EDAF50 – C++ Programming

5. Resource management

Sven Gestegård Robertz

Computer Science, LTH

2022

Outline

1. Resource management
   - Memory allocation
   - Stack allocation
   - Heap allocation: `new` and `delete`
2. Smart pointers
3. Classes, resource management
   - copy assignment
4. Move semantics
   - Move semantics (C++11)
5. type casts

Resource management

A *resource* is

- something that must be *allocated*
- and later *released*

Example:

- memory
- file handles
- sockets
- locks
- ...

Resource handles

Organize resource management with classes that *own* resources

- allocates resources in the constructor
- releases resources in the destructor
- *RAII* User-defined types that behave like built-in types

Memory Allocation

Two kinds of memory allocation:

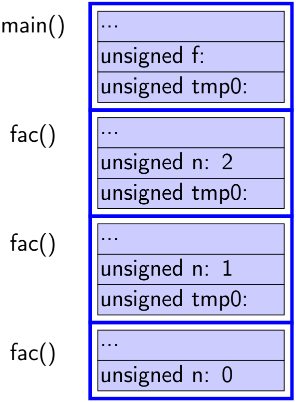
- on the *stack* - *automatic* variables. Are destroyed when the program exits the *block* where they are declared.
- on the *heap* - *dynamically allocated* objects. Live until explicitly destroyed.

Memory allocation

stack allocation

```
unsigned fac(unsigned n)
{
  if(n == 0)
    return 1;
  else return n * fac(n-1);
}

int main()
{
  unsigned f = fac(2);
  cout << f;
  return 0;
}
```



main()
...
unsigned f:
unsigned tmp0:

fac()
...
unsigned n: 2
unsigned tmp0:

fac()
...
unsigned n: 1
unsigned tmp0:

fac()
...
unsigned n: 0

- local variables are allocated on the stack in an activation record
- objects are destroyed when exiting their scope

Memory allocation

Dynamic memory, allocation "on the *heap*", or "*free store*"

Dynamically allocated memory

- is allocated on the *heap*, with **`new`** (like in Java)
  - does not belong to a *scope*
  - unnamed object: access through pointer or reference
  - **`new`** returns a pointer
- remains in memory until deallocated with **`delete`** (difference from Java)
- Objects allocated in dynamic memory can *outlive the scope they were allocated in*

Memory Allocation

Dynamic memory, allocation “on the *heap*”, or “*free store*”

Space for dynamic objects is allocated with **`new`**

```
double* pd = new double;                // allocate a double
*pd = 3.141592654;                      // assign a value
float* px;                              // uninitialized pointers
float* py;                              //   (avoid when possible)
px = new float[20];                     // allocate an array
py = new float[20] {1.1, 2.2, 3.3};     // allocate and initialize
```

Memory is released with **`delete`**

```
delete pd;
delete[] px; // [] is required for an array
delete[] py;
```

Memory Allocation

Warning! be careful with parentheses

Allocating an array: `char[80]`

```
char* c = new char[80];
```

Almost the same...

```
char* c = new char(80);
```

Almost the same...

```
char* c = new char{80};
```

The latter two allocate *one byte*

and *initializes* it with the value 80 (`'P'`).

```
char* c = new char('P');
```

Memory Allocation

Mistake: not allocating memory

```
char name[80];

*name = 'Z'; // OK, name allocated on the stack. name[0]='Z'



char *p;      // Uninitialized  pointer
              // No compiler warning

*p = 'Z';     // Error! 'Z' written to an undefined memory address

cin.getline(p, 80); //(almost) certain error during execution
                    //("Segmentation fault") or memory corruption
```

modern C++: **auto** is safer

```
auto q = new char[80];  // auto --> cannot be uninitialized
```

Memory Allocation

Example: failed `read_line` function

```
char* read_line() {
   char temp[80];
   cin.getline(temp, 80);
   return temp;
}

void exempel () {
  cout << "Enter your name: ";
  char* name = read_line();

  cout << "Enter your town: ";
  char* town = read_line();

  cout << "Hello " << name << " from " << town << endl;
}
```

"Dangling pointer": pointer to object that no longer exists

Memory Allocation

Partially corrected version of `read_line`

```
char* read_line() {
   char temp[80];
   cin.getline(temp, 80);
   size_t len=strnlen(temp,80);
   char *res = new char[len+1];
   strncpy(res, temp, len+1);
   return res; // dynamically allocated: survives
}
void exempel () {
   cout << "Enter your name";
   char* name = read_line();
   cout << "Enter your town";
   char* town = read_line();
   cout << "Hello " << name << " from " << town << endl;
}
```

Works, but memory leak !

Memory Allocation

Further corrected version of `read_line`

```
char* read_line() {
   char temp[80];
   cin.getline(temp, 80);
   size_t len=strnlen(temp,80);
   char *res = new char[len+1];
   strncpy(res, temp, len+1);
   return res;  Dynamically allocated: survives
}
void exempel () {
   cout << "Enter your name: ";
   char* name = read_line();   NB! calling function takes ownership
   cout << "Enter your town ";
   char* town = read_line();
   cout << "Hello " << name << " from " << town << endl;

   delete[] name;        Deallocate strings
   delete[] town;
}
```

Use `std::string`

Simpler and safer with `std::string`

```
#include <iostream>
#include <string>

using std::cin;
using std::cout;
using std::string;

string read_line()
{
  string res;
  getline(cin, res);
  return res;
}
```

```
void example()
{
    cout << "Name:";
    string name = read_line();
    cout << "Town:";
    string town = read_line();

    cout << "Hello, " << name <<
    " from " << town << endl;
}
```

- `std::string` is a *resource handle*
- *RAII*
- Dynamic memory is rarely needed (in user code)

Memory Allocation
ownership of resources

For dynamically allocated objects, *ownership* is important

- An object or a function can *own* a resource
- *The owner* is responsible for deallocating the resource
- If you have a pointer, you must know *who owns the object it points to*
- Ownership *can be transferred* by a function call
  - but is often not
  - be clear about owning semantics

Every time you write **`new`** you are responsible for that someone will do a **`delete`** *when the object is no longer in use*.

Classes
RAII

- *RAII Resource Acquisition Is Initialization*
- An object is initialized by a *constructor*
  - Allocates the resources needed (*"resource handle"*)
- When an object is destroyed, its *destructor* is executed
  - Free the resources owned by the object
  - Example: `Vector`: delete the array `elem` points to

```
class Vector{
private:
  double elem*;  // pointer to an array
  int sz;        // the size of the array
public:
  Vector(int s) :elem{new double[s]}, sz{s} {}  // ctor
  ~Vector() {delete[] elem;}   // dtor, delete the array
};
```

Manual memory management

- Objects allocated with **`new`** must be dellocated with **`delete`**
- Objects allocated with **`new`**`[]` must be dellocated with **`delete`**`[]`
- otherwise the program will *leak memory*

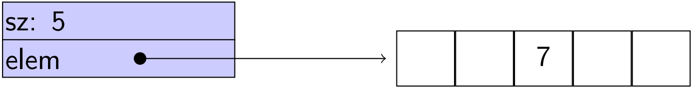
Classes

Resource management, representation

```
struct Vector {
  Vector(int s) :sz{s},elem{new double(sz)} {}
  ~Vector() {delete[] elem;}
  double& operator[](int i) {return elem[i];}
  int sz;
  double* elem;
};

void test()
{
   Vector vec(5);
   vec[2] = 7;
}
```

Vector vec:



- *Resource handle* – Vector owns its **double**[]
- the object: pointer + size, the array is on the heap

Dynamic memory, example

Error handling

```
void f(int i, int j)
{
    X* p=new X;             // allocate new X
    //...
    if(i<99) throw E{};     // may throw an exception
    if(j<77) return;        // may return "early"
    //
    p->do_something();      // may throw
    //
    delete p;
}
```

Will leak memory if `delete p` is not called

Memory allocation

C++: Smart pointers

The standard library `<memory>` has two "smart" pointer types (C++11):

- `std::unique_ptr<T>` – *a single owner*
- `std::shared_ptr<T>` – *shared ownership*

that are *resource handles*:

- their destructor deallocates the object they point to.
- Other examples of *resource handles*:
  - `std::vector<T>`
  - `std::string`

`shared_ptr` contains a *reference counter*: when *the last* `shared_ptr` to an object is destroyed, the object is destroyed. Cf. *garbage collection* in Java.

Smart pointer, example

```
void f(int i, int j)
{
    unique_ptr<X> p{new X};// allocate new X and give to unique_ptr
    //...
    if(i<99) throw E{};     // may throw an exception
    if(j<77) return;        // may return "early"
    //
    p->do_something();      // may throw
}
```

The destructor of p is always executed: no leak

Smart pointer, example
Dynamic memory is rarely needed

```
void f(int i, int j)
{
  X x{};

  if(i<99) throw E{};        // may throw an exception
  if(j<77) return;           // may return "early"

  x.do_something();         // may throw
}
```

Use local variables when possible

`read_line` with `unique_ptr`

```
unique_ptr<char[]> read_line()
{
  char temp[80];
  cin.getline(temp, 80);
  int size = strlen(temp)+1;
  char* res = new char[size];
  strncpy(res, temp, size);
  return unique_ptr<char[]>{res};
}

void example()
{
  cout << "Enter name: ";
  unique_ptr<char[]> name = read_line();
  cout << "Enter town: ";
  unique_ptr<char[]> town = read_line();
  cout << "Hello " << name.get() << " from " << town.get() << endl;
}
```

- To get a **char*** we call `unique_ptr<`**`char`**`[]>::get()`.
- Needed here to get right overload for **operator**<<

`read_line` with `unique_ptr` with no explicit **new** and **delete** (c++14)

```
unique_ptr<char[]> read_line()
{
    char temp[80];
    cin.getline(temp, 80);
    int size = strlen(temp)+1;
    auto res = std::make_unique<char[]> (size);
    strncpy(res.get(), temp, size);
    return res;
}
```

Smart pointers

`Vector` from previous examples

```
class Vector{
public:
    Vector(int s) :elem{new double[s]}, sz{s} {}
    double& operator[](int i) {return elem[i];}
    int size() {return sz;}
private:
    std::unique_ptr<double[]> elem;
    int sz;
};
```

- All member variables are of RAII types
- The default *destructor* works
- The object cannot be copied (no default functions generated)
    - A `unique_ptr` cannot be copied – it is *unique*

Smart pointers

`Vector` from previous examples

```
class Vector{
public:
    Vector(int s) :elem{new double[s]}, sz{s} {}
    double& operator[](int i) {return elem[i];}
    int size() {return sz;}
private:
    std::unique_ptr<double[]> elem;
    int sz;
};
```

- To make the type possible to copy
  - Define a copy constructor
  - Define a copy assignment operator

Memory allocation
C++: Smart pointers

Rules of thumb for pointer parameters to functions:

if ownership *is not* transferred

- Use “raw” pointers
- Use `std::unique_ptr<T> const &`

if ownership *is transferred*

.

- Use *by-value* `std::unique_ptr<T>`
  (then `std::move()` must be used)

- This is an orientation about smart pointers.
- “Raw” pointers are common; you must master them.

C++: Smart pointers
Coarse summary

“Raw” (“naked”) pointers:

- The programmer takes all responsibility
- Risk of memory leaks
- Risk of *dangling pointers*

Smart pointers:

- No (less) risk of memory leaks
- (minor) Risk of *dangling pointers* if used incorrectly (e.g., more than one `unique_ptr` to the same object)

Common pitfall
Default copying

For classes containing *owning pointers*,
the default copying does not work.

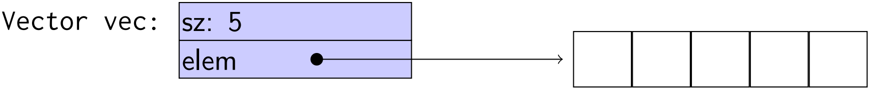
Example: `Vector`

- call by value
- copying pointer values
  (both objects point to the same resource)
- the destructor is executed on **`return`**
- *dangling pointer*
- *double delete*

Classes

Example: Copying the Vector class

```
class Vector{
public:
    Vector(int s) :elem{new double[s]}, sz{s} {}
    ~Vector() {delete[] elem;}
    double& operator[](int i) {return elem[i];}
    int size() {return sz;}
private:
    double* elem;
    int sz;
};
```



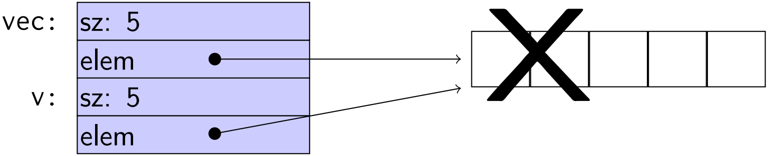
No copy constructor defined ⇒ default generated.

Classes

Default copy construction: shallow copy

```
void f(Vector v);

void test()
{
   Vector vec(5);
   f(vec); // call by value -> copy
   // ... other uses of vec
}
```



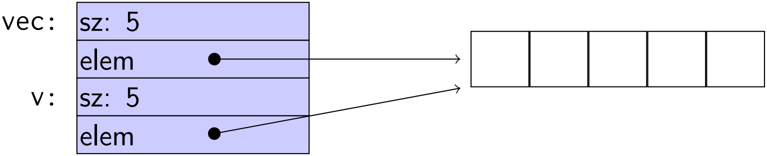
- The parameter v is default copy constructed: the value of each member variable is copied
- When `f()` returns, the destructor of v is executed: (`delete[] elem;`)
- The array pointed to *by both copies* is deleted. Disaster!

Copying objects

the *copy assignment* operator: `operator=`

The *copy assignment operator* is implicitly defined

- with the type `T& T::operator=(const T&)`
- if no `operator=` is declared for the type
- if all member variables can be copied
  - i.e., define a copy-assignment operator
- If all members are of built-in (and RAII) types the default variant works (same problems as with copy ctor).



- For *owning pointers*, the copy member functions must be implemented

"Rule of three"
Canonical construction idiom

IF a class owns a resource, it shall implement a

1. Destructor
2. Copy constructor
3. Copy assignment operator

in order not to leak memory. E.g. the class `Vector`

Rule:

If you define *any* of these, you should define *all*.

Copy control
Example: Vector

Copy constructor

```
Vector::Vector(const Vector& v) :elem{new double[v.sz]}, sz{v.sz}
{
    for(int i=0; i < sz; ++i) {
        elem[i] = v[i];
    }
}
```

Or, use the standard library:

```
std::copy(v.elem, v.elem+v.sz, elem);
```

Copy control
Example: Vector

Copy assignment

```
Vector& Vector::operator=(const Vector& v) {
  if (this != &v) {
      auto tmp = new double[v.sz];
      std::copy(v.elem, v.elem+v.sz,
                tmp);
      sz = v.sz;
      delete[] elem;
      elem = tmp;
  }
  return *this;
}
```

1. check “self assignment”
2. allocate new resources
3. copy values
4. free old resources

Only **delete** *if allocation succeeded.*

Lvalues and rvalues

Object lifetimes

- Applies to *expressions*
- An *lvalue* is an expression identifying an object (that persists beyond an expression)
- Examples:
  - `x`
  - `*p`
  - `arr[4]`
- An *rvalue* is a temporary value
- Examples:
  - `123`
  - `a+b`
- you can take the address of it ⇒ *lvalue*
- it has a name ⇒ *lvalue*
- Better rule than the old "Can it be the left hand side of an assignment?" (because of **`const`**)

Lvalues and rvalues

references

- An *lvalue reference* can only refer to a modifiable object
- An **const** *lvalue reference* can also refer to a temporary
    - Extends the lifetime of the temporary to the lifetime of the reference
- An *rvalue reference* can only refer to a temporary
- Syntax:

  (lvalue) reference: `T&`
  rvalue reference: `T&&` (C++11)

Move semantics

Making value semantics efficient

- Copying is unnecessary if the source will not be used again e.g. if
  - it is a *temporary value* ,e.g.
    - (implicitly) converted function arguments
    - function return values
    - `a + b`
  - the programmer explicitly specifies it
    `std::move()` is a *type cast* to *rvalue-reference* (`T&&`)
    (include `<utility>`)
- Some objects may/can not be copied
  - e.g., `std::unique_ptr`
  - use `std::move`
- Better to “steal” the contents
- Makes *resource handles* even more efficient

Move semantics
Making value semantics efficient

Move operations:

```
class Foo {
public:
    ...
    Foo(Foo&&);                // move constructor
    Foo& operator=(Foo&&); // move assignment
};
```

- look like copying, but
- "steals" owned resources instead of copying

"Rule of ~~three~~ five"
Canonical construction idiom, in C++11

If a class owns a resource, it should implement (or =`default` or =`delete`)

1. Destructor
2. Copy constructor
3. Copy assignment operator
4. *Move* constructor
5. *Move* assignment operator

Move constructor

implicitly generated

An automatically generated move constructor is provided if

- there are no user-declared copy constructors;
- there are no user-declared copy assignment operators;
- there are no user-declared move assignment operators;
- there is no user-declared destructor.

Move constructor
Example: Vector

Move constructor (C++-11)

```
Vector::Vector(Vector&& v) : elem{v.elem}, sz{v.sz}
{
    v.elem = nullptr;
    v.sz = 0;         // v has no elements
}
```

Copy control: (Move semantics – C++11)

Example: Vector

Move assignment

```
Vector& Vector::operator=(Vector&& v) {
    if(this != &v) {
        delete[] elem;      // delete current array
        elem = v.elem;      // "move" the array from v
        v.elem = nullptr;   // mark v as an "empty hulk"
        sz = v.sz;
        v.sz = 0;
    }
    return *this;
}
```

Type casts
Implicit conversions

Automatic conversions

- Expressions of the type $x \odot y$, for some binary operator $\odot$
  E.g.: **`double + int ==> double`**

  `float + long + char ==> float`
- Assignments and initialization: The value of the right-hand-side is converted to the type of the left-hand-side
- Conversion of an argument to the type of the (formal) parameter
- Expresions in `if` statements, etc. ⇒ **`bool`**
- built-in array ⇒ pointer (*array decay*)
- `0` ⇒ `nullptr` (empty pointer in C++11, previously the constant `NULL` was defined)

type casts
Explicit, named type casts (C++-11)

- **`static_cast`**`<new_type> (expr)`
  - convert between compatible types (*does not do range check*)
  - "the inverse of a *standard* implicit conversion" (mostly)
- **`reinterpret_cast`**`<new_type> (expr)`
  - no safety net, same as C-style cast
- **`const_cast`**`<new_type> (expr)` - add or remove **`const`**
- **`dynamic_cast`**`<new_type> (expr)` - use for pointers to objects in class hierarchies. Uses *run-time type info*, like `instanceof` in Java.

Example

```
char c;                // 1 byte
int *p = (int*) &c;    // pointer to int: 4 bytes

*p = 5; // undefined behaviour, stack corruption

int *q = static_cast<int*> (&c); // compiler error
```

Type casting

Explicit type casts, C style

Syntax in C and in C++, like in Java

`(type) expression` , e.g. `(float) 10`

- Greater risk of mistakes - use named casts
  - makes the code clearer, e.g., **`const_cast`** can only change **`const`**
  - easy to search for: casts are among the first to look for when debugging
- Warning in GCC: `-Wold-style-casts`
- Common in older code

Alternative syntax in C++

```
type(expression)
```

`type` must be *a single word*,

`int` `*(...)` eller i.e., **`unsigned long`**`(...)` is not OK.

Data types and variables

- some concepts:
    - a *type* defines the set of possible values and operations (for an *object*)
    - an *object* is a place in memory that holds a *value*
    - a *value* is a sequence of bits interpreted according to a *type*.

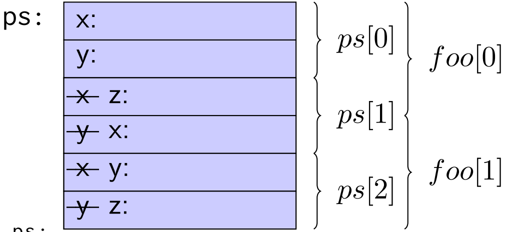
A typecast changes the ***value*** *of a particular memory location by changing how* ***it should be interpreted****.*

Type casts
Warning example

```
struct Point{
    signed char x;
    signed char y;
};

Point ps[3];

struct Point3d{
     signed char x;
     signed char y;
     signed char z;
};
Point3d* foo = (Point3d*) ps;
```



With *named casts*, this requires a **reinterpret_cast**<Point3d*>

With **static_cast**<Point3d*> the compiler gives the error
invalid **static_cast** from type 'Point[3] to type 'Point3d*'

special case: **`void`** pointer

A **`void`**`*` can point to an object of any type

In C
: a **`void`**`*` is implicitly converted to/from any pointer type.

In C++
: a `T*` is implicitly converted to **`void`**`*`. The other direction requires an explicit *type cast*.

Next lecture: Algorithms

References to sections in Lippman

Function templates	16.1.1
Algorithms	10 – 10.3.1, 10.5
Iterators	10.4
Function objects	14.8
Random numbers	17.4.1

Suggested reading

References to sections in Lippman

Dynamic memory and smart pointers 12.1

Dynamically allocated arrays 12.2.1

Classes, resource management 13.1, 13.2

Moving objects 13.6

Type casts 4.11