");

    const mystring& ref1 = first + second;
    static const mystring& ref2 = first + third;
    static const mystring& ref3 = std::max(fourth,fifth);
    std::cerr << "end of block" << '\n';
  }
  std::cerr << "end of main" << '\n';
}
```

Lifetime extension

- When a (const) reference is set to a temporary, the temporary will live until the reference goes out of scope

```
struct mystring : std::string {
  mystring(const std::string& s) : std::string(s) {};
  ~mystring() { std::cerr<<"lifetime end: "<< c_str() << '\n'; }
};
mystring operator+( const mystring& a,const mystring& b){
  std::string aa(a), bb(b);
  std::string res(aa+bb);
  return mystring(res);
}
int main() {
  {
    mystring first("first");
    mystring second("second");
    mystring third("third");
    mystring fourth("fourth");
    mystring fifth("fifth");

    const mystring& ref1 = first + second;                // ok
    static const mystring& ref2 = first + third;          // ok
    static const mystring& ref3 = std::max(fourth,fifth); // wrong!!!
    std::cerr << "end of block" << '\n';
  }
  std::cerr << "end of main" << '\n';
}
```

```
end of block
lifetime end: firstsecond
lifetime end: fifth
lifetime end: fourth
lifetime end: third
lifetime end: second
lifetime end: first
end of main
lifetime end: firstthird
```

Lifetime extension

- When a (const) reference is set to a temporary, the temporary will live until the reference goes out of scope

```
template <class T>
constexpr const T& max(const T& a, const T& b);
```

```
int main() {
  {
    mystring first("first");
    mystring second("second");
    mystring third("third");
    mystring fourth("fourth");
    mystring fifth("fifth");

    const mystring& ref1 = first + second;               // ok
    static const mystring& ref2 = first + third;         // ok
    static const mystring& ref3 = std::max(fourth,fifth); // wrong!!!
    std::cerr << "end of block" << '\n';
  }
  std::cerr << "end of main" << '\n';
}
```

end of block
lifetime end: firstsecond
lifetime end: fifth
lifetime end: fourth
lifetime end: third
lifetime end: second
lifetime end: first
end of main
lifetime end: firstthird

Lifetime extension

- When a (const) reference is set to a temporary, or member of a temporary the temporary will live until the reference goes out of scope

```
struct mystring : std::string {
  mystring(const std::string& s) : std::string(s) {};
  ~mystring() { std::cerr<<"lifetime end: "<< c_str() << ‘\n’; }
  int ifield;
};
mystring operator+( const mystring& a,const mystring& b){
  std::string aa(a), bb(b);
  std::string res(aa+bb);
  return mystring(res);
}
int main() {
  {
    mystring first("first");
    mystring fourth("fourth");
    mystring fifth("fifth");

    static const int &ref4 = (first + fourth).ifield;
    static const char *const &ref5 = (first + fifth).c_str();
    std::cerr << "end of block" << '\n';
  }
  std::cerr << "end of main" << '\n';
}
```

Lifetime extension

- When a (const) reference is set to a temporary, or member of a temporary the temporary will live until the reference goes out of scope

```
struct mystring : std::string {
  mystring(const std::string& s) : std::string(s) {};
  ~mystring() { std::cerr<<"lifetime end: "<< c_str() << ‘\n’; }
  int ifield;
};
mystring operator+( const mystring& a,const mystring& b){
  std::string aa(a), bb(b);
  std::string res(aa+bb);
  return mystring(res);
}
int main() {
  {
    mystring first("first");
    mystring fourth("fourth");
    mystring fifth("fifth");

    static const int &ref4 = (first + fourth).ifield;          // ok
    static const char *const &ref5 = (first + fifth).c_str(); // wrong!!!
    std::cerr << "end of block" << '\n';
  }
  std::cerr << "end of main" << '\n';
}
```

end of block
lifetime end: firstfifth
lifetime end: fifth
lifetime end: fourth
lifetime end: first
end of main
lifetime end: firstfourth

New and delete

- New and delete expressions
- New and delete operators

```
void f()
{
  try
  {
    int *ip1 = new int;     // new expression (uninitialized)
    int *ip2 = new int(42); // new expression (initialized)
    int *ap1 = new int[10]; // array new expression (uninitialized)
    int *ap2 = new int[]{1,2,3}; // array new expression
    auto ip3 = new auto(1); // creates int *ip3 and one int in heap

    // new operator
    int *ptr = reinterpret_cast<int *>(::operator new(sizeof(int)));

    ::operator delete(ptr);  // delete operator
    delete ip;               // delete expression
    delete [] ap;           // array delete expression
  }
  catch(std::bad_alloc e) { ... }
}
```

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 something?

```
X *ptr;

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

New and delete operators

- New expr. guarantees no leak if the constructor throws exception,

```
X *ptr;
try
{
      ptr = new     X(par1, par2); // allocate and run the constructor
}
catch( ... )
{
    // handle exceptions. No memory leak
}
```

New and delete operators

- New expr. guarantees no leak if the constructor throws exception

```
X *ptr;
try
{
   void *vp = operator new( sizeof(X) );  // allocate memory
   try {
      ptr = new(vp) X(par1, par2);  // run the constructor @vp
   }
   catch( ... ) {                   // catch all exceptions
      operator delete(vp);          // delete memory
      throw;                        // rethrow exxception
   }
   ptr = reinterpret_cast<X*>(vp);  // set pointer
}
catch( ... )
{
    // handle exceptions. No memory leak
}
```

New and delete operators

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

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

void* operator new(size_t);    // operator new() may throw bad_alloc
void  operator delete(void *)  // operator delete() never 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);    // operator new() may throw bad_alloc
void  operator delete(void *) noexcept;  // delete() since C++11
```

Nothrow version

- In some environments we want to avoid exceptions
- 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&) noexcept;
void  operator delete(void*, const nothrow_t&) noexcept;

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

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*) noexcept { }

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

#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();  // explicit call of destructor
  }
  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) noexcept;

    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) noexcept
{
    ::operator delete(p);
}
// global new and delete
void *operator new( size_t sz) throw (bad_alloc)
{
    return std::malloc(sz);
}
void operator delete( void *p) noexcept
{
    std::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) noexcept;

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

Alignment

```
#include <cassert> // example from https://www.bfilipek.com/2019/08/newnew-align.html
#include <cstdint>
#include <iostream>
#include <malloc.h>
#include <new>

class alignas(32) Vec3d {
    double x, y, z;
};

int main() {
    std::cout << "sizeof(Vec3d) is " << sizeof(Vec3d) << '\n';
    std::cout << "alignof(Vec3d) is " << alignof(Vec3d) << '\n';

    auto Vec = Vec3d{};
    auto pVec = new Vec3d[10];

    if(reinterpret_cast<uintptr_t>(&Vec) % alignof(Vec3d) == 0)
        std::cout << "Vec is aligned to alignof(Vec3d)!\n";
    else
        std::cout << "Vec is not aligned to alignof(Vec3d)!\n";

    if(reinterpret_cast<uintptr_t>(pVec) % alignof(Vec3d) == 0)
        std::cout << "pVec is aligned to alignof(Vec3d)!\n";
    else
        std::cout << "pVec is not aligned to alignof(Vec3d)!\n";

    delete[] pVec;
}
```

Alignment

```
// Before C++17

sizeof(Vec3d) is 32
alignof(Vec3d) is 32
Vec is aligned to alignof(Vec3d)!
pVec is not aligned to alignof(Vec3d)!
```

Alignment

```
// Before C++17

sizeof(Vec3d) is 32
alignof(Vec3d) is 32
Vec is aligned to alignof(Vec3d)!
pVec is not aligned to alignof(Vec3d)!


// Since C++17

sizeof(Vec3d) is 32
alignof(Vec3d) is 32
Vec is aligned to alignof(Vec3d)!
pVec is aligned to alignof(Vec3d)!
```

Alignment

```
// Before C++17

sizeof(Vec3d) is 32
alignof(Vec3d) is 32
Vec is aligned to alignof(Vec3d)!
pVec is not aligned to alignof(Vec3d)!


// Since C++17

sizeof(Vec3d) is 32
alignof(Vec3d) is 32
Vec is aligned to alignof(Vec3d)!
pVec is aligned to alignof(Vec3d)!


#include <cstdlib>

void *malloc( std::size_t size);  // align to std::max_align_t
// Since C++17 over-aligning:
void *alligned_alloc( std::size_t allignment, std::size_t size);
```

Operator new in C++17

- 14 global new() operator functions
- 8 class specific new() operators
- Corresponding delete() operators
- Overloading on std::align_val_t

```
enum class align_val_t : std::size_t {};

void *operator new( std::size_t sz, std::align_val_t al);
void  operator delete( void *ptr, std::align_val_t al) noexcept;

void f();
{
  auto ip   = new int{};   //   uses usual default alignment == 16
  auto pVec = new Vec3d{}; // uses class specified alignment == 32
  auto p64i = new(std::align_val_t{64}) int{};  // alignment == 64

  delete ip;
  delete pVec;
  delete p64i;
}
```

Destroying delete

- Since C++20

```
void operator delete(void* p, std::destroyig_delete_t);
void operator delete(void* p, std::destroyig_delete_t, std::align_val_t  al);
void operator delete(void* p, std::destroyig_delete_t, std::size_t sz);
void operator delete(void* p, std::destroyig_delete_t, std::size_t sz, std::align_val_t al);

struct B {
  virtual ~B();  // if destructor is virtual, delete operator looked up dynamically
  void operator delete(void*, std::size_t);
};
struct D : B {
  void operator delete(void*);   // using this calls the destructor
};
struct E : B {
  void log_deletion();
  void operator delete(E *p, std::destroying_delete_t) {  // not to call destructor
    p->log_deletion();
    p->~E();               // calling the destructor manually
    ::operator delete(p);
  }
};
void f() {
  B* bp = new D;
  delete bp; // 1: uses D::operator delete(void*)
  bp = new E;
  delete bp; // 2: uses E::operator delete(E*,std::destroying_delete_t) due to virtual destr
}
```

Objects only in heap

- Sometimes restriction for storage location is useful

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

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

Objects never in heap

- Sometimes restriction for storage location is useful

```
class X  // Class should be allocated only outside of heap
{
private:
 static void *operator new( size_t);
 static void operator delete(void *) noexcept;
 // static void operator delete(void *) noexcept = delete; in C++11
};

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

Allocators

- Role of allocators
- Allocators in C++03
- Allocators in C++11
- Polymorphic memory resource

Allocators

- A basic purpose of an allocator is to provide a source of memory for a given type, and a place to return that memory to once it is no longer needed (Bjarne Stroustrup)
- A service that grants exclusive use of a region of memory to a client (Alisdair Meredith)

```
template <typename T, typename Allocator = std::allocator<T>>
class vector;
```

Allocators

- A basic purpose of an allocator is to provide a source of memory for a given type, and a place to return that memory to once it is no longer needed (Bjarne Stroustrup)
- A service that grants exclusive use of a region of memory to a client (Alisdair Meredith)

```
    template <typename T, typename Allocator = std::allocator<T>>
    class vector;

namespace pmr
{
    template <typename T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

Allocators

- Invented by Alexander Stepanov
  - Make containers not to depend the hardware memory model
  - Hide the “huge”, “far” and “near” pointers
  - Part of the original STL
- More granular and than new and delete
  - Separate allocation from construction
  - Separate destruction from deallocation
- Containers may have special memory requirements

Why to write allocators?

- Performace reasons
  - Stack-based
  - Per-thread
  - Pool/Arena
- Relocatable objects
  - Shared memory
  - Persistable tree/list structures
- Control the allocation/deallocation process
  - Instrumenting
  - Debugging

C++98 allocators

```
template <class T, class Alloc = allocator<T> >
class Container
{
    // containers
    typedef T*                  pointer;
    typedef T const*            const_pointer;
    typedef T&                  reference;
    typedef T const&            const_reference;
    typedef void*               void_pointer;
    typedef void const*         const_void_pointer;
    typedef size_t              size_type;
    typedef ptrdiff_t           difference_type;
    // ...
};

// containers allocate memory by allocator’s allocate
T* ptr = alloc.allocate(size);

Allocator a, b;

assert (a == b);        // allocator were stateless

b.deallocate(a.allocate(size));   // well-defined
```

C++03 default allocator

```
template <class T>
struc allocator
{
    typedef size_t         size_type;
    typedef ptrdiff_t      difference_type;
    typedef T*             pointer;
    typedef T const*       const_pointer;
    typedef T&             reference;
    typedef T const&       const_reference;
    typedef void*          void_pointer;
    typedef void const*    const_void_pointer;

    template <class S> struct rebind { typedef allocator<U> other; };

    pointer  allocate( size_type n, allocator<void>::const_pointer hint = 0);
    void     deallocate( pointer p, size_type n);

    void construct( pointer p, T const& val);
    void destroy( pointer p);
}

template <class T, class S>
bool operator==(allocator<T> const&, allocator<S> const&) { return true; }
template <class T, class S>
bool operator!=(allocator<T> const&, allocator<S> const&) { return false; }
```

C++03 default allocator

```
template <class T>
inline T* allocator<T>::allocate( size_type n)
{
    return static_cast<T*>(::operator new(n*sizeof(T)));    // based on ::new
}

template <class T>
inline void allocator<T>::deallocate( T* p, size_t);
{
    ::operator delete(p);     // based n ::delete
}

template <class T>
inline void allocator<T>::construct( pointer p, T const& val)
{
    ::new((void*)p) T(val);   // call of placement-new
}


template <class T>
inline void allocator<T>::destroy( pointer p)
{
    ((T*)p)->~T();        // explicit call of destructor
}
```

C++03 default allocator

```
#include <iostream>
#include <vector>
#include <memory>

template <class T>
class MyAlloc : public std::allocator<T>
{
public:
  using value_type = T;
  using pointer = T*;
  using size_type = size_t;

  T* allocate(size_t n)
  {
    std::cout<<"custom alloc\n";
    return new T[n];
  }
  void deallocate (pointer p, size_t num) {
    std::cout << "custom deallocate\n";
    ::operator delete((void*)p);
  }
};

int main()
{
    std::vector<int, MyAlloc<int> > v;
    for (int i = 0; i < 16; ++i)   v.push_back(i);
    for(auto& e : v) std::cout << e << '\n';
    return 0;
}
```

C++03 default allocator

```cpp
#include <iostream>
#include <vector>
#include <memory>

template <class T>
class MyAlloc : public std::allocator<T>
{
public:
  using value_type = T;
  using pointer = T*;
  using size_type = size_t;

  T* allocate(size_t n)
  {
    std::cout<<"custom alloc\n";
    return new T[n];
  }
  void deallocate (pointer p, size_t num) {
    std::cout << "custom deallocate\n";
    ::operator delete((void*)p);
  }
};

int main()
{
    std::vector<int, MyAlloc<int> > v;
    for (int i = 0; i < 16; ++i)   v.push_back(i);
    for(auto& e : v) std::cout << e << '\n';
    return 0;
}
```

```
$ ./a.out
custom alloc
custom alloc
custom deallocate
custom alloc
custom deallocate
custom alloc
custom deallocate
custom alloc
custom deallocate
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
custom deallocate
```

Rebind

```
template <class T, class Allocator = allocator<T>>
class list
{
    // ...
    typedef typename Alloc::template rebind<list_node<T>>::other node_allocator;
    typedef typename node_allocator::pointer node_pointer;
    // ...
};

template <class T, class Alloc>
typename list<T,Alloc>::node_pointer list<T,Alloc>::alloc_node(T const& t)
{
    node_allocator  na(this->m_alloc);
    node_pointer    np = na.allocate(1u);
    na.construct( np, t);
    return np;
}
```

C++11 allocators

- Support synthetic pointers
  - Not to assume that pointer is T*
- Support stateful allocators
  - Not all allocator instances are equal
- Supporting relocatable objects
- Supporting scoped allocation

```
map<string, list<deque<string>>>   mp; // all should use the same allocator!
```

C++11 allocators

- Pointers nullable (e.g. construct from nullptr, conversion to bool)
- Pointer traits
  - Pointer to pointed-type
  - Conversion between different (or diff. cv qualified) pointer types
- Allocator requirements
- Allocator traits
  - This is the main interface containers use
- Allocator adaptor (scoped_allocator_adaptor helper)
  - Defines allocator propagation
- Container requirements
  - Defines containers’ allocator-aware interface

C++11 allocators

```
template <class T, class Alloc = allocator<T>>
class Container
{
    // typical containers
    using value_type      = T;
    using allocator_type  = Alloc;
    using alloc_traits    = std::allocator_traits<allocator_type>;

    using pointer         = typename alloc_traits::pointer;
    using const_pointer   = typename alloc_traits::const_pointer;
    using size_type       = typename alloc_traits::size_type;
    using difference_type = typename alloc_traits::difference_type;

    using reference       = value_type&;
    using const_reference = value_type const&;

    using iterator        = // ...
    using const_iterator  = // ...
    // ...
};
```

C++11 allocators

```
template <class Alloc>
struct allocator_traits       // reverse compatible with C++03
{
    using allocator_type      = Alloc;
    using value_type          = typename Alloc::value_type;

    using pointer             = // ... these values are infered from Alloc
    using const_pointer       = // ... or if it is not defined in Alloc
    using void_pointer        = // ... then defined "naturally", eg. value_type*
    using const_void_pointer  = // ...   pointer_traits<pointer>::rebind<const value_type>
    using size_type           = // ...
    using difference_type     = // ...

    using propagate_on_container_copy_assignment = // POCCA;  from Alloc or false_type
    using propagate_on_container_move_assignment = // POCMA;  from Alloc or false_type
    using propagate_on_container_swap             = // POCS; from Alloc or false_type
    using is_always_equal                         = // IZEQ; from Alloc or false_type

    template <class T> using rebind_alloc  = // from Alloc or ill formed
    template <class T> using rebind_traits = allocator_traits<rebind_alloc<T>>;

    static pointer allocate( Alloc& a, size_type n) { return a.allocate(n); }
    static pointer allocate( Alloc& a, size_type n, const_void_pointer hint); // may #1
    static void    deallocate( Alloc& a.  pointer p, size_type n) { a.deallocate(p,n); };

    template <class T, class... Args>                        // either a.allocate(...)
    static void construct( Alloc& a, T* p, Args&&... args); //   or placement new
    template <class T, class... Args>                        // either a.deallocate(...)
    static void destroy( Alloc& a, T* p);                    //   or p->~T()
};
```

Polymorphic Memory Resourse

- Provides run-time polymorphism (virtual functions)
  - Client allocators store a pointer to base class memory resource
  - Sticks for the lifetime of the container, not change on op=/swap
  - C++17 in std::pmr

```
namespace std::pmr {
class memory_resource  // in <memory_resource> header  in std::pmr
{
public:
    virtual ~memory_resource();
[[nodiscard]] void*  allocate( std::size_t bytes,
                              std::size_t alignment = alignof(std::max_align_t));
             void deallocate( void* p, std::size_t bytes,
                                std::size_t alignment = alignof(std::max_align_t) );
             bool is_equal( const memory_resource& other ) const noexcept;
private:
    virtual void*  do_allocate( std::size_t bytes, std::size_t alignment ) = 0;
    virtual void do_deallocate( void* p, std::size_t bytes, std::size_t alignment ) = 0;
    virtual bool do_is_equal( const std::pmr::memory_resource& other ) const noexcept = 0
};
bool operator==( const std::pmr::memory_resource& a,
                 const std::pmr::memory_resource& b ) noexcept;
bool operator!=( const std::pmr::memory_resource& a,
                 const std::pmr::memory_resource& b ) noexcept;
}
```

Polymorphic Memory Resourse

- Provides run-time polymorphism (virtual functions)
  - Client allocators store a pointer to base class memory resource
  - Sticks for the lifetime of the container, not change on op=/swap
  - C++17 in std::pmr

```
class monotonic_buffer_resource : pubic std::pmr::memory_resource // in <memory_resource>
{
public:
    monotonic_buffer_resource();
    explicit monotonic_buffer_resource(memory_resource *upstream);
    monotonic_buffer_resource(const monotonic_buffer_resource&) = delete;
    virtual ~monotonic_buffer_resource();
    monotonic_buffer_resource operator=(const monotonic_buffer_resource&) = delete;

    void             release();
    memory_resource* upstream_resource();
protected:
    virtual void*  do_allocate( std::size_t bytes, std::size_t al ) override;
    virtual void do_deallocate( void* p, std::size_t bytes, std::size_t al ) override;
    virtual bool do_is_equal( const std::pmr::memory_resource& o) const noexcept override;
};
```

Monotonic Buffer Resource

- Very fast allocations to build up objects
- Releases all memory at once by release or at destruction
- The virtual overriding of do_deallocate is no-op

```
#include <memory_resource>
#include <list>
#include <iostream>

int main()
{
    std::byte arena[1024];    // allocate raw memory for arena
    std::pmr::monotonic_buffer_resource mrs(arena, sizeof(arena));

    std::pmr::list<int> lst{&mrs};  // lst{{1,2,3,4}, &mrs} mrs is the last parameter
    for ( int i = 0; i < 30; ++i)
        lst.push_back(i);
    std::cout << "arena = " << &arena[0] << " -- " << &arena[1023] << '\n';
    std::cout << "list0 = "  << &lst.front() << " -- list29 = " << &lst.back() << '\n';
    return 0;
}
$ ./a.out
arena = 0x7ffcbe43b2c0 -- 0x7ffcbe43b6bf
list0 = 0x7ffcbe43b2d0 -- list29 = 0x7ffcbe43b588
```

Monotonic Buffer Resource

- Allocates new memory when the resource is fully allocated
  - Usually in the heap
  - Usually in geometric grow rate

```
#include <memory_resource>
#include <list>
#include <iostream>

int main()
{
    std::byte arena[1024];    // allocate raw memory for arena
    std::pmr::monotonic_buffer_resource mrs(arena, sizeof(arena));

    std::pmr::list<int> lst{&mrs};  // lst{{1,2,3,4}, &mrs} mrs is the last parameter
    for ( int i = 0; i < 50; ++i)
        lst.push_back(i);
    std::cout << "arena = " << &arena[0] << " -- " << &arena[1023] << '\n';
    std::cout << "list0 = "  << &lst.front() << " -- list49 = " << &lst.back() << '\n';
    return 0;
}
$ ./a.out
arena = 0x7ffe57479960 -- 0x7ffe57479d5f
list0 = 0x7ffe57479970 -- list49 = 0x558b85dc8f68
```

Synchronized pool resource

- Pool based memory resource
  - Collection of pools representing chunks of different size
  - May be accessed from multiple threads without synchronization
  - If memory resource accessed from only one thread useunsyncgronized_pool_ resource is better

```
// <memory_resource>
class synchronized_pool_resource : pubic std::pmr::memory_resource
{
    //

};

class unsynchronized_pool_resource : pubic std::pmr::memory_resource
{
    //

};
```

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

- 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
```

What about copying Res?

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 careful:
struct BadRes
{
    Res(int n) { cp = new char[n]; ... init(); ... }
    ~Res()     { delete [] cp; }
    char *cp;
    void init()
    {
       ... if (error) throw XXX;
    }
};
```

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 careful:
struct BadRes
{
    Res(int n) { cp = new char[n]; ... init(); ... }
    ~Res()     { delete [] cp; }
    char *cp;
    void init()
    {
       ... if (error) throw XXX;
    }
};
```

Possible memory leak if throws!

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 is here*/
}
```