cache, and — for larger programs — might not even be paged into the computer's **physical memory** (RAM). Regardless of the exception-handling implementation used, throwing an exception is presumed to be *exceptional* (atypical) — i.e., all of its handling properly resides on the cold path.

In a **try**/**catch** construct, the **try** block is *always* on the hot path of that construct, whereas the **catch** block is invariably on the cold path. Even in the absence of an explicit **try** in the source code, the compiler must, in certain cases, generate an implicit **try**/**catch** with a hot path when no exception is thrown and a cold path for **stack unwinding** — i.e., destroying live, local, non-trivially destructible automatic variables, which might include a rethrow.

Note that a single function might require multiple implicit **try** blocks — one for each cluster of such local variables that is followed by a potentially throwing expression; see *Use Cases — Reducing object-code size* on page 1101.

Contemporary compilers make an effort to segregate, within the **executable image**, what they believe to be seldomly executed sequences of machine instructions and might sequester them into a distant region of the virtual address space. The path where an exception is thrown, however, is known with 100% accuracy to be on the cold path, and modern compilers will strive to avoid interleaving such cold-path code alongside frequently executed instructions.

Older exception-handling implementations Prior to the ubiquitous adoption of the zero-cost exception model (described below), most compilers would perform stack-based bookkeeping to dynamically track the currently active nested **try** blocks at any point in the execution of the function.

For example, one implementation[36] would create a new exception-registration record for each **try** block, including those generated implicitly. Each registration record would hold two pointers: one to the exception-handler code and the other to the previous exception registration record. On entry to each **try** block (implicit or otherwise), the compiler would insert instructions to construct the exception registration record, add it to the linked list of active **try** blocks, and update a thread-local pointer to the new exception registration record. If an exception were to be thrown, the exception-handling logic would walk the linked list of registered blocks, calling each handler in turn as it unwound the stack. The compiler would also insert code at the end of the **try** block to handle the normal (i.e., exception-free) case of restoring the thread-local pointer to the previous registration record, thereby removing the block from the linked list and restoring the previous dynamic exception handling state.

In these old models, registration and deregistration instructions were executed on the hot path on, respectively, entry to and exit from each **try** block — even when no exception was thrown. The runtime burden of exception support strongly favored the normal, nonexception case, but the runtime cost was nonzero. These instructions added overhead on the hot path that many considered onerous enough to choose to forgo using exceptions entirely, building their programs with exceptions disabled (e.g., using a compiler switch).

[36]Compilers for 32-bit Windows platforms continue to use this non-zero-cost exception-handling model; see **pietrek97**.