Part 7: Odds and Ends

Contents

[DOCUMENT NOT FINALIZED YET]

Error Handling and Exceptions

Historical method used in C:

Use function return codes to indicate error conditions

E.g. `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

Example:

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

void foo1() {
  vector<int> v1;
  foo2();
  g(); // not reached
} // but v1 destroyed

void foo2() {
  string s2;
  foo3();
  g(); // not reached
} // but s2 destroyed

void foo3() {
  throw MyException(); // object created
  g(); // not reached
}
```

Catching 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() { }` [deprecated in C++11]

f is not allowed to throw anything

`void f() throw(IOException, MyException){}` [deprecated in C++11]

If `f` throws an exception not on the list, function `std::unexpected()` is called (program terminates)

`void f() { }`

f is allowed to throw anything

`void f() noexcept { }` [new in C++11]

f is not allowed to throw anything

Usual throw syntax:

```
throw MyException();
// creates object and starts the stack-unwinding
// process to find a matching catch clause;
// can use any type we want
```

try-catch Blocks

```
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

```
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:

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

Catch block ordering matters!

Exact types are matching, but also ancestor types in the inheritance hierarchy!

```
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) ...` delete or not delete?
  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!

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

Operator `new` and Exceptions

`new` throws `std::bad_alloc` in case memory is unavailable

Thus, checking the result of `new` (!=0) is a waste of time — it's always `!= 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

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:

```
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

Everytime (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 rethrown), because destructors are not called on partly constructed objects

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

RAII Examples

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

Another RAII Application: Locking

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

```
#include <thread>
#include <mutex>
#include <iostream>
#include <unistd.h>
using namespace std;
mutex my_mutex;
int shared = 0;

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);
    }
  }
};
```

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

  // create thread, running Count(2)()
  thread t2(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 (since C++11)

Boost's `shared_ptr` made it into the C++11 standard. Its `scoped_ptr` 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)
  So, storing them in STL containers is problematic if elements get copied

`shared_ptr<T>`

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

unique_ptr Examples

```
#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 Examples

```
#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
}
```

Using smart pointers helps making functions exception-safe:

```
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 $\leadsto$ no memory leak

When using old-style memory allocation code (red) the `delete` statement is not reached when `bar()` throws an exception $\leadsto$ memory leak

In addition, we don't have to worry about matching new/delete brackets anymore!

unique_ptr Implementation Sketch

```
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; }
};
```

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

Examples

```
#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

```
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`:

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

This disables the implicit conversion:

```
A a = 37; // now illegal
```

We have to use

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

instead

What else is 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 `en.wikipedia.org/wiki/C%2B%2B11` 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

New C++11/14 Features: Overview

Core language usability enhancements:

- Initializer lists vector<int> x{3,4,5}
- Uniform initialization X x{0,1};
- 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)
```

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)

```
vector<int> w..., v(std::move(w));  // (1) moves w into v
                                    // 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

// (6) compile time check
static_assert(sizeof(x) == 8, "wrong size")
```

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:

Examples

```
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 (see p.40)

```
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

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

```
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
```

Braced Initialization

There are three ways of initializing a variable:

```
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:

```
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:

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

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

```
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:

```
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 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:

```
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, like the function pointer above, it can make the code slightly more readable. However, the most compelling reason to use them is alias templates:

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

MyAllocList<Widget> lw;
```

Compare this to the equivalent `typedef` code:

```
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:

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

While with the alias template you can use it directly:

```
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:

```
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:

```
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:

```
enum Coord {X, Y};

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

Const 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:

```
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
```

`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):

```
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:

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

This function can only be used where a `constexpr` expression is needed if the arguments passed to it are `constexpr`

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:

```
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:

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

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)

Examples

```
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
  [=, this]{};   // error: this when = is the default
  [i, i] {};     // error: i repeated
}
```

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

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

```
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
```

Review: C++ Programming Tips

"Wisdom and beauty form a very rare combination" (Petronius Arbiter, Satyricon XCIV)

"With great power comes great responsibility" (Spiderman's Uncle)

Why C ?

- Code is FAST; compiler is FAST; often only little slower than hand-written assembly language code
- Lingua Franca of computing
- Portable. C compilers are available on all systems
- Compilers/interpreters for new languages are often written in C

Why C++ ?

- C + classes + templates: FAST code + coding CONVENIENCE + SAFETY
- You are still in total control, unlike Java or C#

From C to C++

Use `const` and `inline` instead of `#define`

- Macros are not type-safe
- Macros may have unwanted side effects. Use templates instead.

Prefer C++ library I/O over C library I/O

- C's `fprintf` and friends are unsafe and not extensible.
  Like the syntax `"%6.2f"`? Use `boost::format`
- C++ iostream class safe and extensible
- `iostream` speed has caught up, so speed is hardly a reason anymore for choosing C-library I/O

Prefer C++ style casts — easy to find with grep

Distinguish between pointers and references

References always point to existing objects, no arithmetic, safer

Memory Management

Use the same form in corresponding calls to new and delete

```
int *p = new Foo; ...  delete p;
int *p = new Foo[100]; ...  delete [] p;
```

For each `new` there must be at least one corresponding `delete`

Delete pointer members in destructors

Otherwise you are creating memory leaks

No need for checking the return value of `new`

It throws an exception if no memory available (in an ideal world)

`delete p` with `p = nullptr` is OK
(ignored, no `!= nullptr` check required)

Beware of double deletes ⇝ undefined behaviour

- Make sure all objects have sole owners
- For debugging consider adding `p = nullptr` after `delete p` or use template function:

```
template <typename T>
void destroy(T* &p)
{
  delete p;
#ifndef NDEBUG
  p = nullptr; // code created in debug mode
#endif
}

int *p = new int;
...
destroy(p);
*p = 0; // error caught in debug mode
```

Better yet: say good-bye to raw pointers, `new` and `delete`, and use C++11 smart pointers and `make_*` functions instead!

The "Big-4"

When designing new classes decide which operators you have to define: constructor, destructor, CC, AO

Things to consider: Do I want to risk undefined variables for gaining a little bit of speed for not initializing all components? Do I allocate resources like memory or file descriptors?

Define the CC and AO operators when resources are dynamically allocated

> Default component-wise copy is often insufficient in this case

Make destructors virtual in base classes

> Otherwise base class pointers can't call the right destructor

Have the AO return reference to `*this`

For iterated assignments `a = b = c ...`

Assign to all data members in the AO

Check for self-assignment in the AO

```
if (this == &rhs) { return *this; }
```

Operators for which you know that the default implementation the compiler provides is wrong and you don't want to implement need to be made inaccessible by using `= delete` (or by making them private)

C++11 adds move-semantics (see `p4u.cpp` in Lecture 18). If for your class `X` moving is faster than copying, implement the move-constructor and move-assignment `X(X &&)` and `X &X(X &&)` which bind to rvalue references. For more details, see books listed at the end or

https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers

Operators

Never overload `&` `&&` `||` `,`

Distinguish between prefix and postfix forms of `++` `--`

They (should) return different types:

`++i` : returns reference to `i`

`i++` : returns value of temporary object (can be slower!)

Be consistent

E.g., `++` `+=` `prefix++` `postfix++` should have related semantics

Class/Function Design (1)

Guard header files against multiple inclusion

`#ifndef ClassName_H ...` or `#pragma once`

Strive for complete and minimal public interfaces

- complete: users can do anything they need to do
- minimal: as few functions as possible, no overlaps

Minimize compilation dependencies between files

Consider forward declaration in conjunction with pointers/references to minimize file dependencies:

```
class Address;

class Person { ... Address *address; ... }
```

No need to `#include "Address.h"` in `Person.h`. Why?

Never use `using namespace X;` in header files

it forces users of your class to use the same namespace, even if they don't want to

Class/Function Design (2)

Avoid data members in public/protected interfaces

Use get/set functions – more flexible and safer

Use `const/constexpr` whenever possible

Pass and return objects by reference if you can

But don't return references to vanishing objects such as local variables!

Avoid returning writable "handles" to internal data from const member functions

Otherwise constant objects can be altered from the outside

Inheritance

Make sure public inheritance models “is a”

Never redefine an inherited non-virtual function

  Different results for `pBase->f()` and `pDeriv->f()`

Never redefine an inherited default parameter value

  Virtual functions are dynamically bound

  Default parameters are statically bound

Avoid casting down the inheritance hierarchy (base to derived class)

  Use virtual functions instead

Exceptions

Prefer exceptions over C-style error codes

Use destructors to prevent resource leaks

Say "good-bye" to pointers that manipulate local resources – use smart pointers instead

Prevent resource leaks in constructors

Destructors are only called for fully constructed objects

Prevent exceptions from leaving destructors

Exceptions within exceptions terminate program and unwinding exceptions calls destructors ...

Catch exceptions by reference

All alternatives create problems

Efficiency

Choose suitable data structures and efficient algorithms

Consider the empirical "80-20" rule:

- 80% of the resources are used by 20% of the code
- Focus your optimization efforts by using profilers (e.g. gprof)

Avoid frequent heap memory allocation, prefer stack variables

Know how to save space

bits, bytes, unions, home-brewed memory allocators

If necessary, optimize memory access patterns and data alignment to benefit from fast cache memory architectures

Understand costs of virtual functions, multiple inheritance, exception handling

Consider alternative libraries (e.g., `iostream` vs. `stdio`)

STL Tips (1)

Choose your containers wisely

- sequence vs. associative?
- tree-based vs. hash-based?
- speed vs. memory consumption?

Prefer C++ arrays over C-arrays. C++ arrays check for index violations and know their size

If speed matters, use C++ arrays, vectors, or hashed associative containers

Careful when storing pointers in containers

- if the container owns the objects they have to be destroyed before the container
- possible dangling pointers to vanished objects

Make sure comparison functors implement strict weak orderings

STL Tips (2)

Make sure destination ranges are big enough

Note which algorithms expect sorted ranges

Have realistic expectations about thread safety of STL containers: YOU need to lock containers

Call `empty()` instead of checking `size()` against 0. It may be faster.

Make element copies cheap and correct

STL copies elements often

More tips in Scott Meyers'

- Effective Modern C++ (C++11/14)
- Effective C++ (C++98/03 — but still relevant)
- More Effective C++
- Effective STL

C++ FIN