C++ ビルドの高速化

完全ガイド

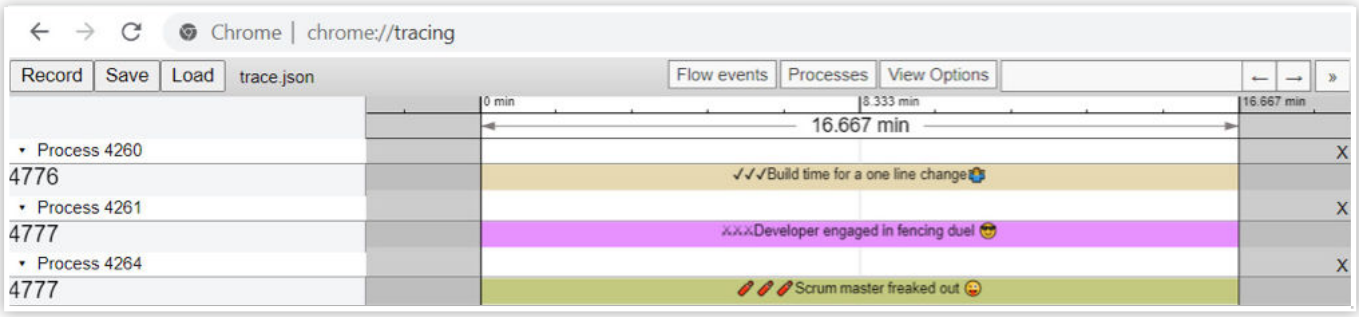
INCREDIBUILD

目次

C++ ビルドの問題点

C++ は優れたプログラミング言語で多くの人に愛されていますが、そのビルド時間には現実的な問題が横たわっています。あなたは C++ で書かれたソースコードをビルドしていますか? ビルドが遅すぎてあなたやあなたの上司、そして会社全体の課題になっていませんか? これは今に始まった問題ではありません。特に継続的インテグレーション / 継続的デプロイメント (CI/CD) が主流のアジャイル ワーキングな環境では深刻な問題です。

敢えてこの問題を無視するという選択肢もあります。また (これをネタにした『XKCD』の風刺画のように) 長いビルド時間を利用してレジャーを楽しむのもいいかもしれません。とはいえ、一方でこれを解決すべき問題として真剣に取り組んでいる人々もいます (当社がどちらに当てはまるかはご想像にお任せします……)。



(上の画像は仮想のプロジェクトのものです。どこかで — 例えばあなたの現在のプロジェクトで — 見たような気がするとしたらそれは全くの偶然です)

なぜ C++ ビルドはこんなにも時間がかかるのか?

これは多くの C++ 開発者を悩ませている問いですが、いくつかの理由が考えられます。

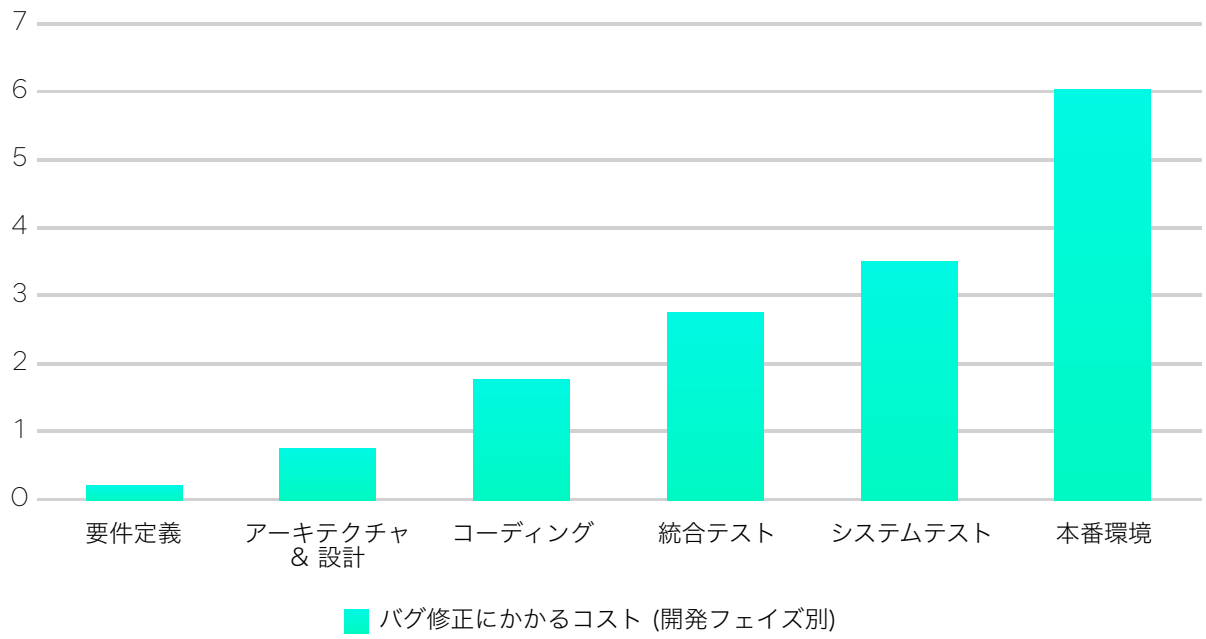
- リソース不足のマシンでビルドが実行されている
- ビルドの依存関係が多すぎる
- ビルドで古いコンパイラやリンカが使用されている
- ビルドでプリコンパイル済みヘッダー ファイルが使われていない / 誤用されている
- コンパイラが最適化を要求されている
- コードベースが整備されていない / 余計なコードが追加されている
- コードベースの規模が大きすぎる

解析が必要なヘッダー ファイルが多いことも理由の一つですが、コンパイル時のデータ処理が複雑で動的なことがビルド時

長いビルドの何が問題なのか?

回答に詰まるかもしれませんが、当たり前といえば当たり前です。C++ 財団が最近行った調査によると、ビルド時間が「問題である」と回答した開発者が 80% を超えた (「大きな問題である」と「ある程度問題である」がそれぞれ 40% 強) 一方で、「全く問題ではない」と回答した開発者は 17% に留まりました。ビルドの終了を待ったり、開発プロセスに優先順位付け (ウィークリー ビルド) をするのはともかく、コンパイルを避けるためにテストを後回しにしたり、タスクをスキップしたりすることは、組織に壊滅的なダメージを与える可能性があります。

長いビルド時間は納期に影響しますが、それを回避するとソフトウェアの根本的な品質にかかわります。



上記はバグを発見した時点を基準として修正にかかる相対的なコストを表したグラフです。ビルド時間が押してテストが後回しにされた結果、本番環境までバグに気付かず開発が進んでしまうことはよくあります。これにより全体の開発コストが高くなり、ソフトウェアの品質も犠牲になります。競争の激しい現在のマーケットでは高品質かつ頻繁なリリースが不可欠になっているにもかかわらずです。

最近ではシフトレフトへの移行が勢いを増しています。これは開発サイクルの中でテストの工程を前倒しして実施する手法です。リポジトリにコードをコミットする前にテストを実行することでソフトウェアの外部品質を向上させることができます。静的コード解析も並行して行えばソフトウェアの内部品質も改善できます。

長いビルド時間は開発者の生産性だけでなく、その思考回路にも悪影響を及ぼします。これはビルドからのフィードバックに時間がかかることが原因です。

つまり、ビルド時間が長くなると直接的なコストと間接的なコストの両方が発生するということです。

ビルド時間を短縮するためにできること

このガイドでは C++ のコンパイル時間を短縮するためのさまざまな方法を紹介します。コンパイルの高速化ソリューションを提供する当社がなぜこのような情報を提供するのか? それはビルド時間を短縮することに真剣に取り組んでいるからにほかなりません。

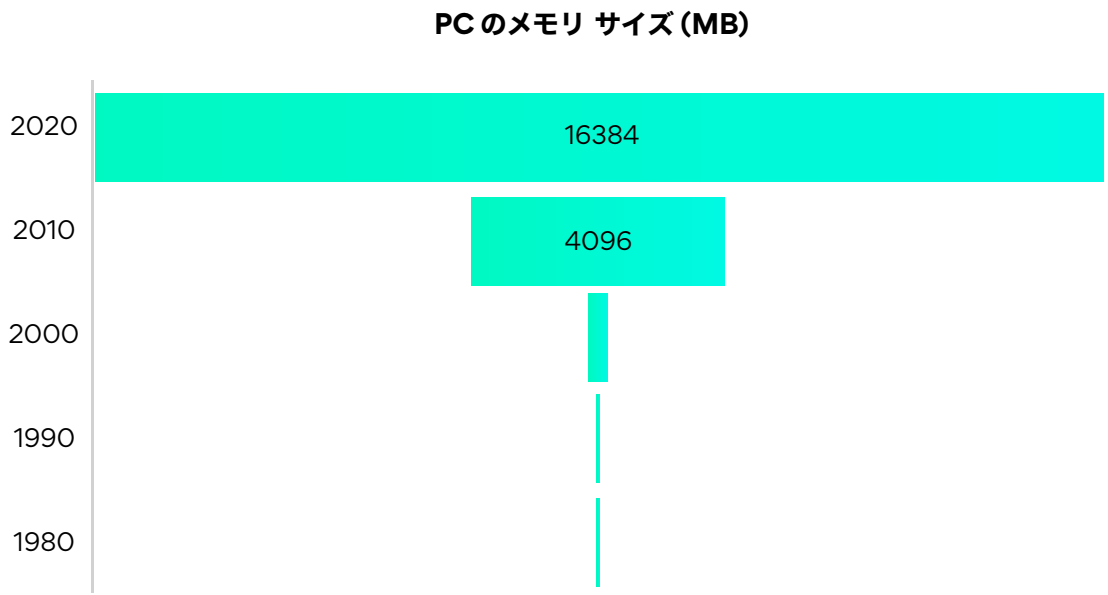
コンパイル時間を改善するためにできることはたくさんあります。実際、ここで紹介する対策は概ね有効です。場合によってはメリットを得るために多少の労力が必要になるかもしれませんが、それに見合うだけの価値が得られるはずです。Incredibuild でビルドを高速化するのはそれとはまったく別の話です。

それではさっそく見ていきましょう。

高性能なビルドマシンの導入

これは当然かもしれません。高性能なハードウェアへの投資は問題解決への近道であり、C++ のビルド時間を短縮する方法として最初に頭に浮かぶものでしょう。その直感は間違っていません。大容量のメモリやハードディスク、性能の高い CPU を使うことでビルド時間は間違いなく改善します。

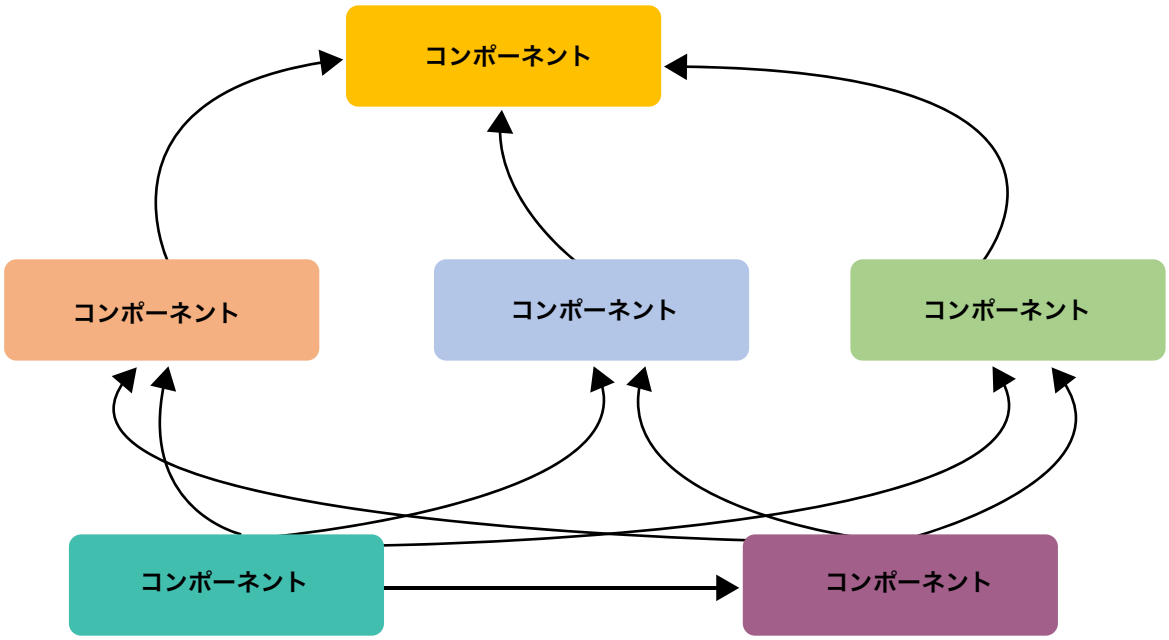
PC のメモリにおけるムーアの法則:



依存関係の削減

ファイル、コンポーネント、モジュール、レイヤー間の密結合はビルド時間を長引かせる要因の一つです。下図のようなプロジェクト設計 (ボックス内をファイル、モジュール、レイヤーなどに変えても構いません) が問題なのはお分かりですよね?

これらの依存関係はすべて必要ですか? 分離できそうなものはありませんか? どの依存関係を分離するのが一番面倒で、どれが最もリスキーですか? コストと効果を天秤にかけてプロジェクトの設計を見直しましょう。内部コードの品質が改善すればコンパイル時間も短くなります。これは C++ ベースのプロジェクトだけではく、あらゆる言語に当てはまります。



密結合のシステムはいつでも好ましくない

Lyrics.h

```
#pragma once
namespace SongLibrary
{
	class	Lyrics
	{
}		// The code for Lyrics is complicated...
	};
```

Song.h

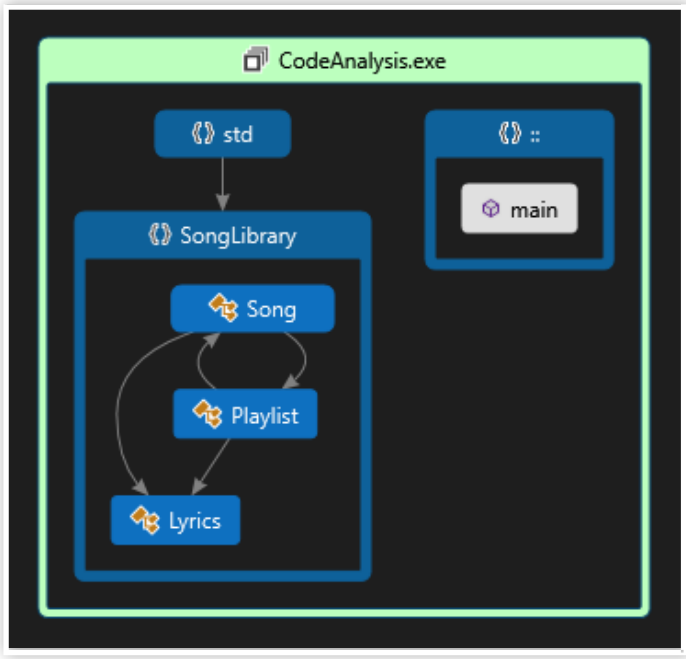
```
#pragma once
#include    <memory>
#include    <string>
#include    <vector>

namespace SongLibrary
{
        class    Lyrics   ;
         class   Playlist    ;
         class   Song
        {
         private   :
                  std:: wstring   m_Writer;
                  std:: wstring   m_CoAuthor;
                  std:: shared_ptr   < Lyrics > m_Lyrics;
                  int  m_YearOfRelease;
         public   :
                  int  GetYearOfRelease()    const  {   return   m_YearOfRelease; }
                  std:: wstring   GetWriter()   const  {   return   m_Writer; }
                  std:: wstring   GetCoAuthor()  const  {   return   m_CoAuthor; }
                  // Constructor for clients still using classic C++
                  Song(std:: wstring   writer, std:: wstring   coauthor,   const
Lyrics*    const  lyrics,  int   yearofrelease);
                  // Constructor for clients of modern C++
                  Song(std:: wstring  writer , std:: wstring   coaauthor  ,
std::   shared_ptr   <Lyrics  >  lyrics   ,  int   yearofrelease    );
                  // Design Decision. Copy and Move allowed. No assignment
                  Song(const Song&) = default;
                  Song&  operator   =( const   Song&) =   delete   ;
                  Song( Song&&) =   default   ;
                  // Design Creep. Unfortunately we have a requirement
                  std:: vector  <std::    weak_ptr  <   Playlist   >> m_Playlists;
         };
}
```

Playlist.h

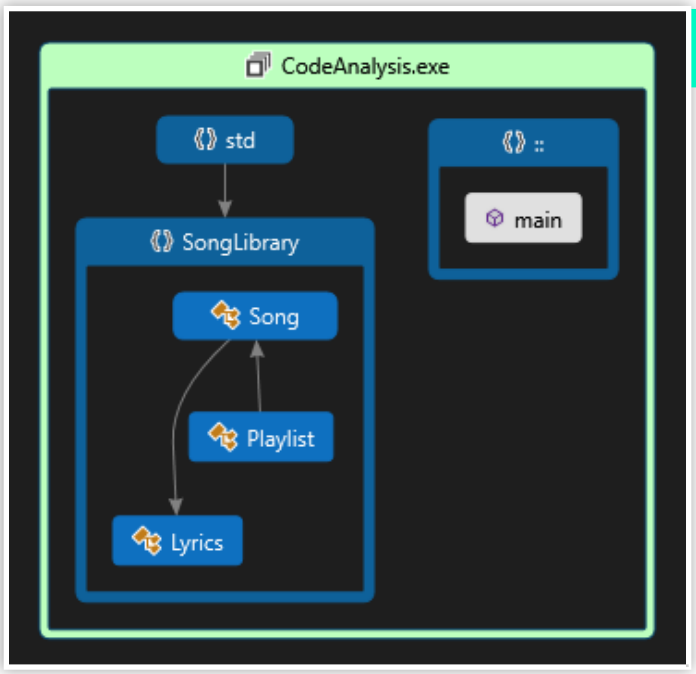
```
#pragma once
#include    "Song.h"
#include    "Lyrics.h"
#include    <vector>
#include    <chrono>

namespace SongLibrary
{
        class     Playlist
         {
        private:
                    std:: vector  <Song> m_Songs;
                    std::chrono::duration<int> m_TotalPlayingTime;
                    std:: shared_ptr  <  Lyrics  > m_LyricsOfCurrentSong;
         public:
                    std::   wstring   getPlayingTime();
                    bool  RemoveSongFromPlaylist( Song  toberemoved );
                    bool  AddSongToPlayList(Song  tobeadded );
         };
}
```

このシステムはどのように見えるでしょうか?

悪い設計



上の例では Song と Playlist の間に循環依存の関係があります。これを修正したのが下のものです。



修正した設計

システムの設計を見直すことでビルド時間を根本的に改善することができます（ボトルネックや依存関係を簡単に分析するためのビルドモニターや可視化ツールについてはこちらをご覧ください）。また、設計が改善されれば、次のセクションで説明している新たなビルド時間の改善方法への道も開かれるでしょう。

静的リンクと動的リンク

この手法はプラットフォームに依存するかもしれませんが、最近はどの OS も動的リンクに対応しています。

- Dynamic Linked Libraries (dll、Windows)
- Dynamically Loaded Module (dylib、Mac)
- Dynamically Loaded Libraries (dl、Linux)

めったに変更されないユーティリティを動的ライブラリに入れておけば、ビルドごとにコードがコンパイルされるわけではないので大幅にコンパイル時間を短縮できます。

ただし、このような変更を繰り返し行うにはリファクタリングやリエンジニアリングが必要になります。また、そのような動的コードのバージョンを管理するのにもコストがかかります。DLL のインターフェイスが変更するのに合わせて新しいバージョンを割り当てることをお勧めしています。次のような階層的なバージョン管理が一般的です。

メジャー バージョン	マイナー バージョン	ビルド番号	リビジョン

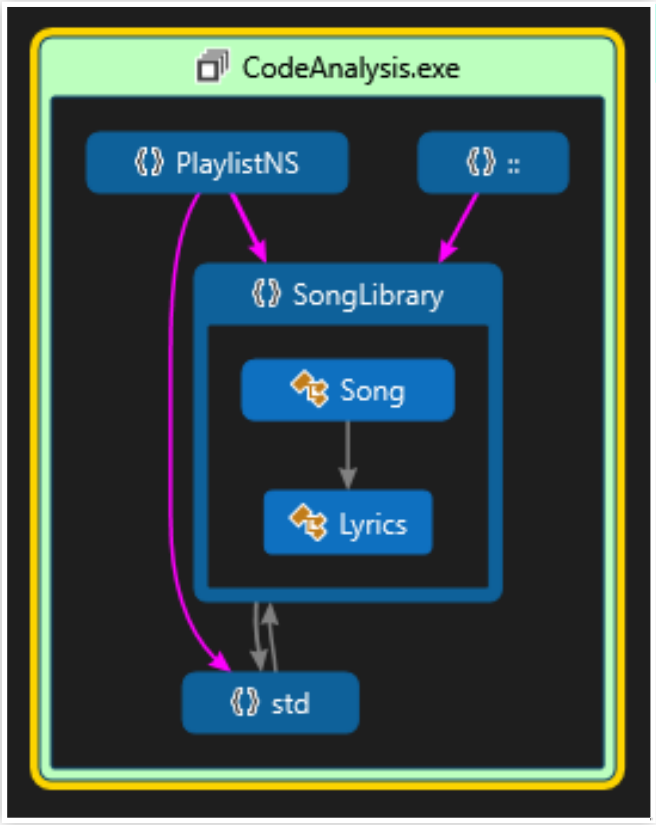
バージョンが違えば DLL のメジャー バージョン番号は異なります。バージョンアップに伴うメンテナンスにはコストがかかりますが、ビルドの観点からは静的リンクよりも DLL に軍配が上がります。

前のセクションでは例を紹介して設計を見直しましたが、ここではさらに一歩踏み込んでみます。まずは Playlist を SongLibrary の名前空間から分離します。

Playlist.h (変更後)

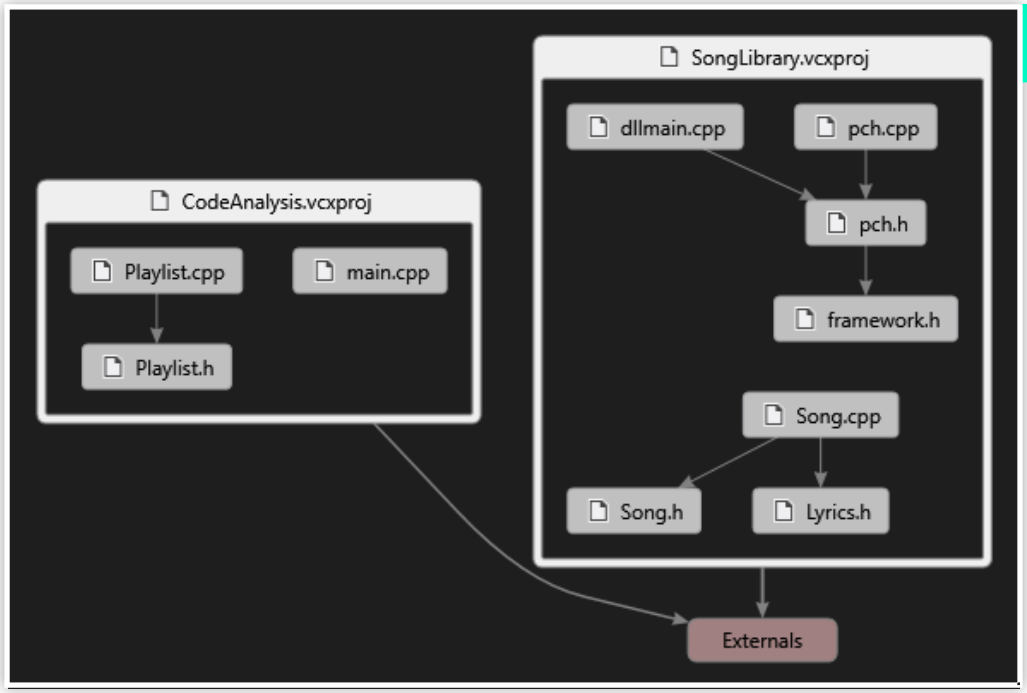
```
#pragma once
#include    "Song.h"
#include    "Lyrics.h"
#include    <vector>
#include    <chrono>

namespace PlaylistNS
{
        class   Playlist
        {
        private:
                std:: vector  <SongLibrary:: Song> m_Songs;
                std::chrono:: duration   <int  > m_TotalPlayingTime;
                std:: shared_ptr  <SongLibrary::  Lyrics >
m_LyricsOfCurrentSong;
      public:
                std::wstring getPlayingTime();
                bool  RemoveSongFromPlaylist(SongLibrary:: Song
toberemoved );
                bool  AddSongToPlayList(SongLibrary::Song  tobeadded );
                Playlist();
      };
}
```

かなりすっきりしました。



クリアになった
コンポーネント

これで SongLibrary 内のすべてを独立したライブラリに分離できることが分かりました。SongLibrary には特に変更がないようなので動的ライブラリに分離します。



DLL への
モジュール化

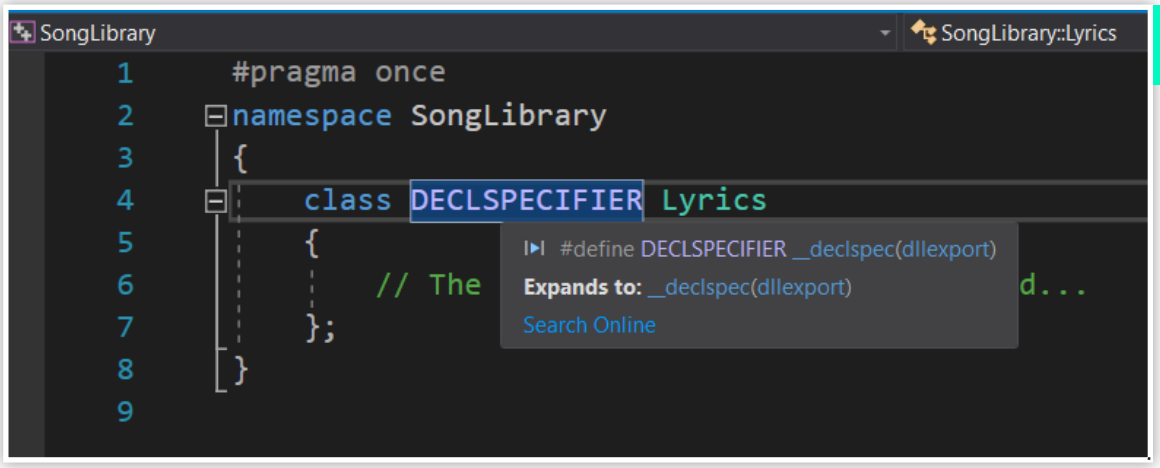
pch.h / pch.cpp ファイルについては後ほど解説する予定なので、ここでは framework.h ファイルに注目します。

Framework.h

```
#pragma once

#ifdef    SONGLIBRARY_EXPORTS
#define   DECLSPECIFIER __declspec ( dllexport )
#define   EXPIMP_TEMPLATE
#else
#define   DECLSPECIFIER __declspec ( dllimport )
#define   EXPIMP_TEMPLATE  extern
#endif
```

これは、Windows で DLL から関数をエクスポートするためのメカニズムです。別に DLL を作成したことでほかのファイルがどう変わったか見てみましょう。

Lyrics.h
(変更後)



ご覧の通り Lyrics というクラスは DLL からエクスポートされます。Song.h も同じような変更が必要ですが、警告も表示されています。

重大度	コード	詳細	プロジェクト	ファイル	ライン
Warning	C4251	SongLibrary::Song::m_Writer': class 'std::basic_string<wchar_t,std::char_traits<wch ar_t>,std::allocator<wchar_t>> は 'SongLibrary::Song' のクラスが使用するために dll-interface が必要です	CodeAnalysis	D:\Demo\Song Library\Song.h	13

```
#include    <vector>
#include    "framework.h"

namespace SongLibrary
{
        class   Lyrics;

EXPIMP_TEMPLATE template class  DECLSPECIFIER
std:: shared_ptr <  Lyrics >;

        class  DECLSPECIFIER   Song
        {
        private   :
                        ...
```

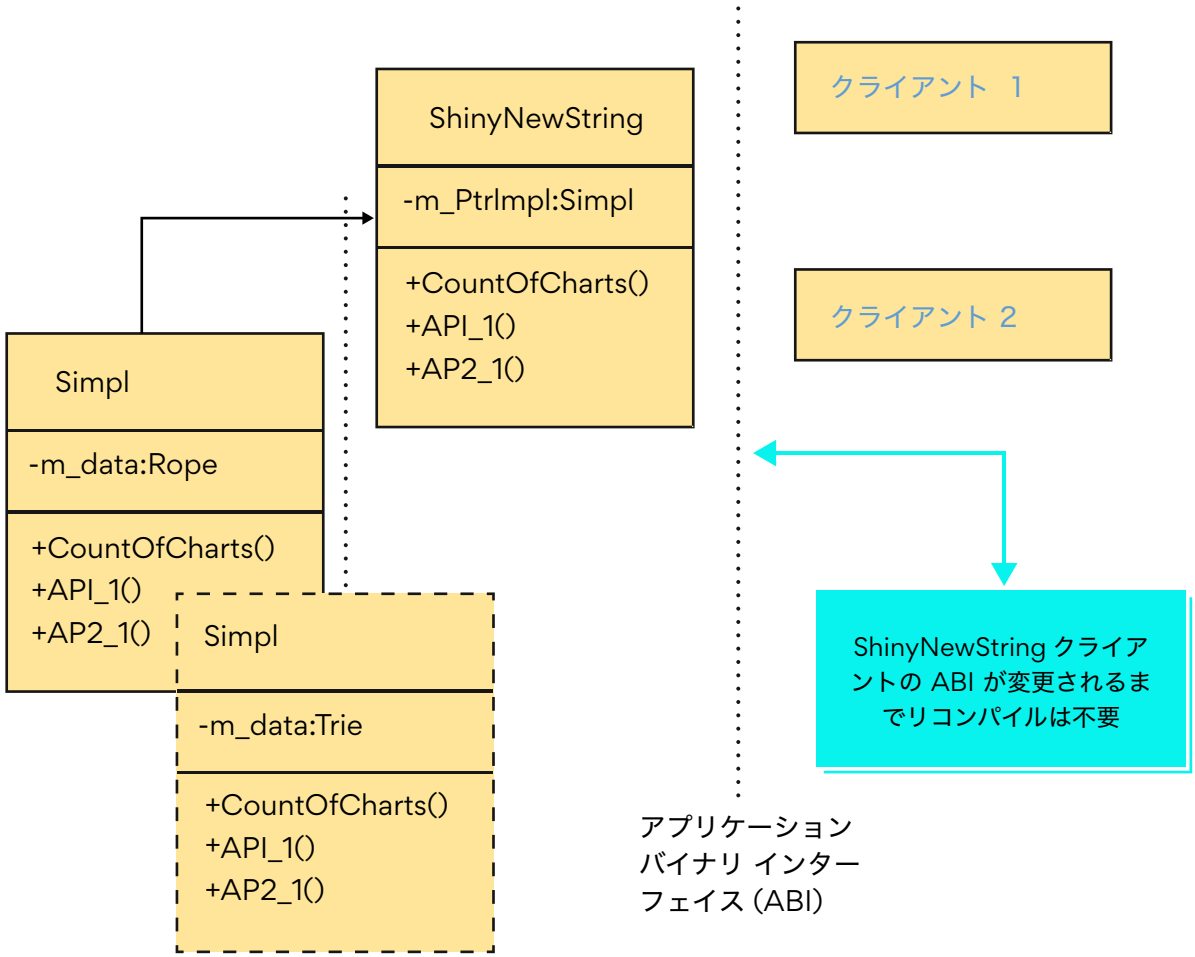
修正が必要な警告

ご覧の通り std::shared_ptr<Lyrics> クラスのすべてのメンバを生成しなければならないようです。これは DLL のインターフェイスに STL クラスが使われているからです。ではどうすればよいのか? その答えは Pimpl です。

Pimpl イディオムとそのメリット

Pimpl はクラス間の依存関係を減らし、C++ ベースのプロジェクトのビルド時間を減らすのに使われます。実装を非表示にしてコンパイラから隠すため「コンパイラ ファイアウォール」とも呼ばれます。実装は変更されることがありますが、クラスを使うクライアントのインターフェイスはそのままなのでリコンパイルは不要です。これによりパフォーマンスが大幅に向上します。

設計が必要な真新しい文字列クラスを例に見ていきましょう。



Pimpl 不透明なポインタを介してアクセスが行われるため、内部実装の変更はクラスの外部クライアントには表示されません。これが「コンパイル ファイアウォール」と呼ばれる所以です。

もちろんトレードオフはつきもので、Pimpl の場合それはパフォーマンスです。クラスのメンバ関数を実行する際にその下にある実装クラスにデリゲートされるため、間接参照のオーバーヘッドが増えます。また、コードが少し複雑になり、テスト容易性も低下します。とはいえ、C++ ベースのプロジェクトでビルド時間を短縮するための方法としては悪くない選択肢です。

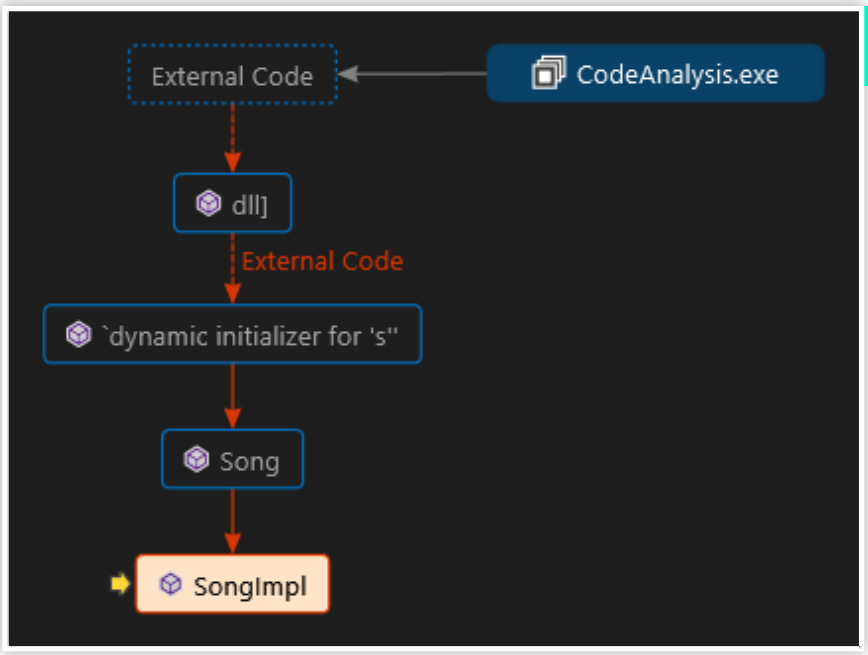
例に戻って、Pimpl イディオムが設計を修正するためにどのように使われているかを見てみましょう。

このインターフェイスを

Song.h (With STL)

```
class   DECLSPECIFIER   Song
{
        private:
                std:: wstring  m_Writer;
                std:: wstring  m_CoAuthor;
                std:: shared_ptr <  Lyrics  > m_Lyrics;
                int m_YearOfRelease;
public:
...
```

次のように設計し直します。



Song クラスの Pimpl 設計

コードは次のようになります。

Song.h (With pImpl)

```
#pragma once
#include <memory>
#include <string>
#include <vector>
#include "framework.h"

namespace SongLibrary
{
	class Lyrics;
	class SongImpl;
	class DECLSPECIFIER Song
	{
	private:
		SongImpl * m_songImpl;
	public:
		int GetYearOfRelease() const;
		std::wstring GetWriter() const;
		std::wstring GetCoAuthor() const;
		// Constructor for clients still using classic C++
		Song(std::wstring writer, std::wstring coauthor, const
Lyrics * const lyrics, int yearofrelease);
		// Constructor for clients of modern C++
		Song(std::wstring writer, std::wstring coauthor,
std::shared_ptr<Lyrics> lyrics, int yearofrelease);
		// Design Decision. Copy and move allowed. No assignment.
		Song(const Song&); // Can no longer be default. Why?
		Song& operator=(const Song&) = delete;
		Song(Song&&); // Can no longer be default. Why?
		~Song(); // Naked pointer member needs a destructor.
	};
}
```

これはデモ用です。プロダクション コードではスマートポインタを使用してください。

SongImpl クラスは次の通りです。

SongImpl.h

```
#pragma once
#include   <string>
#include   <memory>

namespaceSongLibrary
{
        class   Lyrics ;
       class    SongImpl
       {
        private  :
                std::   wstring  m_Writer;
                std::   wstring  m_CoAuthor;
                std::   shared_ptr <Lyrics  > m_Lyrics;
                 int   m_YearOfRelease;
        public  :
                 int   GetYearOfRelease()    const  {   return   m_YearOfRelease; }
                std::   wstring  GetWriter()    const  {   return   m_Writer; }
                std::   wstring  GetCoAuthor()   const  {   return   m_CoAuthor; }
                SongImpl(std::   wstring   writer , std::   wstring   coauthor ,
const  Lyrics  *  const  lyrics   ,  int   yearofrelease   )
                        :m_Writer{     writer   }, m_CoAuthor{     coauthor  },
m_YearOfRelease{   yearofrelease   },
m_Lyrics(   const_cast <Lyrics  *>(  lyrics   ))
                {}
                SongImpl(std::    wstring   writer , std::    wstring   coauthor ,
std::  shared_ptr <Lyrics  >  lyrics   ,  int   yearofrelease   )
                        :m_Writer{ writer }, m_CoAuthor{ coauthor },
m_YearOfRelease{   yearofrelease   }, m_Lyrics(     lyrics   )
                {}
                SongImpl(   const   SongImpl& other ) :m_Writer{     other .m_Writer
} , m_CoAuthor{ other .m_CoAuthor},
                        m_Lyrics{    other .m_Lyrics}
                {}
        };
}
```

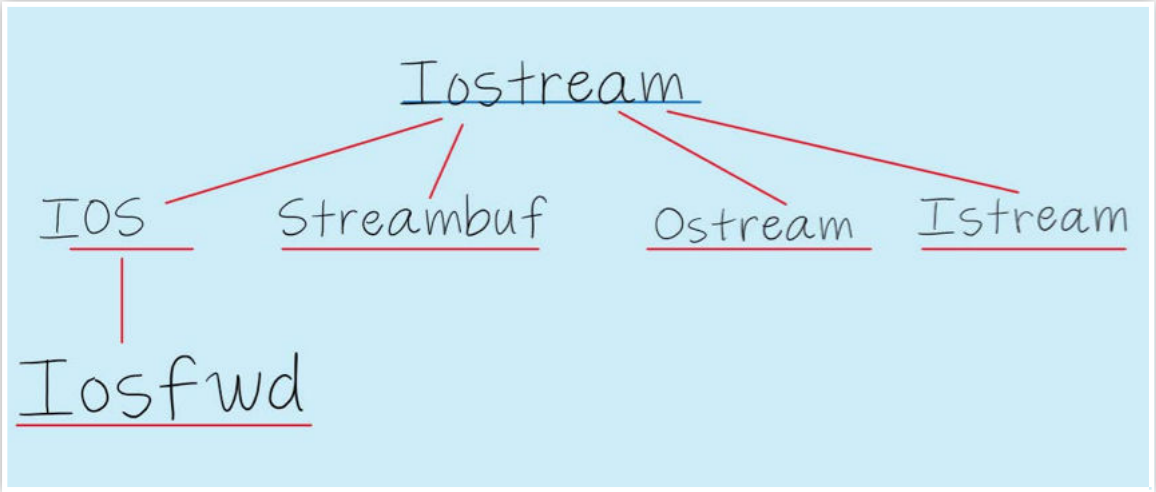
内部実装を変更できます。クライアントのリコンパイルは不要です。

前方宣言

Pimpl イディオムの解説を丁寧に読めば前方宣言の要点は理解できるはずです。ShinyNewString のヘッダーでは、(実装クラスの) Simpl の前方宣言のみに留めて、Simpl ヘッダー全体をインクルードしないようにしましょう。

つまり、ヘッダー内のクラスやストラクチャに対する前方宣言は、それらのクラスを使用する実装ファイルの関連ヘッダーを含めるだけで構いません。これにより、ヘッダーがほかのヘッダー内にインクルードされることが減り、コンパイル時間が短くなります。

参考までに、#inlcude <iostream> には以下がインクルードされています。



コンパイル時間を短縮したいなら、ヘッダー ファイルにできるだけ前方宣言を使用してほかのヘッダーのインクルードを減らしてください。下は前方宣言を使用した例です。

```
namespaceSongLibrary
{
        class   Lyrics  ;
        class   SongImpl
        {
        private  :
                std:: wstring  m_Writer;
                std:: wstring  m_CoAuthor;
                std:: shared_ptr  < Lyrics > m_Lyrics;
                 int   m_YearOfRelease;
        public  :
```

前方宣言。ヘッダーにはポインタのみが使用されています。

プリコンパイル済みヘッダー

プリコンパイル済みヘッダーとは、解析・事前処理が終わった C や C++ のヘッダー ファイルから生成されたバイナリ ファイルのことで、元のファイルに存在するマクロと宣言の両方をソートすることでコンパイル時間を短縮します。コンパイル時にヘッダーの変更されたタイムスタンプとプリコンパイル済みヘッダーのタイムスタンプがコンパイラによって比較され、変更されたタイムスタンプの方が後に作成されていれば、同期を行ってプリコンパイル済みヘッダー ファイルを再作成します。

プリコンパイル済みヘッダーを使用することで、コンパイルを 6 倍も高速化できる可能性がありますが、分散ビルド中はプリコンパイル済みヘッダーがいつも有効とは限らないので注意してください。プリコンパイル済みヘッダーはいくつかのコンパイル プロセスを起動して複数のユニットを並列処理する代わりに、複数のコンパイル ユニットが中断されないようにユニットを集約してしまうからです。

プリコンパイル済みヘッダーの内容が頻繁に変更されると、そのメリットが損なわれることには注意が必要です。

本ガイドではすでにプリコンパイル済みのヘッダー ファイル (pch.h) と CPP ファイル (pch.cpp) について言及しました。pch.h は次の通りです。

Pch.h (VS で自動生成)

```
// pch.h: This is a precompiled header file.
// Files listed below are compiled only once, improving build
performance for future builds.
// This also affects IntelliSense performance, including code
completion and many code browsing features.
// However, files listed here are ALL re-compiled if any one of them
is updated between builds.
// Do not add files here that you will be updating frequently as
this negates the performance advantage.

#ifndef  PCH_H
#define PCH_H

// add headers that you want to pre-compile here
#include   "framework.h"

#endif  //PCH_H
```

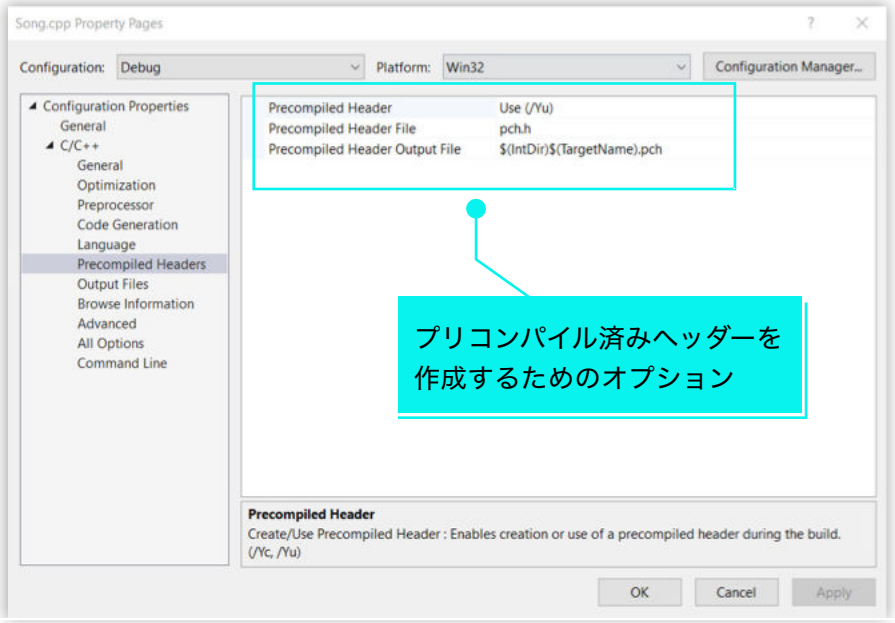
Visual Studio では、プリコンパイル済みヘッダーについて分かりやすいコメントがあります。

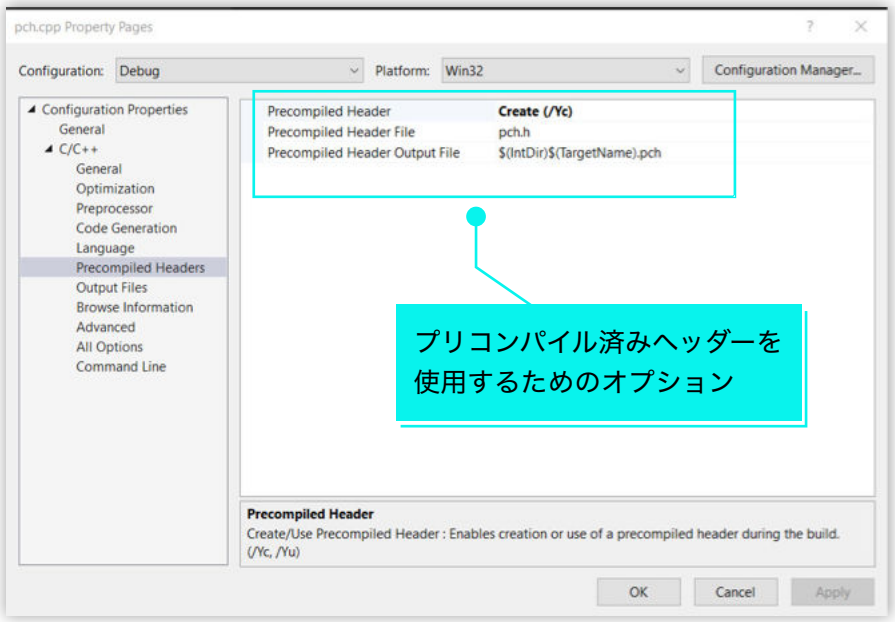
pch.cpp は次の通りです。

Pch.cpp

```
// pch.cpp: source file corresponding to the pre-compiled header

#include    "pch.h"

// When you are using pre-compiled headers, this source file is
necessary for compilation to succeed.
```

とはいえ、もっと重要なのは Visual Studio では CPP を使ってプリコンパイル済みヘッダーを作成できることです。





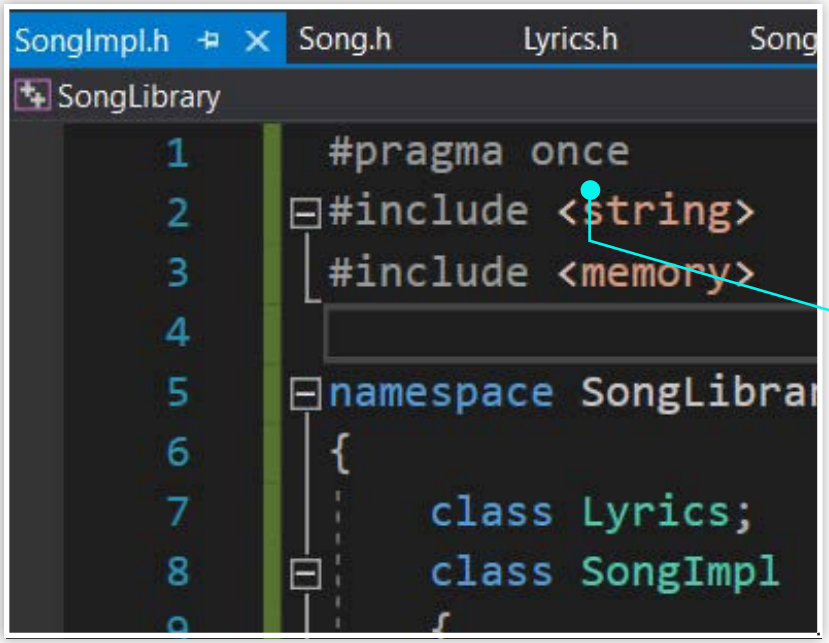
インクルードガード

インクルードガードを使用すると、ユニットのコンパイル時にヘッダー ファイルが二重にインクルードされるのを防ぐことができます。コーディング規約を遵守している多くのプロジェクト (Google のコーディング規約など) では、すべてのヘッダーが多重インクルードを防ぐためのインクルードガード (#define guards) を持つ必要があります。

ヘッダーガードには一意の名前が必要で、これがなければコードの整合性が損なわれる可能性があります。最近のコンパイラでは #pragma once マクロが提供されており、コンパイラは内部でヘッダーガードの一意の名前を選択できます。なので、できれば #pragma once の使用をお勧めします。

直感的にはヘッダーが二重に含まれるのを防げば、コンパイル時間は改善されます。これはその通りで、C++ ビルドのコンパイル時間が著しく短縮するはずです。

本ガイドの例では Microsoft Visual Studio コンパイラを対象にしているため、インクルードガードとして #pragma once を使用しています。クロスプラットフォーム向けのコードの場合は、#pragma ディレクティブをサポートしていないコンパイラがあれば、インクルードガードを手動で作成した方がよいでしょう。



必ずしもサポートされているとは限りません。クロスプラットフォームのコードではインクルード ガードを手作業で記述しなければならないことがあります。例:
#ifndef UNIQUE_NAME
#define UNIQUE_NAME
...
#endif

コンパイル単位の一元化 (ユニティビルド)

これには賛否両論があるものの、C++ のコンパイル時間を短縮するために実践している現場を未だに見かけます。ただし、この方法はコンパイル単位のモジュール性に反し、また小規模なインクリメンタル ビルドには悪影響を与えるのでお勧めできません。

ユニティビルドは複数のコンパイル単位 (CPP ファイル) を 1 つにまとめて、より大きなファイルを作成することでビルド時間を短縮するために行われます。異なる CPP ファイルに含まれる複数のヘッダーを解析する手間を省いてビルド時間を改善するわけです。また、作成されるオブジェクト ファイルの数も減らせるのでリンク時間の短縮も期待できます。ユニティビルドを使用するデメリットとしては、先に述べたようにインクリメンタル ビルドへの悪影響が挙げられます。また、ヘッダーオンリー ライブラリには多くのメリットがありますが、C++ のコンパイル時間が長くなることは指摘しておかなければなりません。ユニティビルドのメリットとデメリットについてはこちらのブログ (https://onqtam.com/programming/2018-07-07-unity-builds/) でご確認ください。

コンパイラの最適化をやめる

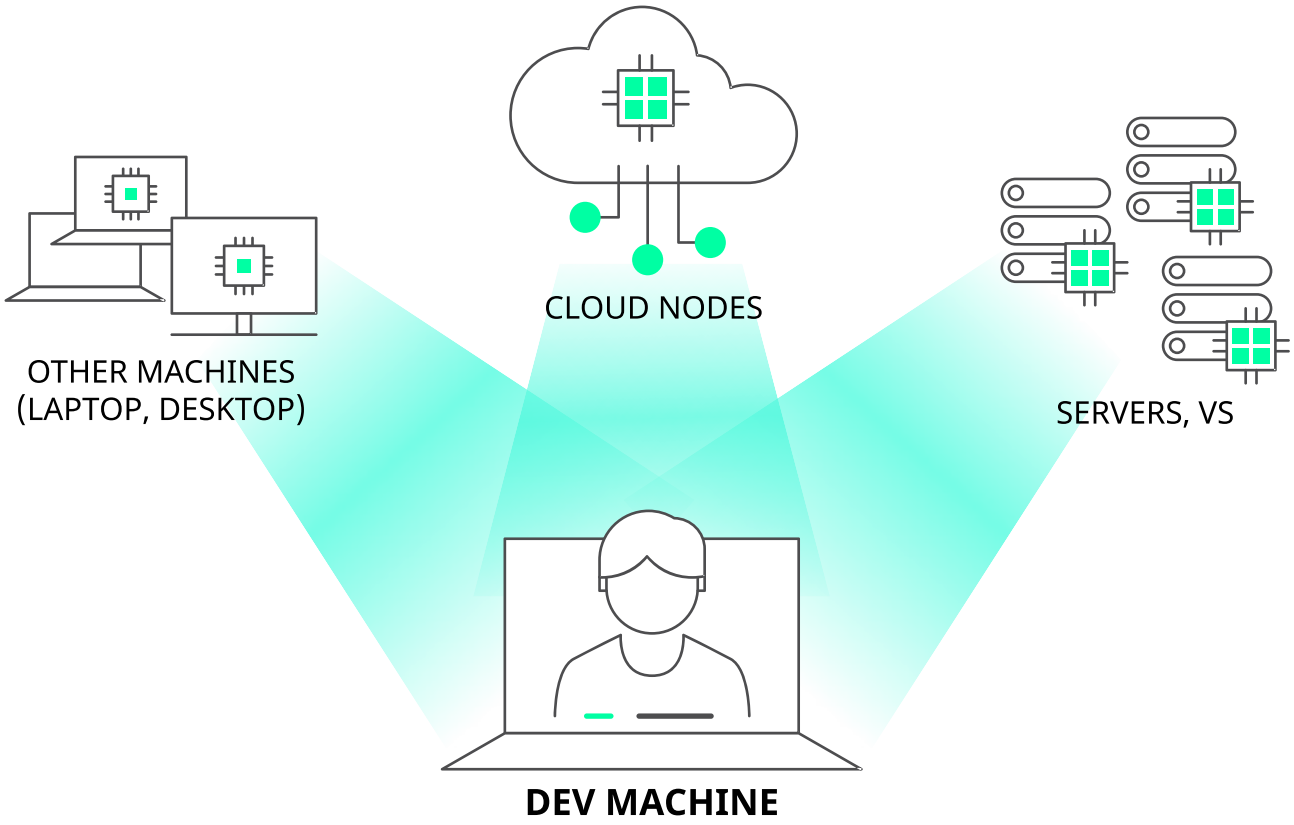
これはお勧めできません。コンパイラはプログラマーより何倍も賢く、コード実行時のパフォーマンスを大幅に向上できます。そのため、ビルド時間を短縮する目的でコンパイラの最適化に手を加えるのは得策とはいえません。通常、デバッグビルド時にはコンパイラのアグレッシブな最適化は自動的にオフになります。これは、デバッグしたバイナリとソースが一致するのを確認するためです。自分が何をしているのかについて確信がない限り、コンパイラの最適化をオフにしないでください。

コンパイルの分散 - Incredibuild のソリューション

もちろんこの手法を最も強くお勧めします。

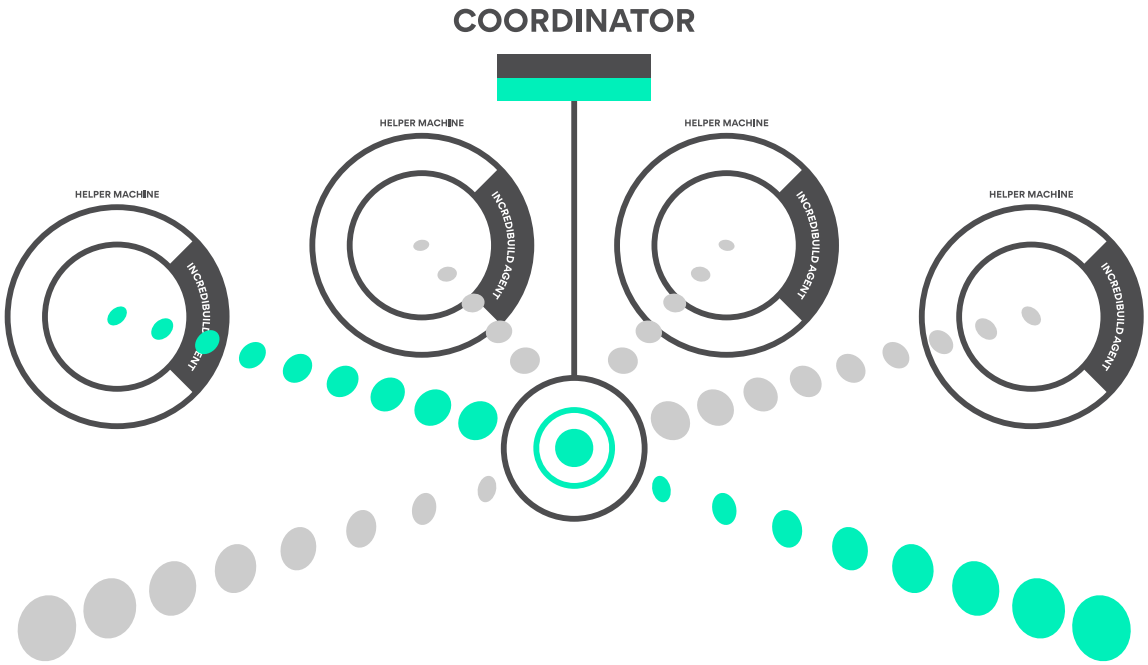
Incredibuild は C++ ビルドの長いコンパイル時間に対処するために開発されており、ビルド高速化を主導するソリューションです。

Virtualized Distributed Processing™ (仮想分散処理) 技術は、ネットワークやクラウド上でアイドル状態の CPU を活用し、リモートマシン上でローカル環境をエミュレートし、すべてのホストを数百から数千コアのスーパーコンピューターにシームレスに変換します。これにより、ビルド パフォーマンスが大幅に向上します。



Incredibuild は使用前に各マシンに Agent のインストールが必要です。Agent はネットワークを管理する Coordinator に接続されており、Agent をインストールしたマシンであれば、Incredibuild 環境内のほかのマシンの使われていないパワーを活用することができます。

企業によってはアイドル状態の CPU が数千に及ぶこともあります。これらの遊んでいる CPU の処理能力を有効活用することでビルドを高速化するのです。



これを利用者の側から見ると、ホストマシンが数百コアを持つスーパー コンピューターに変身し、コンパイルが次々と処理されていくだけです。

Incredibuild のプロセスはリモートマシン上の安全なサンドボックス内で実行されます。プロセスの実行に必要なものは、すべてローカルホストからリモートマシンへ動的にエミュレートされ、安全・安心して処理できます。

当社のウェブサイトで詳細をご覧になるか、無料のライセンスをダウンロードして今すぐお試しください。