Part 7 — Odds and Ends

Error Handling and Exceptions

Historical method used in C:

- Use function return codes to indicate error conditions

```
int fgetc(FILE *stream);
```

- Returns read character (value in 0..255)
- or -1 if read error occurred

Drawbacks

- What if function returns full range of values?
- Errors can be easily ignored

Modern solution: Exceptions

Exceptions

- Dealing with rare error conditions
- Write code as if nothing can go wrong
- Enclose it in try-block which will be exited if some operation fails and throws an exception
- Add a catch-block to handle exceptions

Exceptions

```cpp
void foo1() {
    vector<int> v1;
    foo2();
    g();        // Not reached
}               // v1 destroyed

void foo2() {
    std::string s2;
    foo3();
    g();        // Not reached
}               // s2 destroyed

void foo3() {
    throw MyException();    // Object created
    g();                    // Not reached
}

int main() {
    try { foo1(); }
    catch (MyException &e) {
        // Execution continues here
    }   // Exception obj. destroyed
}
```

Exceptions

Once an exception is thrown (can be any type!), program execution is suspended.

The runtime system then looks for the next catch statement whose type is compatible (i.e., exact match or inheritance ancestor) with the thrown value:

- If the exception was thrown in a try block, the following catch statements are checked
- If there is no match, the search for an exception handler resumes in the caller ("*stack unwinding*") after all local objects have been destroyed
- If no matching catch statement is found, the program is aborted by calling `std::terminate()`

When match is found, the execution resumes there and at the end of the catch block, the thrown object is destroyed.

Function Definitions and throw

```
void f() throw {}
```

- [Depreciated in C++11]
- `f` is not allowed to throw anything

```
void f() throw(IOException, MyException){}
```

- [Depreciated in C++11]
- `f` is not allowed to throw anything

```
void f() {}
```

- `f` is allowed to throw anything

```
void f() noexcept {}
```

- New in C++11
- `f` is not allowed to throw anything

Try-Catch Blocks

```cpp
try {
    // do stuff as if nothing can ever happen
}
catch (MyException &e) {
    // handle MyException here, get access to
    // exception data through variable e
}
```

There can be multiple catch blocks, including

```cpp
catch (...) { } // catches all exceptions
```

After doing some work in the catch block, the exception can be re-thrown to continue the search for the next matching catch statement up the call stack:

```cpp
... throw; ...
```

Try-Catch Blocks

Catch block ordering matters! Exact types are matching, but also ancestor types in the inheritance hierarchy!

```cpp
struct MyException : public std::exception { };

try {
    ...
    throw MyException();
    ...
}
catch (std::exception &e) {
    // matches
    ...
}
catch (MyException &e) {
    // never reached because
    // std::exception is an
    // ancestor of MyException
    ...
}
```

How to Catch Exceptions

- Catch-by-pointer?

```
catch (T *p) { ... }
```

E.g. We could do this

```
throw new MyException;
// or
MyException e;  // global variable
...
throw &e;
...
catch (MyException *p) { ... }
```

At this point it is impossible to know whether to delete or not.

- Catch-by-value?

```
catch (T v) { ... }
```

One additional copy, possible slicing!

How to Catch Exceptions

Consequently, the only option left for catching exceptions is by reference:

```
catch (T &v) { ... }
```

Also, be aware that catching exceptions is expensive — exceptions should be rare events, and not used in the regular flow of your program

- The rules with `const` still applies!
- If we are not going to modify any part of the exception object, catch by const-reference!

Operator new and Exceptions

- `new` throws `std::bad_alloc` in case memory is unavailable
- Thus, checking the result of new $\neq 0$ is a waste of time — it's always $\neq 0$
- The C++ standard demands that memory is available if `new` doesn't throw
- In practice, however, this is operating system dependent

I.e.: In some operating systems such as Linux memory allocation almost always succeeds, and you'll learn that you don't have enough memory later when you start accessing memory — segfault ...

Other Exception Pitfalls

Prevent resource leaks in constructors.

- Destructors are only called for fully constructed objects

Prevent exceptions from leaving destructors

- Exceptions within exceptions terminate program
- Since C++11, destructors are by default `noexcept`!
- **Special case**: exceptions call destructors ...

Exception Safety

A program is called exception safe if in the case exceptions are thrown no resources are leaked

Here is an example which is exception unsafe:

```cpp
void bar() {
    throw MyException();
}

void foo() {
    int *p = new int[1000];
    bar();
    delete [] p;    // not executed -> memory leak
}
```

Resource Acquisition is Initialization (RAII)

Such resource leaks can be eliminated by following the Resource Acquisition is Initialization (RAII) programming paradigm:

- Acquire resources only in constructors and release them only in destructors
- Every time (even when exceptions are thrown) when objects go out of scope, their destructor is called. This, when applying the RAII paradigm, will then release resources like memory, file handles, or mutex locks
- However, exceptions thrown in constructors MUST be handled right away to free resources (and maybe re-thrown), because destructors are not called on partly constructed objects

Resource Acquisition is Initialization (RAII)

Also: Exceptions must not leave destructors

- If an exception occurs in a destructor while unwinding the stack, the program terminates
- A partly completed destructor has not done its job

Resource Acquisition is Initialization (RAII)

Say *good-bye* to using local pointers for memory allocation

- `T *p = new T; ... delete p;`
- `delete p` may not be executed if an exception is thrown in ... !
- **Solution**: smart pointers (coming up)

Open output file stream (ofstream) with constructor call

- `ofstream os("output.txt");`
- When os goes out of scope, the file is closed automatically

RAII Examples

To prevent data corruption by concurrent write accesses to shared data, locking critical regions in concurrent programs is crucial

```cpp
struct Count {
    int id;
    Count(int id) : id(id) { }
    void operator()() {
        for (int i = 0; i < 10; ++i) {
        { // critical region, make sure only one thread prints to cout
          // and changes shared data by using a lock:
          // - constructor of lock locks mutex
          // - if mutex is locked, no other thread can enter region
          lock_guard<mutex> lock(my_mutex);
          cout << id << ": " << shared++ << endl;
          // when leaving scope, mutex gets unlocked; if not done in
          // destructor program could get dead-locked when exception
          // is thrown, meaning that all other threads wait, but the
          // mutex never gets released
        }
        sleep(1);
        }
    }
};
```

RAII Examples

```
#include <thread>
#include <mutex>
#include <iostream>
#include <unistd.h>
using namespace std;

mutex my_mutex;
int shared = 0;

// struct Count ...
int main(int argc, char *argv[])  {
    thread t1(Count(1));    // create thread, running Count(1)()
    thread t2(Count(2));    // create thread, running Count(2)()

    // wait for both threads to finish
    t1.join();
    t2.join();
    return 0;
}
// g++ thread.c -lpthread
```

Smart Pointers

Objects that look, act, and feel like regular pointers. Used for resource management. E.g.

- Reference counting
- Solving the pointers and exceptions problem

Gain control over:

- Construction and destruction
- Copying and assignment
- Dereferencing

Smart Pointers

Boost's `shared_ptr` bade it into the C++11 standard. Its `scoped_tr` and `scoped_array` functionality is supported by the new `unique_ptr` smart pointer class.

`unique_ptr<T>`, `unique_ptr<T[]>`

- Simple sole ownership of single object or array, resp.
- Will free memory correctly when going out of scope (calls `delete` or `delete[]` resp.)
- Cannot be copied (safeguard), storing them in STL containers is problematic if elements get copied

`shared_ptr`

- Shared, reference counted ownership of single object
- Causes no problems when stored in STL containers
- Cannot handle cyclic data structures

unique_ptr Example

```cpp
#include <memory>
using namespace std;

void foo() {
    // unique_ptr owns new Foo object
    auto p = make_unique<Foo>();
    // old way (obsolete):
    // unique_ptr<Foo> p(new Foo);

    unique_ptr<Foo> q = p; // illegal, safeguard!
    p->bar(); ... // use like regular pointer

    // also works for arrays
    auto pa = make_unique<Foo[]>(100);
    // old way (obsolete):
    // unique_ptr<Foo[]> pa(new Foo[100]);

    unique_ptr<Foo[]> qa = pa;  // illegal
    pa[10].bar(); // use like regular array
    pa->bar();    // illegal

    // p destroyed here => destroys Foo object
    // pa destroyed here => destroys Foo array
}
```

shared_ptr Example

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

void foo(shared_ptr<Foo> &q) {
    // allocate new Foo and initialize shared
    // pointer with address
    auto p = make_shared<Foo>();        // ref. count 1
    // old way (obsolete):
    // shared_ptr<Foo> p(new Foo);
    q = p;                              // pointer copy => reference count 2

    // p destroyed here => reference count 1
    // Foo object not destroyed yet!
}

void main() {
    shared_ptr<Foo> q;

    foo(q); ...
    // q destroyed here
    // => reference count 0 => object destroyed
}
```

Smart Pointers

Using smart pointers helps making functions exception-safe:

```cpp
void foo() {
    // old: int *p = new int[100];
    auto p = make_unique<int[]>(100);
    bar();
    // p goes out of scope: release array
    // even if bar() throws an exception
    // old: delete [] p;
}
```

- If `bar()` throws an exception then p is destroyed in the stack unwinding process $\longrightarrow$ no memory leak
- When using old-style memory allocation code the `delete` statement is not reached when `bar()` throws an exception $\longrightarrow$ memory leak
- In addition, we don't have to worry about matching new/delete brackets anymore!

unique_ptr

Example implementation

```cpp
template <class T>
class unique_ptr {
private:
    T *px; // wrapping a plain old pointer

    // non-copyable
    unique_ptr(const unique_ptr &) = delete;
    unique_ptr &operator=(const unique_ptr &) = delete;

public:
    // explicit: need to pass on value of exact
    // type T*; no implicit conversions performed
    // to create match (see below)

    explicit unique_ptr(T *p=nullptr): px(p) { }
    ~unique_ptr() { delete px; }
    T &operator*() const { return *px; }
    // member data access, e.g. p->a = 0
    T *operator->() const { return px; }
};
```

Move Semantics

`unique_ptr` also supports "*move semantics*", i.e. moving ownership around without having to copy and delete temporary objects. `unique_ptrs` can be stored in STL containers as long as no copy operations are performed.

To allow this for *move semantics*, C++11 introduces the *move constructor* and the *move assignment operator*.

```
X(const X &other);      // Copy Constructor, other is an lvalue reference
X(X &&other);           // Move Constructor, other is an rvalue reference

int main() {
    X x1;
    X x2(std::move(x1));
}
```

Note: `std::move` doesn't actually move! It simply type-casts an lvalue to an rvalue.

Move Semantics Example

```cpp
#include <memory>
auto pA = make_unique<int[]>(10); // array, def. constr.
auto p1 = make_unique<int>(5);    // single value
unique_ptr<int> p2 = p1;          // error: copy not allowed
unique_ptr<int> p3 = move(p1);    // transfers ownership:
// p3 owns the obj. and p1 is rendered invalid
p3.reset();                       // frees memory
p1.reset();                       // does nothing

using V = vector<unique_ptr<int>>;

V v1;                             // fine
V v2(begin(v1), end(v1));         // error (copy)
sort(begin(v1), end(v1));         // fine, because sort
// can move things around instead of copying
```

Explicit Constructors

```cpp
struct A {
    A(int x) { }
};

// this is legal C++ code:
A a = 37;
// really? compiler is looking for conversion int -> A
// and finds "converting constructor" A(int)
```

To disallow this confusing syntax, use `explicit`:

```cpp
struct A {
    explicit A(int x) { }
};
```

This disables the implicit conversion:

```cpp
A a = 37; // Now illegal
```

Instead, we have to use:

```cpp
A a(37);
```

What else in C++11/14?

After 8 years in the making a new C++ standard was passed in 2011

It introduced a multitude of new features. Some of them are beyond this introduction. Others we have seen already

Here is a brief description of some of the new features. To learn more about C++11/14, please visit Wikipedia or read some newer books on the subject, such as

- B. Stroustrup: The C++ Programming Language (4th Edition)
- S. Meyers: Effective Modern C++
- S. Meyers: Overview of the New C++ (C++11/14)
- N.M. Josuttis: The C++ Standard Library - A Tutorial and Reference, 2nd Edition

What else in C++11/14?

Core language usability enhancements:

- Initializer lists `vector<int> x{3,4,5}`
- auto type inference `auto it = begin(cont);`
- decltype type inference (1)
- Range-based for loop : `for (auto &x : cont)`
- Lambda functions (see below)
- Null pointer constant `nullptr`
- Strongly typed enumerations (see below)
- Alias templates (2)
- Constant expressions (3)

```
decltype(l) x;                                          // (1) has same type as l
template <class T, unsigned I, unsigned J>              // (2)
using array2 = std::array<std::array<T, J>, I>;
array2<int, 3, 4> a34;
constexpr int square(int x) { return x*x; }             // (3)
// can be evaluated at compile time - square(10)
```

What else in C++11/14?

Core language functionality improvements

- Move semantics (1)
- Variadic templates (2)
- User-defined literals (3)
- Explicitly defaulted and deleted special member functions (4)
- Type `long long int` (5)
- Static assertions (6)

```cpp
vector<int> w...,                                 // (1) moves w into v
v(std::move(w));                                  // w empty afterwards
template<typename... Values> class tuple;         // (2)
Length l = 3.5_cm;                                // (3)

struct X {
    X(const X &) = default;                       // (4) default CC
};
long long int x;                                  // (5) >= 64 bit integer
static_assert(sizeof(x) == 8, "wrong size")       // (6) compile time check
```

What else in C++11/14?

C++ Standard Library additions

- Upgrades to standard library components
- Threading facilities
- Tuple types (1)
- Hash tables (2)
- Regular expressions (3)
- General-purpose smart pointers (4)
- Extensible random number facility (5)
- Type traits for meta-programming

```
std::tuple<int,char,double> my_tuple;                  // (1) 3 values
std::unordered_set<int> hash_table;                    // (2)
std::regex rx("hello");                                // (3)
regex_match(begin(str), end(str), rx);
std::unique_ptr<int> p(new int);                       // (4)
std::uniform_int_distribution<int> distr(0, 99);    // (5)
std::mt19937 engine;              // Mersenne twister MT19937
int random = distr(engine);       // generate random number
```

Type Inference, auto, decltype

auto can be used to infer rhs types automatically:

```cpp
auto x = 27;                        // int
const auto cx = x;                  // const int
const auto &rx = x;                 // const int&
for (const auto &p : m){            // iterate through elems.
    ...                             // of m via const references
}
```

C++14 adds the ability to deduce function return types and lambda parameters

```cpp
auto func() {    // C++14: return type int is deduced
    return 1;
}
```

auto variables must be initialized, are generally immune to type mismatches that can lead to portability or efficiency problems, can ease the process of refactoring, and typically require less typing than variables with explicitly specified types.

Type Inference, auto, decltype

`decltype` infers types of expressions and function return values and can be used in declarations like so:

```cpp
double x;
decltype(x) y;       // y has x's type (double)

decltype(foo()) z;   // z has foo's return type

std::vector<decltype(foo())> v{foo()}; // cannot use auto

auto foo() -> int; // declares function foo returning int
                   // using functional trailing return
                   // type notation

template <typename T, typename U>
auto sum(T t, U u) -> decltype(t+u);
// return type is the type of t+u
```

This is a good talk about type deduction: www.youtube.com/watch?v=wQxj20X-tIU

Braced Initialization

There are three ways of initializing a variable:

```cpp
int x(0);                  // initializer in parenthesis
int y = 0;                 // initializer follows "="
int z{0};                  // initializer is in braces
int z = {0};               // equivalent to braces
```

Initialization using = is not an assignment:

```cpp
Widget w1;                 // calls default constructor
Widget w2 = w1;            // not an assignment! calls copy ctor
w1 = w2;                   // assignment, calls operator=
```

The new "*Braced Initialization*" (or Uniform Initialization) allows for previously inexpressible initializations. Using braces, specifying the initial contents of a container is easy:

```cpp
std::vector<int> v{1};         // v's initial content is 1
std::vector<int> u(1);         // u created with size one, but content is 0
```

Braced Initialization

Braces can also be used to specify default initialization values for non-static data members

```cpp
class Widget {
    ...
private:
    int x{0};           // fine, x's default value is 0
    int y = 0;          // also fine
    int z(0);           // error!
};
```

Uncopyable objects (like `std::atomic`) may be initialized using braces or parenthesis, but not equals:

```cpp
std::atomic<int> ai1{0};            // fine
std::atomic<int> ai2(0);            // fine
std::atomic<int> ai3 = 0;           // error!
```

This is why braced initialization is called uniform initialization, it can be used everywhere

Braced Initialization

Braced initialization forbids narrowing conversions among built-in types (for safety), and it can be used explicitly without parameters

```
double x, y, z;
int sum{x + y + z};     // error! sum of doubles may not
                        // be expressible as an int
int sum(x + y + z);     // OK, value truncated to an int
Widget w1(10);          // calls ctor with argument 10
Widget w2();            // declares function w2 that returns a Widget - oops!
Widget w3{};            // calls Widget default constructor
```

Classes can support brace initializations like so:

```
#include <initializer_list>
struct S {
    std::vector<int> v;
    S(std::initializer_list<int> list) {
        for (const auto &x : list) { v.push_back(x); }
    }
};
int main() {
    S s{1, 2, 3, 4, 5};     // copy list-initialization
}
```

Alias Declarations

C++11 introduced an alternative to `typedef`: alias declarations:

```cpp
typedef std::unordered_map<std::string, std::string> MapSS;
using MapSS = std::unordered_map<std::string, std::string>;

// FP is a synonym for a pointer to a function taking an
// int and a const std::string & and returning nothing
typedef void (*FP)(int, const std::string &);
using FP = void (*)(int, const std::string &);
```

In some cases it can make the code slightly more readable. However, the most compelling reason to use them is alias templates:

```cpp
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>;

MyAllocList<Widget> lw;
```

Alias Declarations

Compare this to the equivalent `typedef` code:

```cpp
template<typename T>
struct MyAllocList {
    typedef std::list<T, MyAlloc<T>> type;
};

MyAllocList<Widget>::type lw;
```

And if you want to use the typedef inside a template to specify the type of a data member, you need to add `typename` to the declaration:

```cpp
template<typename T>
class Widget {
    typename MyAllocList<Widget>::type list;
};
```

While with the alias template you can use it directly:

```cpp
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>; // as before

template<typename T>
class Widget {
    MyAllocList<Widget> list;
};
```

Scoped Enums

C++98 style enums can pollute the namespace:

```cpp
enum Color { black, white, red };   // black, white, red are in same scope as Color
auto white = false;                 // error! white already declared
```

C++11 scoped enums don't leak names into the scope containing their enum definition:

```cpp
enum class Color {black, white, red};   // black, white, red are scoped to Color
auto white = false;                     // fine, no other "white" in scope
Color c = white;                        // error!
Color c = Color::white;                 // OK
auto c = Color::white;                  // OK
```

Scoped enums don't implicitly convert to integral types. For this, `static_cast` is necessary:

```cpp
enum Coord {X, Y};
std::cout << "X= "   << static_cast<int>(Coord::X)
          << ", Y= " << static_cast<int>(Coord::Y);
```

Constant Expressions

The `constexpr` specifier declares that it is possible to evaluate the value of the function or variable at compile time. Such variables and functions can then be used where only compile time constant expressions are allowed (provided that appropriate function arguments are given).

Let's start with `constexpr` variables: they are `const` variables with values know at compile time:

```cpp
int i;                        // non-constexpr variable
constexpr auto j = i;         // error, i value not know at compilation time
std::array<int, i> a1;        // error, same problem
constexpr auto k = 10         // OK
std::array<int, k> a1;        // OK
```

Constant Expressions

`constexpr` functions can also be used wherever a compile-time value is needed. In C++11 these functions are very restricted, they can basically contain just a single return statement (which can include the conditional "?:" operator):

```cpp
constexpr int pow(int base, int exp) {
    return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

C++14 relaxed the conditions a bit, by allowing local variables, conditionals and loops:

```cpp
constexpr int pow(int base, int exp) {
    auto result = 1;
    for (int i = 0; i < exp; ++i) {
        result *= base;
    }
    return result;
}
```

Lambda Functions

A lambda function creates a *closure*, an unnamed function object capable of capturing variables in scope. Their main use is to simplify the use of STL's generic algorithms. Previously, you had to create a named function object:

```cpp
struct Bigger {
    bool operator()(int i) {return i > 3;}
};
std::vector<int> v{1, 2, 3, 4, 5};
int n= std::count_if(begin(v), end(v), Bigger());
```

With lambda functions, you can do it on the spot:

```cpp
std::vector<int> v{1, 2, 3, 4, 5};
    int n = std::count_if(begin(v), end(v),
                            [](int i) { return i > 3; }
                         );
```

Lambda Functions

The lambda function can capture local variables and the `this` pointer in the scope where it is defined. The capture list is a comma-separated list of zero or more captures, optionally beginning with the capture-default.

The only capture defaults are & (implicitly catch local variables and this by reference) and = (implicitly catch local variables and this by value).

```cpp
struct S2 { void f(int i); };

void S2::f(int i) {
    [&, i] {};    // ok: capture all by reference and i by value
    [&, &i] {};   // error: i preceded by & when & is the default
    [i, i] {};    // error: i repeated
}
```

Lambda Functions

Closures can also be referred to by variable names and copied:

```cpp
int x{2};                  // a local variable
auto c1 = [x](int y){      // c1 is a copy of the closure
    return x*y;            //   produced by the lambda
};

auto c2 = c1;
int z = c2(3);             // z = 6;
```

As a final example, the following code replaces all elements in a vector smaller than 5 with 55, and then prints all its elements:

```cpp
std::vector<int> v{1, 2, 3, 4, 5, 6, 7};

int x = 5;

std::replace_if(begin(v), end(v),
                [x](int i) { return i < x; }, 55);

std::for_each(begin(v), end(v),
              [](int i) { std::cout << i << ' '; });

// output: 55 55 55 55 5 6 7
```