Coding Style

One man's constant is another man's variable.

—Alan Perlis

In this section, we tighten our focus from general design issues to issues that arise most often during actual coding.

The rules and guidelines in this section target coding practices that aren't specific to a particular language area (e.g., functions, classes, or namespaces) but that improve the quality of your code. Many of these idioms are about getting your compiler to help you, including the powerful tool of declarative **const** (Item 15) and internal **#include** guards (Item 24). Others will help you steer clear of land mines (including some outright undefined behavior) that your compiler can't always check for you, including avoiding macros (Item 16) and uninitialized variables (Item 19). All of them help to make your code more reliable.

Our vote for the most valuable Item in this section goes to Item 14: Prefer compile-and link-time errors to run-time errors.

14. Prefer compile- and link-time errors to run-time errors.

Summary

Don't put off 'til run time what you can do at build time: Prefer to write code that uses the compiler to check for invariants during compilation, instead of checking them at run time. Run-time checks are control- and data-dependent, which means you'll seldom know whether they are exhaustive. In contrast, compile-time checking is not control- or data-dependent and typically offers higher degrees of confidence.

Discussion

The C++ language offers many opportunities to "accelerate" error detection by pushing it to compilation time. Exploiting these static checking capabilities offers you many advantages, including the following:

- *Static checks are data- and flow-independent:* Static checking offers guarantees that are independent of the program inputs or execution flow. In contrast, to make sure that your run-time checking is strong enough, you need to test it for a rep resentative sample of all inputs. This is a daunting task for all but the most triv ial systems.
- *Statically expressed models are stronger:* Oftentimes, a program that relies less on run-time checks and more on compile-time checks reflects a better design be cause the model the program creates is properly expressed using C++'s type system. This way, you and the compiler are partners having a consistent view of the program's invariants; run-time checks are often a fallback to do checking that could be done statically but cannot be expressed precisely in the language. (See Item 68.)
- *Static checks don't incur run-time overhead:* With static checks replacing dynamic checks, the resulting executable will be faster without sacrificing correctness.

One of C++'s most powerful static checking tools is its static type checking. The debate on whether types should be checked statically (C++, Java, ML, Haskell) or dynamically (Smalltalk, Ruby, Python, Lisp) is open and lively. There is no clear winner in the general case, and there are languages and development styles that favor either kind of checking with reportedly good results. The static checking crowd argues that a large category of run-time error handling can be thus easily eliminated, resulting in stronger programs. On the other hand, the dynamic checking camp says that compilers can only check a fraction of potential bugs, so if you need to write unit tests anyway you might as well not bother with static checking at all and get a less restrictive programming environment.

One thing is clear: Within the context of the statically typed language C++, which provides strong static checking and little automatic run-time checking, programmers should definitely use the type system to their advantage wherever possible (see also Items 90 through 100). At the same time, run-time checks are sensible for data- and flow-dependent checking (e.g., array bounds checking or input data validation) (see Items 70 and 71).

Examples

There are several instances in which you can replace run-time checks with compile-time checks.

Example 1: Compile-time Boolean conditions. If you are testing for compile-time Boolean conditions such as **sizeof(int)** > = **8,** use static assertions instead of run-time tests. (But see also Item 91.)

Example 2: Compile-time polymorphism. Consider replacing run-time polymorphism (virtual functions) with compile-time polymorphism (templates) when defining generic functions or types. The latter yields code that is better checked statically. (See also Item 64.)

Example 3: Enums. Consider defining **enums** (or, better yet, full-fledged types) when you need to express symbolic constants or restricted integral values.

Example 4: Downcasting. If you frequently use **dynamic_cast** (or, worse, an unchecked **static_cast)** to perform downcasting, it can be a sign that your base classes offer too little functionality. Consider redesigning your interfaces so that your program can express computation in terms of the base class.

Exceptions

Some conditions cannot be checked at compile time and require run-time checks. For these, prefer to use assertions to detect internal programming errors (see Item 68) and follow the advice in the rest of the error handling section for other run-time errors such as data-dependent errors (see Items 69 through 75).

References

[AlexandrescuOl] §3 • [Boost] • [Meyers97] §46 • [StroustrnpOO] §2.4.2 • [SutterO2] §4 • [SutterO4] §2, §19

15. Use const proactively.

Summary

const is your friend: Immutable values are easier to understand, track, and reason about, so prefer constants over variables wherever it is sensible and make **const** your default choice when you define a value: It's safe, it's checked at compile time (see Item 14), and it's integrated with C++'s type system. Don't cast away **const** except to call a const-incorrect function (see Item 94).

Discussion

Constants simplify code because you only have to look at where the constant is defined to know its value everywhere. Consider this code:

```
void Fun( vector<int>& v)
 { //...
  const size_t len = v.size();
 //... 30 more lines ... }
```

When seeing **len's** definition above, you gain instant confidence about **len's** semantics throughout its scope (assuming the code doesn't cast away **const,** which it should not do; see below): It's a snapshot of v's length at a specific point. Just by looking up one line of code, you know len's semantics over its whole scope. Without the **const, len** might be later modified, either directly or through an alias. Best of all, the compiler will help you ensure that this truth remains true.

Note that **const** is not deep. For example, consider a class C that has a member of type X*. In C objects that are **const,** the **X*** member is also **const**—but the **X** object that is pointed to is not. (See [Saks99].)

Implement logical constness with **mutable** members. When a **const** member function of a class legitimately needs to modify a member variable (i.e., when the variable does not affect the object's observable state, such as cached data), declare that member variable **mutable.** Note that if all private members are hidden using the Pimpl idiom (see Item 43), **mutable** is not needed on either the cached information or the unchanging pointer to it.

Yes, **const** is "viral"—add it in one place, and it wants to propagate throughout your code as you call other functions whose signatures aren't yet const-correct. This is a feature, not a bug, and this quality greatly increases **const's** power even though it was unjustly demeaned in the days when **const** wasn't well understood and appreciated. Retrofitting an existing code base to make it const-correct takes effort, but it is worthwhile and likely to uncover latent bugs.

Const-correctness is worthwhile, proven, effective, and highly recommended. Understanding how and where a program's state changes is vital, and **const** documents that directly in code where the compiler can help to enforce it. Writing **const** appropriately helps you gain a better understanding of your design and makes your code sturdier and safer. If you find it impossible to make a member function **const,** you usually gain a better understanding of the ways in which that member function might modify an object's state. You might also understand which data members bridge the gap between physical constness and logical constness, as noted in the following Examples.

Never cast away **const** except to call a const-incorrect function, or in rare cases as a workaround for lack of **mutable** on older compilers.

Examples

Example: Avoid ***const*** *pass-by-value function parameters in function declarations.* The following two declarations are exactly equivalent:

```
void Fun( int x );

void Fun( const int x );        //redeclares the same function: top-level const is ignored
```

In the second declaration, the **const** is redundant. We recommend declaring functions without such top-level **consts,** so that readers of your header files won't get confused. However, the top-level **const** does make a difference in a function's *definition* and can be sensible there to catch unintended changes to the parameter:

```
void Fun( const int x ) {       //Fun's actual definition

  ++x;                          // error: cannot modify a const value

}
```

References

[Allison98] §10 • [Cline99] §14.02-12 • [DewhurstO3] §6, §31-32, §82 • [Keffer95] pp. 5-6 • [Koenig97] §4 • [Lakos96] §9.1.6, §9.1.12 • [Meyers97] §21 • [Murray93] §2.7 • [StroustrupOO] §7.2, §10.2.6, §16.3.1 • [SutterOO] §43

16. Avoid macros.

Summary

TO_PUT_IT_BLUNTLY: Macros are the bluntest instrument of C and C++'s abstraction facilities, ravenous wolves in functions' clothing, hard to tame, marching to their own beat all over your scopes. Avoid them.

Discussion

It's hard to find language that's colorful enough to describe macros, but we'll try. To quote from [SutterO4] §31:

> *Macros are obnoxious, smelly, sheet-hogging bedfellows for several reasons, most of which are related to the fact that they are a glorified text-substitution facility whose effects are applied during preprocessing, before any* C++ *syntax and semantic rules can even begin to apply.*

Lest there remain any ambiguity on this point, we note also that Bjarne Stroustrup has written:

> *I dislike most forms of preprocessors and macros. One ofC++'s aims is to make C's preprocessor redundant (§4.4, §18) because I consider its actions inherently error prone.* —[Stroustrup94] §3.3.1
>
> *Macros are almost never necessary in C++. Use* ***const*** *(§5.4) or* ***enum*** *(§4.8) to define manifest constants* **[see Item** 15], ***inline*** *(§7.1.1) to avoid function-calling overhead* **[but see Item 8],** ***templates*** *(Chapter 13) to specify families of functions and types* **[see Items** 64 **through** 67], *and* ***namespaces*** *(§8.2) to avoid name clashes* [see Items *57* through 59]. —[StroustrupOO] §1.6.1
>
> *The first rule about macros is: Don't use them unless you have to. Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer.* —[StroustrupOO] §7.8

The main problem with C++ macros is that they seem much better at what they do than they really are. Macros ignore scopes, ignore the type system, ignore all other language features and rules, and hijack the symbols they **#define** for the remainder of a file. Macro invocations look like symbols or function calls, but are neither. Macros are not "hygienic," meaning that they can expand to significantly and surprisingly different things depending on the context in which they are used. The text substitution that macros perform makes writing even remotely proper macros a black art whose mastery is as unrewarding as it is tedious.

People who think that template-related errors are the worst to decipher probably haven't seen those caused by badly formed or badly used macros. Templates are part of C++'s type system and thus allow compilers to get better at handling them (which they do), whereas macros are forever divorced from the language and hence intractable. Worse, unlike a template, a macro might expand to some transmission line noise that undesirably compiles by pure chance. Finally, an error in a macro can only be reported after the macro is expanded and not when it is defined.

Even in the rare cases where you do legitimately write a macro (see Exceptions), never ever even consider starting to think about writing a macro that is a common word or abbreviation. Do #undefine macros as soon as possible, always give them SCREAMING_UPPERCASE_AND_UGLY names, and avoid putting them in headers.

Examples

Example: Passing a template instantiation to a macro. Macros barely understand C's parentheses and square brackets well enough to balance them. C++, however, defines a new parenthetical construct, namely the < and > used in templates. Macros can't pair those correctly, which means that in a macro invocation

```
MACRO( Focxint, double> )
```

the macro thinks it is being passed two arguments, namely Focxint and double>, when in fact the construct is one C++ entity.

Exceptions

Macros remain the only solution for a few important tasks, such as #include guards (see Item 24), #ifdef and #if defined for conditional compilation, and implementing assert (see Item 68).

For conditional compilation (e.g., system-dependent parts), avoid littering your code with #ifdefs. Instead, prefer to organize code such that the use of macros drives alternative implementations of one common interface, and then use the interface throughout.

You may want to use macros (cautiously) when the alternative is extreme copying and pasting snippets of code around.

We note that both [C99] and [Boost] include moderate and radical extensions, respectively, to the preprocessor.

References

[Boost] • [C99] • [DewhurstO3] §25-28 • [Lakos96] §2.3.4 « [Stroustrup94] 53.3.2 *[StroustrupOO] §1.6.1, §7.8 • [SutterO2] §34-35 • [SutterO4] §31 [SutterO4a]*

17. Avoid magic numbers.

Summary

Programming isn't magic, so don't incant it: Avoid spelling literal constants like **42** or **3.14159** in code. They are not self-explanatory and complicate maintenance by adding a hard-to-detect form of duplication. Use symbolic names and expressions instead, such as **width * aspectRatio.**

Discussion

Names add information and introduce a single point of maintenance; raw numbers duplicated throughout a program are anonymous and a maintenance hassle. Constants should be enumerators or **const** values, scoped and named appropriately.

One 42 may not be the same as another 42. Worse, "in-head" computations made by the programmer (e.g., "this 84 comes from doubling the 42 used five lines ago") make it tedious and error-prone to later replace 42 with another constant.

Prefer replacing hardcoded strings with symbolic constants. Keeping strings separate from the code (e.g., in a dedicated **.cpp** or resource file) lets non-programmers review and update them, reduces duplication, and helps internationalization.

Examples

Example 1: Important domain-specific constants at namespace level,

```
const sizej PAGE_SIZE           = 8192,
            WORDS_PER_PAGE      = PAGE_SIZE/sizeof(int),
            INFO_BITS_PER_PAGE = 32 * CHAR_BIT;
```

Example 2: Class-specific constants. You can define static integral constants in the class definition; constants of other types need a separate definition or a short function.

```
//File widget.h
class Widget {
  static const int defaultWidth = 400;        // value provided in declaration
  static const double defaultPercent;          // value provided in definition
  static const char* Name() {return "Widget";}
};

//File widget.cpp
const double Widget::defaultPercent = 66.67; // value provided in definition
const int Widget::defaultWidth;              //definition required
```

References

[DewhurstO3] §2 • [Kernighan99] §1.5 • [StroustrupOO] §4.8, §5.4

18. Declare variables as locally as possible.

Summary

Avoid scope bloat, as with requirements so too with variables): Variables introduce state, and you should have to deal with as little state as possible, with lifetimes as short as possible. This is a specific case of Item 10 that deserves its own treatment.

Discussion

Variables whose lifetimes are longer than necessary have several drawbacks:

- *They make the program harder to understand and maintain:* For example, should code update the module-wide **path** string if it only changes the current drive?
- *They pollute their context with their name:* As a direct consequence of this, name-space-level variables, which are the most visible of all, are also the worst (see Item 10).
- *They can't always be sensibly initialized:* Never declare a variable before you can initialize it sensibly. Uninitialized variables are a pervasive source of bugs in all C and C++ programs, and they require our proactive attention because they can't always be detected by compilers (see Item 19).

In particular, older versions of C before [C99] required variables to be defined only at the beginning of a scope; this style is obsolete in C++. A serious problem with this restriction is that at the beginning of the scope you often don't yet have enough information to initialize variables with pertinent information. This leaves you with two choices—either initialize with some default blank value (e.g., zero), which is usually wasteful and can lead to errors if the variable ends up being used before it has a useful state, or leave them uninitialized, which is dangerous. An uninitialized variable of user-defined types will self-initialize to some blank value.

The cure is simple: Define each variable as locally as you can, which is usually exactly the point where you also have enough data to initialize it and immediately before its first use.

Exceptions

It can sometimes be beneficial to hoist a variable out of a loop. (See Item 9.)

Because constants don't add state, this Item does not apply to them. (See Item 17.)

References

[DewhurstO3] §3, §48, §66 [Dewhurst03] §95 [McConnell93] §5.1-4, §10.1 [StroustrupOO] §4.9.4, §6.3

19. Always initialize variables.

Summary

Start with a clean slate: Uninitialized variables are a common source of bugs in C and C++ programs. Avoid such bugs by being disciplined about cleaning memory before you use it; initialize variables upon definition.

Discussion

In the low-level efficiency tradition of C and C++ alike, the compiler is often not required to initialize variables unless you do it explicitly (e.g., local variables, forgotten members omitted from constructor initializer lists). Do it explicitly.

There are few reasons to ever leave a variable uninitialized. None is serious enough to justify the hazard of undefined behavior.

If you've used a procedural language (e.g., Pascal, C, Fortran, or Cobol) you might be used to defining variables in separation from the code that uses them, and then assigning them values later when they're about to be used. This approach is obsolete and not recommended (see Item 18).

A common misconception about uninitialized variables is that they will crash the program, so that those few uninitialized variables lying around here and there will be quickly revealed by simple testing. On the contrary, programs with uninitialized variables can run flawlessly for years if the bits in the memory happen to match the program's needs. Later, a call from a different context, a recompilation, or some change in another part of the program will cause failures ranging from inexplicable behavior to intermittent crashes.

Examples

Example 1: Using a default initial value or ?: to reduce mixing dataflow with control flow.

```
//Not recommended: Doesn't initialize variable
int speedupFactor;
if( condition)
  speedupFactor = 2;
else
  speedupFactor = -1;

//Better: Initializes variable
int speedupFactor = -1;
if( condition ) speedupFactor
= 2;
```

```
// Better: Initializes variable
int speedupFactor = condition ? 2 : -1;
```

The better alternatives nicely leave no gap between definition and initialization.

Example 2: Replacing a complicated computational flow with a function. Sometimes a value is computed in a way that is best encapsulated in a function (see Item 11):

```
//Not recommended: Doesn't initialize variable
int speedupFactor;

if( condition ) {
 //... code...
  speedupFactor = someValue; }
else {
 //... code...
  speedupFactor = someOtherValue; }
// Better: Initializes variable
int speedupFactor = ComputeSpeedupFactor();
```

Example 3: Initializing arrays. For large aggregate types such as arrays, proper initialization does not always mean having to really touch all the data. For example, say you use an API that forces you to use fixed arrays of **char** of size **MAX_PATH** (but see Items *77* and 78). If you are sure the arrays are always treated as null-terminated C strings, this immediate assignment is good enough:

```
//Acceptable: Create an empty path
char path[MAX_PATH]; path[0] = \0';
```

The following safer initialization fills all the characters in the array with zero:

```
//Better: Create a zero-filled path
char path[MAX_PATH] = {'\0' };
```

Both variants above are recommended, but in general you should prefer safety to unneeded efficiency.

Exceptions

Input buffers and **volatile** data that is directly written by hardware or other processes does not need to be initialized by the program.

References

[DewhurstO3] §48 • [StroustrupOO] §4.9.5, §6.3

20. Avoid long functions. Avoid deep nesting.

Summary

Short is better than long, flat is better than deep: Excessively long functions and nested code blocks are often caused by failing to give one function one cohesive responsibility (see Item 5), and both are usually solved by better refactoring.

Discussion

Every function should be a coherent unit of work bearing a suggestive name (see Item 5 and the Discussion in Item 70). When a function instead tries to merge such small conceptual elements inside a long function body, it ends up doing too much.

Excessive straight-line function length and excessive block nesting depth (e.g., **if, for, while,** and **try** blocks) are twin culprits that make functions more difficult to understand and maintain, and often needlessly so.

Each level of nesting adds intellectual overhead when reading code because you need to maintain a mental stack (e.g., enter conditional, enter loop, enter **try,** enter conditional, ...). Have you ever found a closing brace in someone's code and wondered which of the many fors, whiles, or ifs it matched? Prefer better functional decomposition to help avoid forcing readers to keep as much context in mind at a time.

Exercise common sense and reasonableness: Limit the length and depth of your functions. All of the following good advice also helps reduce length and nesting:

- *Prefer cohesion:* Give one function one responsibility (see Item 5).
- *Don't repeat yourself:* Prefer a named function over repeated similar code snippets.
- *Prefer &&:* Avoid nested consecutive ifs where an && condition will do.
- *Don't **try** too hard:* Prefer automatic cleanup via destructors over **try** blocks (see Item 13).
- *Prefer algorithms:* They're flatter than loops, and often better (see Item 84).
- *Don't **switch** on type tags.* Prefer polymorphic functions (see Item 90).

Exceptions

A function might be legitimately long and/or deep when its functionality can't be reasonably refactored into independent subtasks because every potential refactoring would require passing many local variables and context (rendering the result less readable rather than more readable). But if several such potential functions take similar arguments, they might be candidates for becoming members of a new class.

References

[Piwowarski82] • *[Miller56]*

21. Avoid initialization dependencies across compilation units.

Summary

Keep (initialization) order: Namespace-level objects in different compilation units should never depend on each other for initialization, because their initialization order is undefined. Doing otherwise causes headaches ranging from mysterious crashes when you make small changes in your project to severe non-portability even to new releases of the same compiler.

Discussion

When you define two namespace-level objects in different compilation units, which object's constructor is called first is not defined. Often (but not always) your tools might happen to initialize them in the order in which the compilation units' object files are linked, but this assumption is usually not reliable; even when it does hold, you don't want the correctness of your code to subtly depend on your makefile or project file. (For more on the evils of order dependencies, see also Item 59.)

Therefore, inside the initialization code of any namespace-level object, you can't assume that any other object defined in a different compilation unit has already been initialized. These considerations apply to dynamically initialized variables of primitive types, such as a namespace-level **bool reg_success = LibRegister("mylib");**

Note that, even before they are ever constructed using a constructor, namespace-level objects are statically initialized with all zeroes (as opposed to, say, automatic objects that initially contain garbage). Paradoxically, this zero-initialization can make bugs harder to detect, because instead of crashing your program swiftly the static zero-initialization gives your yet-uninitialized object an appearance of legitimacy. You'd think that that string is empty, that pointer is null, and that integer is zero, when in fact no code of yours has bothered to initialize them yet.

To avoid this problem, avoid namespace-level variables wherever possible; they are dangerous (see Item 10). When you do need such a variable that might depend upon another, consider the Singleton design pattern; used carefully, it might avoid implicit dependencies by ensuring that an object is initialized upon first access. Still, Singleton is a global variable in sheep's clothing (see again Item 10), and is broken by mutual or cyclic dependencies (again, zero-initialization only adds to the confusion).

References

[DewhurstO3] §55 • [Gamma95] • [McConnell93] §5.1-4 • [StronstrupOO] §9.4.1, §10.4.9

22. Minimize definitional dependencies. Avoid cyclic dependencies.

Summary

Don't be over-dependent: Don't **#include** a definition when a forward declaration will do.

Don't be co-dependent: Cyclic dependencies occur when two modules depend directly or indirectly on one another. A module is a cohesive unit of release (see page 103); modules that are interdependent are not really individual modules, but super-glued together into what's really a larger module, a larger unit of release. Thus, cyclic dependencies work against modularity and are a bane of large projects. Avoid them.

Discussion

Prefer forward declarations except where you really need a type's definition. You need a full definition of a class C in two main cases:

- *When you need to know the size of a C object:* For example, when allocating a C on the stack or as a directly-held member of another type.
- *When you need to name or call a member of C:* For example, when calling a member function.

In keeping with this book's charter, we'll set aside from the start those cyclic dependencies that cause compile-time errors; you've already fixed them by following good advice present in the literature and Item 1. Let's focus on cyclic dependencies that remain in compilable code, see how they trouble your code's quality, and what steps need be taken to avoid them.

In general, dependencies and their cycles should be thought of at module level. A module is a cohesive collection of classes and functions released together (see Item 5 and page 103). In its simplest form, a cyclic dependency has two classes that directly depend upon each other:

```
class Child;                 //breaks the dependency cycle

class Parent {//...
Child* myChild_; };

class Child {//...           //possibly in a different header
  Parent*
myParent_; };
```

Parent and **Child** depend upon each other. The code compiles, but we've set the stage for a fundamental problem: The two classes are not independent anymore, but have become interdependent. That is not necessarily bad, but it should only occur when both are part of the same module (developed by the same person or team and tested and released as a whole).

In contrast, consider: What if **Child** did not need to store a back link to its **Parent** ob-ject? Then **Child** could be released as its own separate, smaller module (and maybe under a different name) in total independence from **Parent**—clearly a more flexible design.

Things get only worse when dependency cycles span multiple modules, which are all stuck together with dependency glue to form a single monolithic unit of release. That's why cycles are the fiercest enemy of modularity.

To break cycles, apply the Dependency Inversion Principle documented in [Mar-tin96a] and [Martin00] (see also Item 36): Don't make high-level modules depend on low-level modules; instead, make both depend on abstractions. If you can define in-dependent abstract classes for either **Parent** or **Child,** you've broken the cycle. Oth-erwise, you must commit to making them parts of the same module.

A particular form of dependency that certain designs suffer from is transitive de-pendency on derived classes, which occurs when a base class depends on all of its descendants, direct and indirect. Some implementations of the Visitor design pattern leads to this kind of dependency. Such a dependency is acceptable only for excep-tionally stable hierarchies. Otherwise, you may want to change your design; for ex-ample, use the Acyclic Visitor pattern [Martin98].

One symptom of excessive interdependencies is incremental builds that have to build large parts of the project in response to local changes. (See Item 2.)

Exceptions

Cycles among classes are not necessarily bad—as long as the classes are considered part of the same module, tested together, and released together. Naive implementa-tions of such patterns as Command and Visitor result in interfaces that are naturally interdependent. These interdependencies can be broken, but doing so requires ex-plicit design.

References

[AlexandrescuOl] §3 • [Boost] • [Gamma95] • [Lakos96] §0.2.1, §4.6-14, §5 • [Martin96a] • [Martin96b] • [Martin98] §7 • [MartinOO] • [McConnell93] §5 • [Meyers97] §46 • [StroustrupOO] §24.3.5 • [SutterOO] §26 • [SutterO2] §37 • [SutterO3]

23. Make header files self-sufficient.

Summary

Behave responsibly: Ensure that each header you write is compilable standalone, by having it include any headers its contents depend upon.

Discussion

If one header file won't work unless the file that includes it also includes another header, that's gauche and puts unnecessary burden on that header file's users.

Years ago, some experts advised that headers should not include other headers because of the cost of opening and parsing a guarded header multiple times. Fortunately, this is largely obsolete: Many modern C++ compilers recognize header guards automatically (see Item 24) and don't even open the same header twice. Some also offer precompiled headers, which help to ensure that often-used, seldom-changed headers will not be parsed often.

But don't include headers that you ***don't*** need; they just create stray dependencies.

Consider this technique to help enforce header self-sufficiency: In your build, compile each header in isolation and validate that there are no errors or warnings.

Examples

Some subtler issues arise in connection with templates.

Example 1: Dependent names. Templates are compiled at the point where they are defined, except that any dependent names or types are not compiled until the point where the template is instantiated. This means that a **template<class T> class Widget** with a **std::deque<T>** member does not incur a compile-time error even when **<deque>** is not included, as long as nobody instantiates **Widget.** Given that **Widget** exists in order to be instantiated, its header clearly should **#include <deque>.**

Example 2: Member function templates, and member functions of templates, are instantiated only if used. Suppose that **Widget** doesn't have a member of type **std::deque<T>,** but **Widget's Transmogrify** member function uses a **deque.** Then **Widget's** callers can instantiate and use **Widget** just fine even if no one includes **<deque>,** as long as they don't use **Transmogrify.** By default, the **Widget** header should still **#include <deque>** because it is necessary for at least some callers of **Widget.** In rare cases where an expensive header is being included for few rarely used functions of a template, consider refactoring those functions as nonmembers supplied in a separate header that does include the expensive one. (See Item 44.)

References

[Eakos96] §3.2 • [StroustrupOO] §9.2.3 • [SutterOO] §26-30 • [Vandevoorde03] §9-10

24. Always write internal #include guards. Never write external #include guards.

Summary

Wear head(er) protection: Prevent unintended multiple inclusions by using **#include** guards with unique names for all of your header files.

Discussion

Each header file should be guarded by an internal **#include** guard to avoid redefinitions in case it is included multiple times. For example, a header file **foo.h** should follow the general form:

```
#ifndef FOO_H_INCLUDED_
#define FOO_HJNCLUDED_ //...
contents of the file... #endif
```

Observe the following rules when defining include guards:

- *Use a unique guard name:* Make sure it is unique at least within your application. We used a popular convention above; the guard name can include the applica
tion name, and some tools generate guard names containing random numbers.
- *Don't try to be clever:* Don't put any code or comments before and after the guarded portion, and stick to the standard form as shown. Today's preproces
sors can detect include guards, but they might have limited intelligence and ex
pect the guard code to appear exactly at the beginning and end of the header.

Avoid using the obsolete external include guards advocated in older books:

```
#ifndef FOO_HJNCLUDED_            //NOT recommended
#include "foo.h"
#define FOO_HJNCLUDED_
#endif
```

External include guards are tedious, are obsolete on today's compilers, and are fragile with tight coupling because the callers and header must agree on the guard name.

Exceptions

In very rare cases, a header file may be intended to be included multiple times.

References

[C++03, §2.1] • [StroustrupOO] §9.3.3