EDAF30 – Programming in C++

6. Resource management

Sven Gestegård Robertz
Computer Science, LTH

2018

Outline

① Resource management
- Memory allocation
- Stack allocation
- Heap allocation: new and delete

② Smart pointers

③ Classes, resource management
- Rule of three
- copy assignment

④ Function calls

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
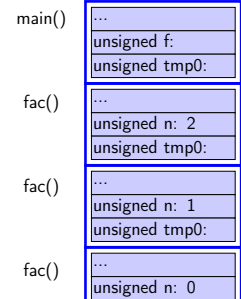
Memory Allocation

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
```

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*
  - ▶ remains in memory until deallocated with **delete** (difference from Java)

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;
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=strlen(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`

```cpp
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`

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

using std::cin;                void example()
using std::cout;               {
using std::string;                 cout << "Name:";
                                   string name = read_line();
string read_line()                 cout << "Town:";
{                                  string town = read_line();
    string res;
    getline(cin, res);             cout << "Hello, " << name <<
    return res;                    " 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

```cpp
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

```cpp
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

```cpp
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().

---

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

```cpp
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_pointer<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

```cpp
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;
};
```
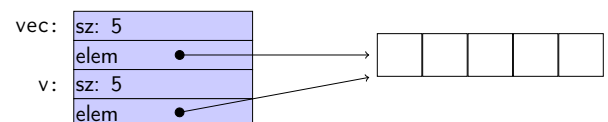

Vector vec: sz: 5 / elem
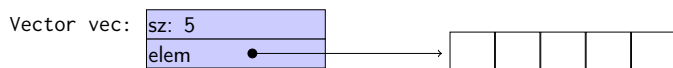
No copy constructor defined ⇒ default generated.

Classes
Default copy construction: shallow copy

```cpp
void f(Vector v);

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


vec: sz: 5 / elem
v: sz: 5 / elem

- ► 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).
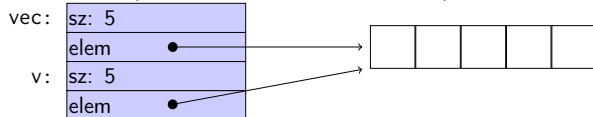
- ▶ For *owning pointers*, the copy member functions must be implemented

---

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

Copy control
Example: Vector

> **Copy assignment**
> ```cpp
> Vector& Vector::operator=(const Vector& v) {
>     if (this != &v) {
>         auto tmp = new double[v.sz];
>         for (int i=0; i<v.sz; i++)
>             tmp[i] = v.elem[i];
>         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.*

Function calls and results
Returning objects by value

- ▶ A function cannot return references to local variables
  - ▶ the object is destroyed at **return** – *dangling reference*
- ▶ How (in)efficient is it to return objects by value (a copy)?

---

return value optimization (RVO)

The compiler may optimize away copies of objects on **return** from functions

- ▶ *return by value* often efficient, also for larger objects
- ▶ RVO allowed *even if the copy-constructor or destructor has side effects*
- ▶ avoid such side effects to make code portable

Rules of thumb for function parameters

- ▶ Return by value more often
- ▶ Do not over-use call-by-value

"reasonable defaults"

	cheap to copy	moderately cheap to copy	expensive to copy
In	f(X)	f(const X&)	
In/Out	f(X&)		
Out	X f()		f(X&)

For results, if the cost of copying is

- ▶ small, or moderate ($< 1k$, contiguous): return by value (modern copilers do RVO: return value optimization)

- ▶ large : call by reference as *out parameter*
  - ▶ or maybe allocate with **new** and return pointer

Call by reference or by value?
Rules of thumb

For passing an object to a function when

- ▶ you may want *to change the value* of the object
  - ▶ reference: **void** f(T&); or
  - ▶ pointer: **void** f(T*);
- ▶ you *will not* change it, it is *large* (or impossible to copy)
  - ▶ constant reference: **void** f(**const** T&);
- ▶ otherwise, *call by value*
  - ▶ **void** f(T);

reference or pointer?

- ▶ required parameter: pass reference
- ▶ optional parameter: pass pointer (can be nullptr)

```
void f(widget& w)
{
    use(w); //required parameter
}

void g(widget* w)
{
    if(w) use(w); //optional parameter
}
```

Call by reference or by value?
Rules of thumb

- ▶ How big is "large"?
  - ▶ more than a few *words*
- ▶ When to use out parameters?
  - ▶ prefer code that is obvious
    Example: two functions:

```
void incr1(int& x)              int v = 0;
{                               ...
    ++x;
}                               incr1(v);
                                ...
int incr2(int x)
{                               v = incr2(v);
    return x + 1;          Here it is much clearer
}                           that v = incr2(v) changes v
```

- ▶ For multiple output values, consider returning a **struct**, a std::pair or a std::tuple

Next lecture: Error handling

References to sections in Lippman

Error handling, exceptions (5.6, 18.1.1)

static assert   *not in Lippman*

assert   6.5.3

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