The Complete Guide to

Speed Up Your C++ Builds

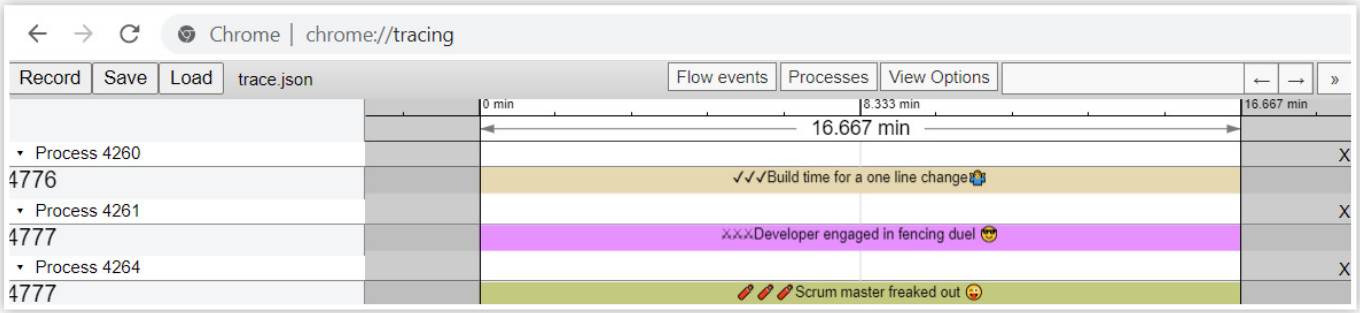
INCREDIBUILD

Table of Contents

The problem with C++ builds

C++ is a great programming language. We are big fans. But there is a real issue with C++ build times. If you are building in C++, there is a good chance that your build times are long, presenting a real challenge to you, your managers, and the entire organization. This issue is not new nor is it rare. The issue is exacerbated especially in Agile work environments where continuous integration/continuous deployment (CI/CD) is the norm. You wouldn't want your check-in to take a whole afternoon to get accepted, would you?

Some handle this issue by not doing much. Some others make use of these long build times to engage in leisure activities (as the old XKCD fencing joke goes). Others, however, identify this issue as something that requires their attention. We will leave it to you to guess which group we belong to.



(The above image is hypothetical, any resemblance to your current project is purely coincidental)

Why do C++ builds take so long?

This question is on a lot of C++ developers' minds. There are several reasons and possible explanations:

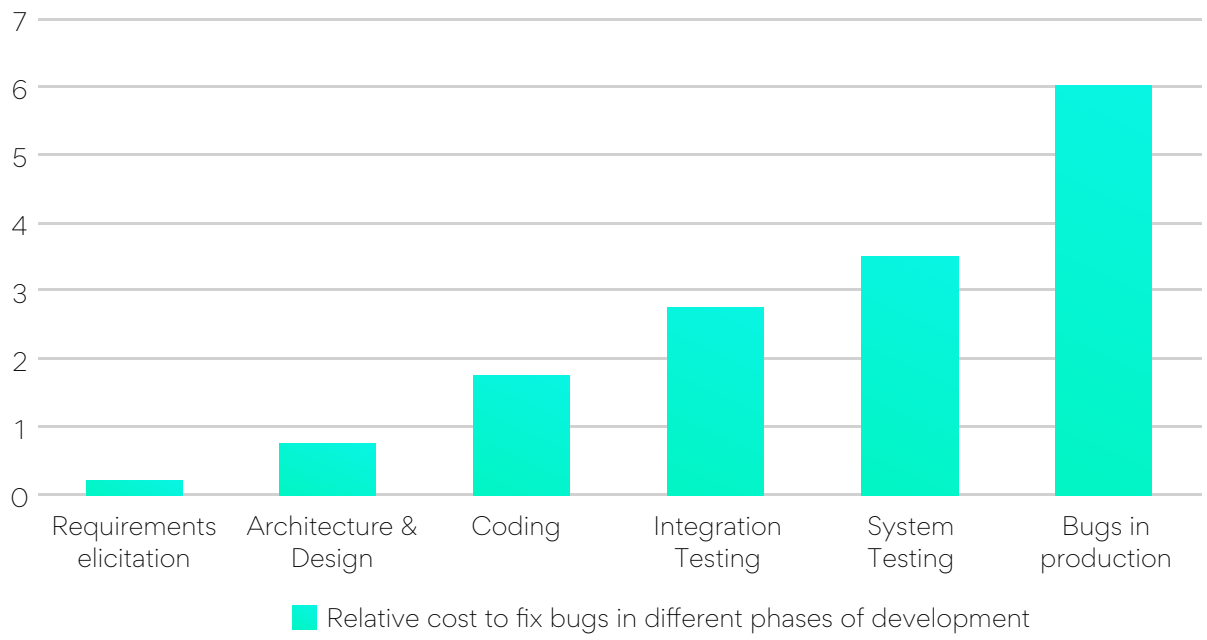
- The build is running on a machine low on resources
- The build has too many dependencies
- The build is using an outdated compiler/linker
- The build is not using precompiled header files / is using them wrong
- The compiler is asked to do the best optimizations
- The codebase is not maintained / gold-plated code
- The codebase is large

While a large number of header files that require parsing is one of the reasons it is not the only one as can be seen above. In general, the complex and dynamic way in which data is processed during compilation is one of the major reasons for longer build times.

Why are longer builds such a problem?

This is a tricky question; however obvious it may sound. In the latest C++ foundation survey more than 40% of developers reported build times are a major problem and around more than 40% consider them as a minor problem. Only 17% don't see it as a problem at all. Waiting for the build to finish or prioritizing the development process accordingly (weekly builds) while delaying tests and skipping tasks just to avoid compiling them has a potentially devastating effect on the organization.

It is not just the delivery time that is affected. The underlying quality of the software is affected by ignoring large build times.



This is a well-known graph of relative cost to fix bugs based on the time of detection. If testing suffers due to longer build times, it is natural that more and more bugs will escape undetected to production. This will increase the overall cost of development and the quality of the software will suffer. In the current fiercely competitive market, top-quality frequent releases are vital.

Shift left is swiftly gaining momentum. If developers are empowered to run tests before committing their code to the repository the external quality of the software will increase. If static code analysis is also done alongside, the internal quality of the software will increase too.

Developer productivity is also adversely impacted by longer build times. If the feedback from the build takes longer, the train of thoughts leading to better code can adversely get affected too. So, there are both direct and indirect costs associated with longer build times.

The good news is: there are things you can do about it!

In this guide, we are going to walk you through various ways to reduce C++ compilation time. Why would we want to do that? Are we not in the business of speeding up compilation? Isn't writing a guide that presents other solutions to this problem a direct conflict of interest to our business?

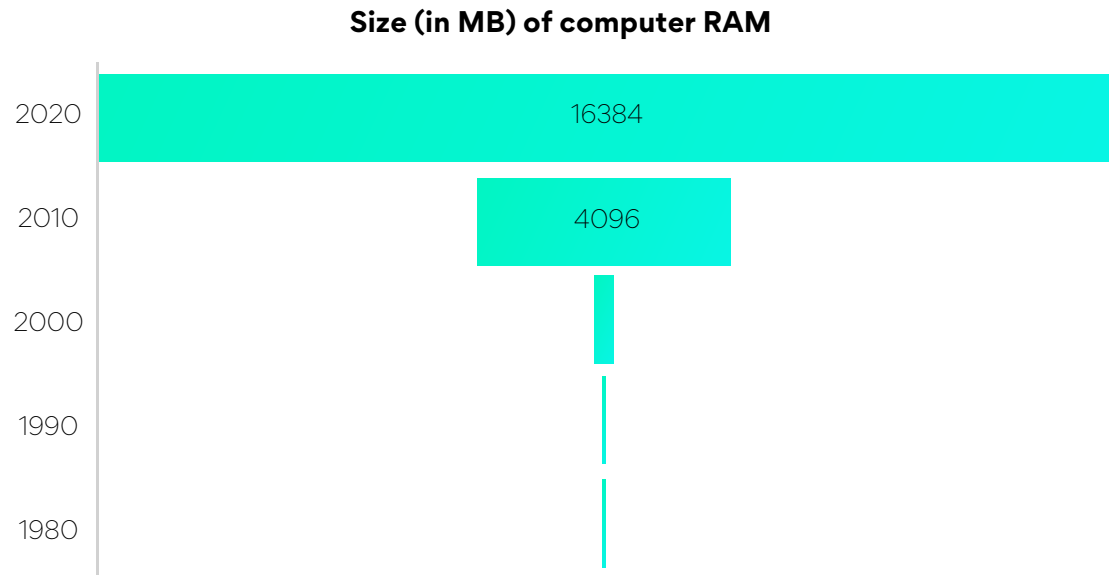
We sincerely believe there are plenty of things that can be done to improve compilation times. Indeed, the solutions presented in this guide are all valid and can offer real value in some cases, although it might require a bit of work before reaping their benefits. Using Incredibuild for speeding up the builds is a different ball game on a whole new league of its own.

Without further ado, let us jump right in.

Getting a better build machine

This is a given. Investing in quality hardware is an obvious solution. We couldn't write about reducing C++ build times without at least considering the possibility of acquiring better hardware. More RAM, better hard disks, and better CPU can improve your build times.

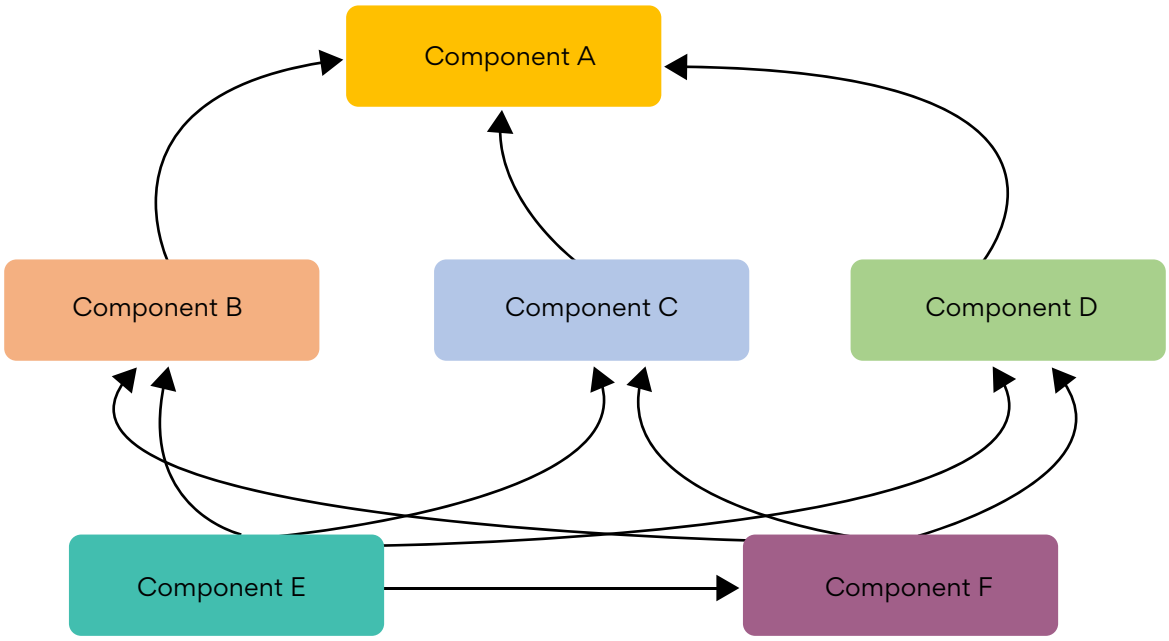
Moore's law depicted for computer RAMs:



Reduce dependencies

Tight coupling between files, components, modules, and layers increases the build time. If the design diagram of the project is along these lines (the boxes can be files, modules, or layers too) then there is a problem.

Questions are: Are all these dependencies required? Which of them can be easily decoupled? Which decoupling would need more work or is most risky? Bring in an architect to analyze the costs and benefits and do a re-engineering of the project. As the internal code quality improves there will be a decrease in the compilation times. This is a general technique not constrained to C++-based projects alone.



A tightly coupled system is always undesirable

Let us take a running example to showcase some of the issues we talk about in this guide and show you how to mitigate them. We introduce some header files first:

Lyrics.h

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

Song.h

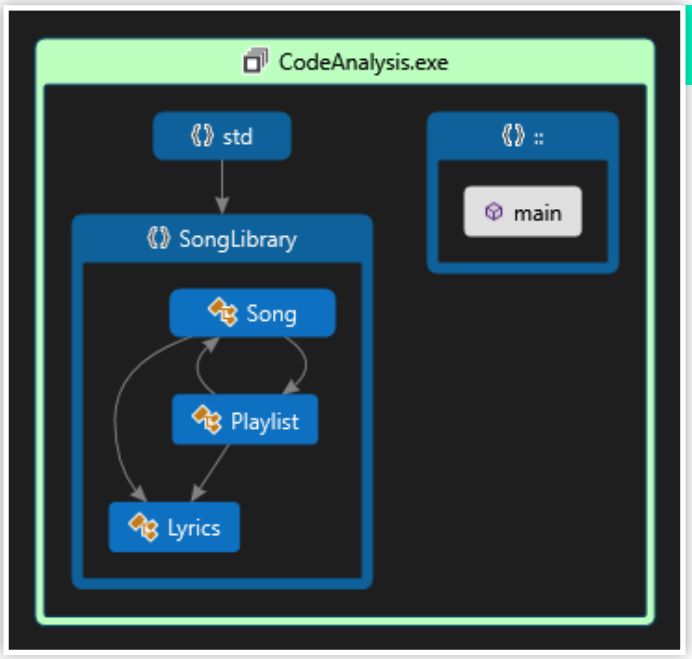
```
#pragma once
#include <memory>
#include <string>
#include <vector>

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

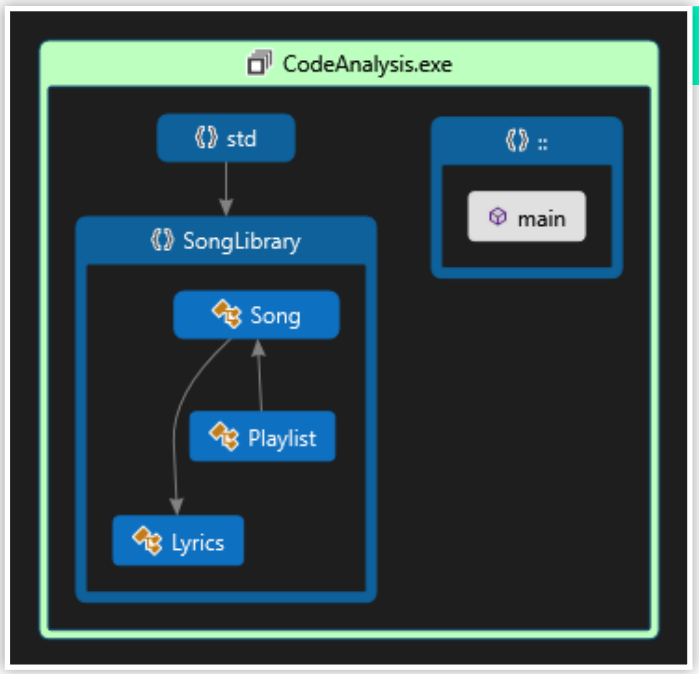
```
#pragma once
#include "Song.h"
#include "Lyrics.h"
#include <vector>
#include <chrono>

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);
	};
}
```

How does the design of this system look like?

Not a good design



Do you see the circular dependency between Song and Playlist in the above diagram? How should the system actually look like?



Much better design

As you improve your design of your system, it will intrinsically improve your build times. (Checkout our excellent build monitor and visualization tool for easier analysis of bottlenecks and dependencies here). Also, as you improve your design other different ways to improve your build times will also become apparent as we see in the next section.

Static Vs Dynamic linking

This option might be platform-specific, but all modern operating systems have a way to dynamically link to code.

- Dynamic Linked Libraries (DLL in Windows)
- Dynamically loaded modules (dylib in Macintosh)
- Dynamically loaded libraries (DL in Linux)

Utilities that are rarely changed can go into dynamic libraries. Since the code is not getting compiled for every build of the system, this can drastically improve the compilation times.

Making such a change again involves refactoring and reengineering. One of the costs you will have to incur is the versioning of such dynamic code. As and when the interface of a DLL changes, it is recommended to assign it a new version. One of the well-known versioning schemes is shown below:

Major Version.	Minor Version.	Build Number.	Revision

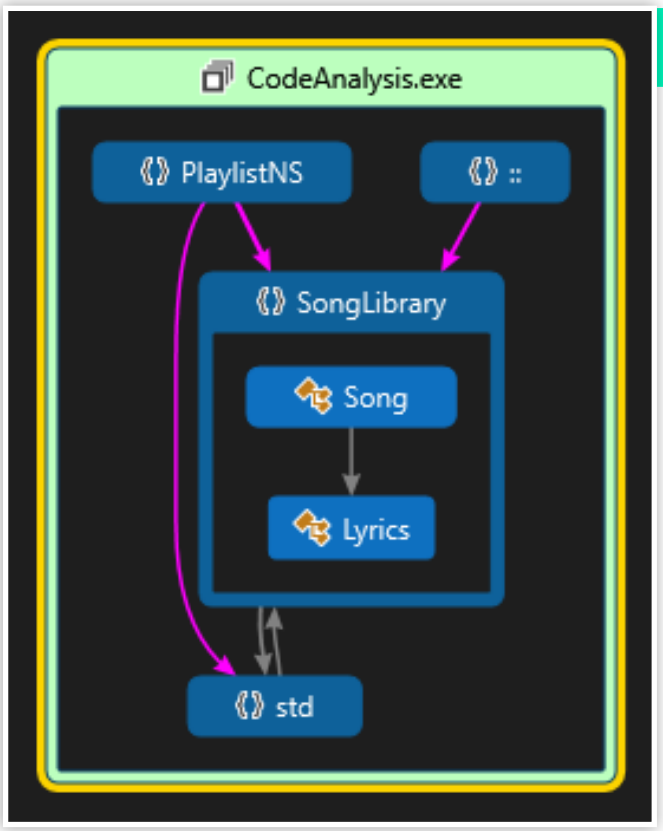
If customers are using different versions of the product then the major version numbers for DLLs will be different. Maintaining them across product versions incurs a cost, but from the point of view of builds having DLLs are better than static linking.

We had introduced a running example in the previous section where we saw how to improve the design. Now let us further improve the design. First we notice that playlist has to be separated out of the SongLibrary namespace to an independent namespace. Let us do so:

Playlist.h (Changed)

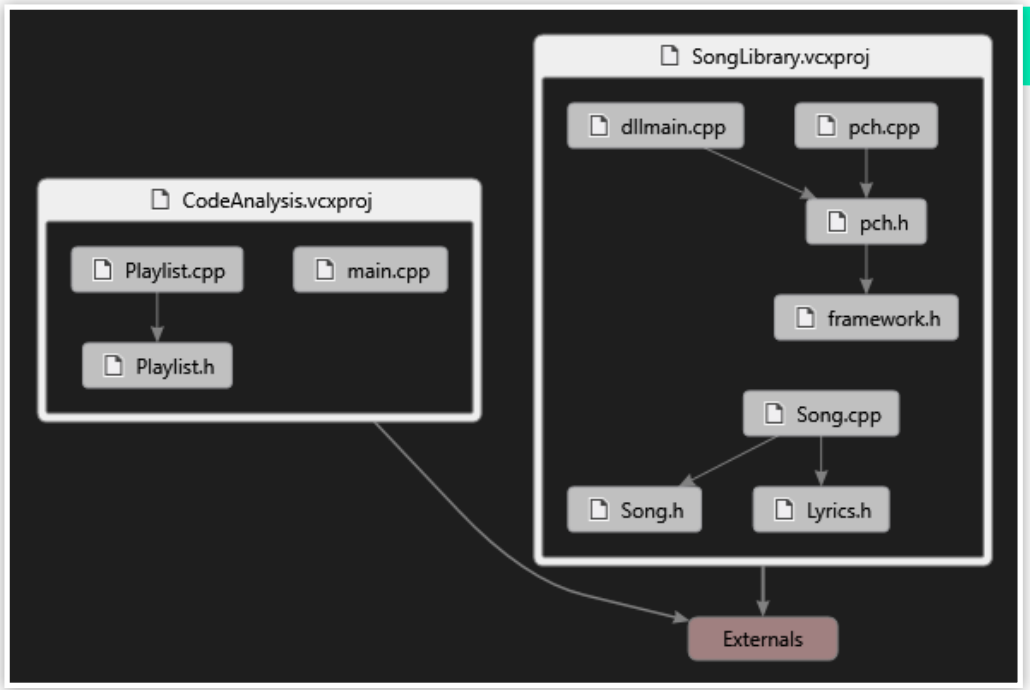
```
#pragma once
#include "Song.h"
#include "Lyrics.h"
#include <vector>
#include <chrono>

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();
	};
}
```

What do we see?



Components getting clearer

Now it is clear that we can separate everything in SongLibrary into an independent library. We see that nothing much changes in the SongLibrary. So we separate it out to a dynamic library.



Modularization into DLL

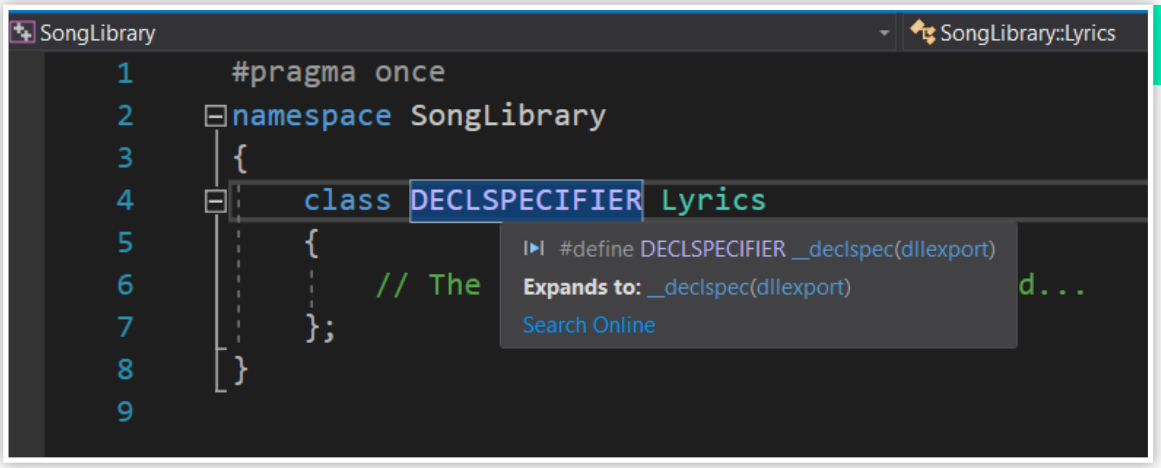
We will discuss the new pch.h/pch.cpp files in one of the upcoming sections. Let us concentrate on the the framework.h file. Here is what it has:

Framework.h

```
#pragma once

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

This is a mechanism in Windows to export functions from a DLL. Let us see how has the other files changed due to the creation of separate DLL.

Lyrics.h (Changed)



As can be seen, the class Lyrics is exported from the DLL. A similar change will have to be done on Song.h too. But we also notice a warning which needs to be fixed:

Severity	Code	Description	Project	File	Line
Warning	C4251	SongLibrary::Song::m_Writer': class 'std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t>>' needs to have dll-interface to be used by clients of class 'SongLibrary::Song'	CodeAnalysis	D:\Demo\SongLibrary\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:
                  ...
```

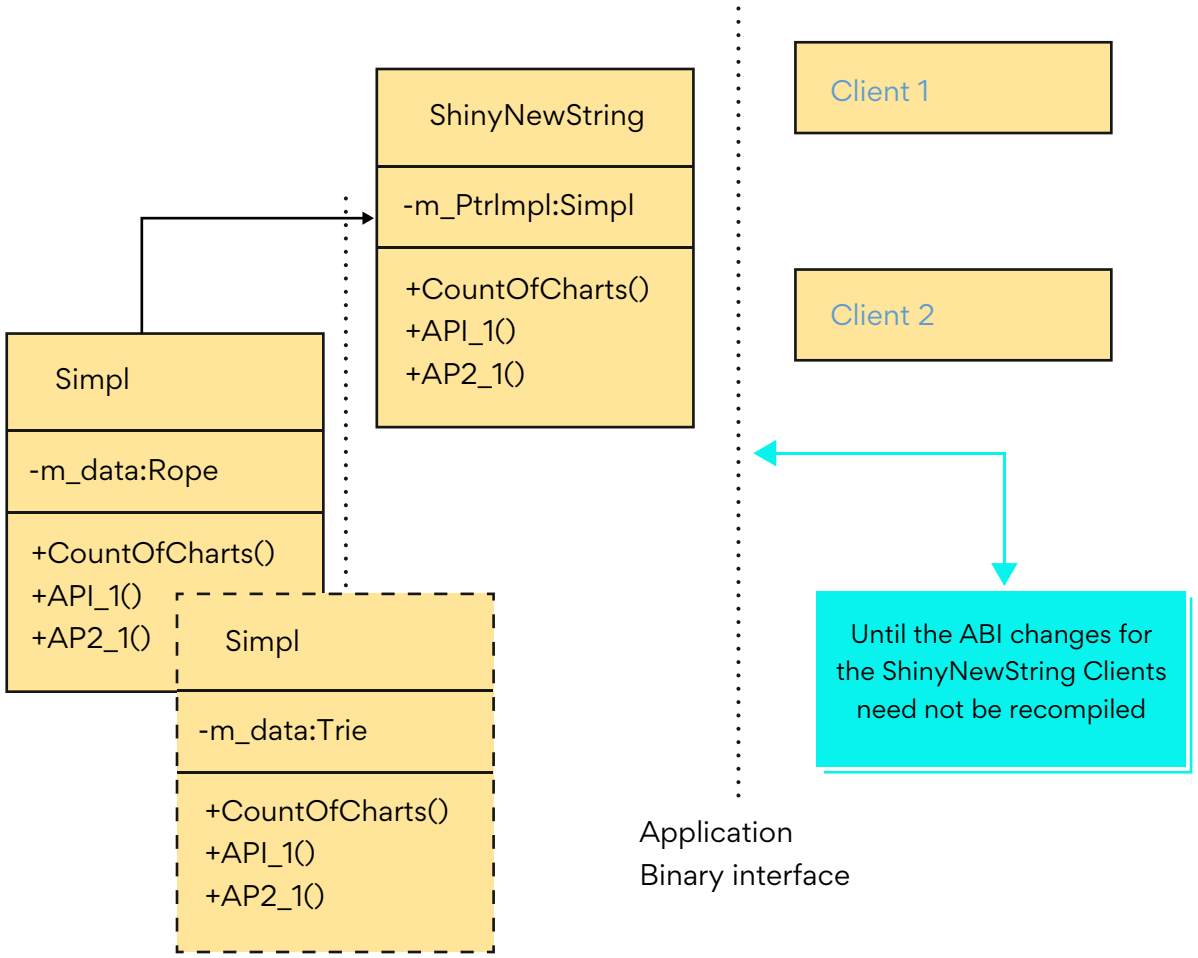
Fix for the warning

As can be seen, we are forced to generate all members of class std::shared_ptr<Lyrics>. Why so? Because STL classes at DLL interfaces is not a good design choice. What can we do about it? Enter the PImpl!

PImpl idiom and its advantages

PImpl is a well-known technique to improve the build time of C++-based projects by reducing the dependencies between classes. It is also known as compile-time firewall as it prevents the compiler from seeing the details of implementation. The implementation might change, but since the interface remains the same for the clients using the class, they don't have to be recompiled. This greatly improves the performance.

Let us take an example of the shiny new string class that was required to be designed:



Since the design uses a pointer to implementation, changes to internal implementation are opaque to external clients of the class. This is exactly why the technique is also known as compile-time firewall.

As always, remember that there is a tradeoff and in the case of pImpl, this cost is performance. A level of indirection is necessary to execute the member functions of the class as it gets delegated to the underlying implementation class. Code becomes a bit more complex and also the testability of code decreases. But pImpl is a good technique to improve build times of C++-based projects.

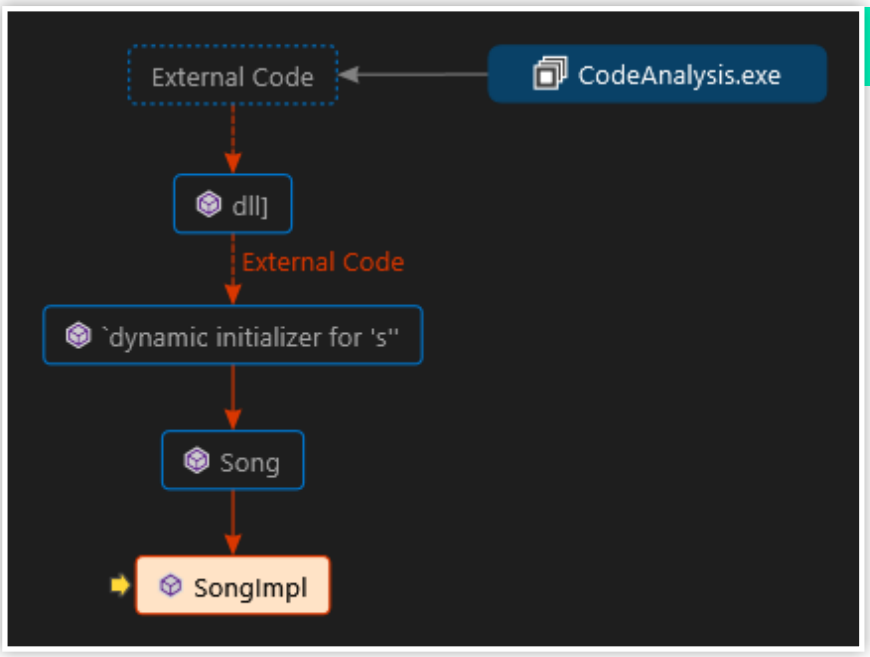
Let us go back to our running example to illustrate in code how pImpl idiom is used to fix our design.

We have this interface:

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:
...
```

We change the design to:



pImpl Design for Song Class

In the code this looks:

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.
	};
}
```

Only for demonstration. Please use smart pointers in production code.

Finally, here is the SongImpl class:

SongImpl.h

Internal implementation can be changed. Clients need not be recompiled.

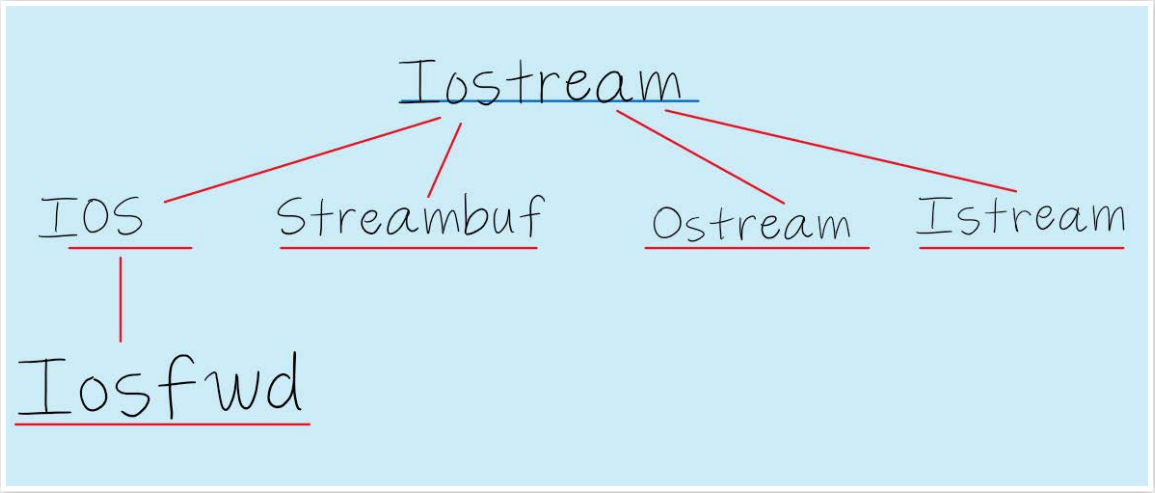
```
#pragma once
#include <string>
#include <memory>

namespace SongLibrary
{
	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}
		{}
	};
}
```

Forward declarations

If you have closely followed the pImpl idiom exposition, you will understand the gist of forward declarations. In the header for ShinyNewString, you should only be having a forward declaration of Simpl (which is the implementation class) and not #include the whole Simpl header.

Forward declarations to classes and structures inside a header imply that you only need to include the relevant headers in the implementation file that use those classes. This decreases the inclusion of headers inside other headers thereby reducing the compilation times.

For your reference, this is what gets included when you #inlcude <iostream>



For faster compilation times, try to use forward declarations as much as possible in header files reducing the inclusion of other headers. Below we highlight one instance where we have used forward delcarations in our running example.

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

Forward declaration. Only a pointer is used in the header.

Precompiled headers

Precompiled headers are binary files that have been generated from C or C++ header files that have been parsed and pre-processed. In a precompiled header file, both macros and declarations present in the original file are sorted resulting in a faster compilation. During compilation, the compiler checks if the modification timestamp of the header is later than that of the precompiled header and if so, do a sync to recreate the precompiled header file.

It is possible to get a 6X reduction in compilation times using precompiled headers. But remember that during distributed builds pre-compiled headers are not always a win as instead of building multiple units in parallel by invoking multiple compilation processes precompiled headers aggregate the units thereby preventing the breaking of compilation to multiple units. Remember that if contents of a precompiled header change frequently, then the advantages thereof are negated.

In our running example we had already alluded to pch.h and pch.cpp. These are the precompiled header and CPP files. The pch.h contain:

Pch.h (Autogenerated by VS)

```
// 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
```

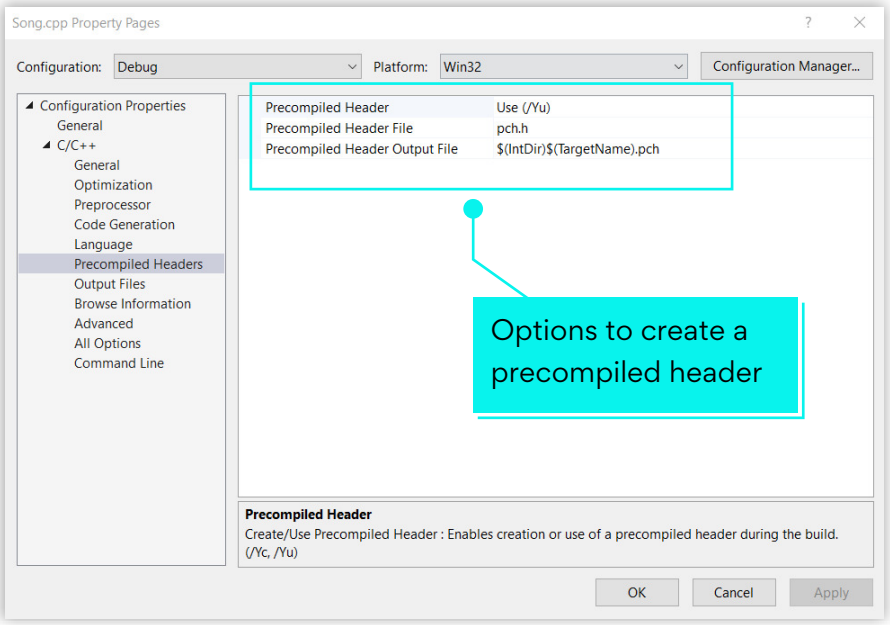
Visual Studio gives excellent comment on precompiled headers.

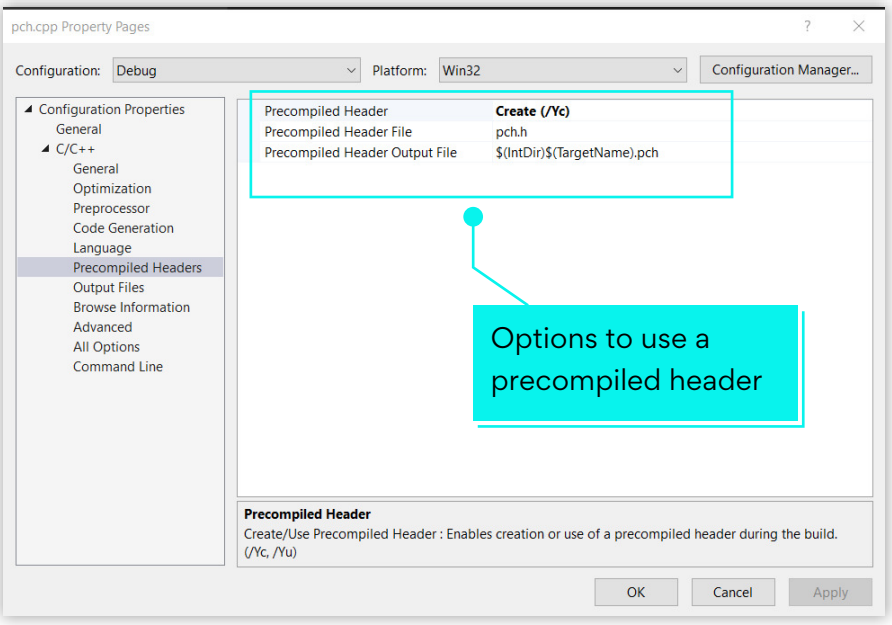
The CPP file pch.cpp contains:

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.
```

But more importantly, Visual Studio has set the options to create the precompiled header using this CPP





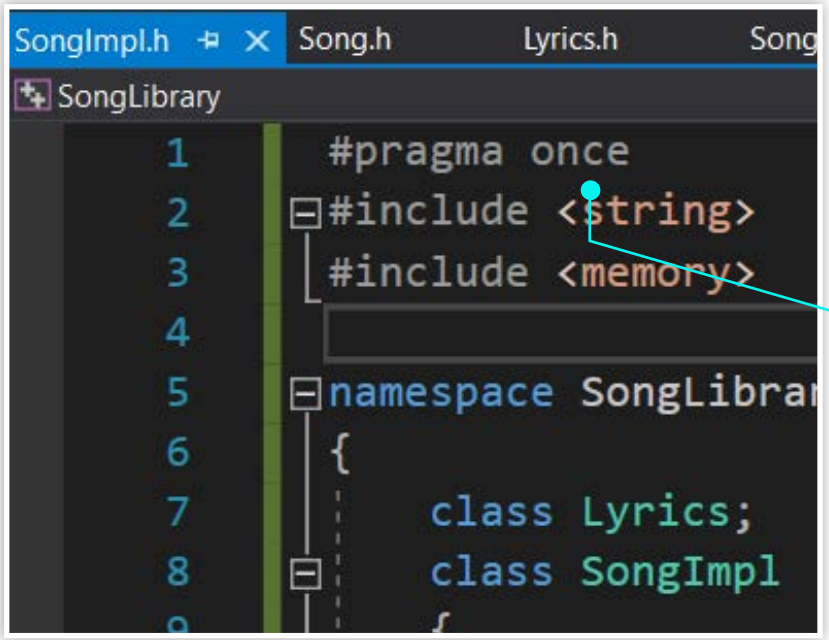
Include guards

By using include guard, you can prevent a header file from being included multiple times during the compilation of a unit. In most of the projects that adhere to a coding standard (e.g., Google's coding standard) all headers must have #define guards to prevent multiple inclusion.

Developers are not always successful in coming up with unique names for header guards. This can harm the correctness of the code. Modern compilers provide a #pragma once macro that lets the compiler internally chose a unique name for the header guard. We advise using #pragma once wherever it is available.

Intuitively, preventing a header from getting included multiple times will improve the compilation times. So, follow this advice diligently and see a marked reduction in compilation times of your C++ builds.

In our running example, we have always used #pragma once as header guards since our code is targeted for Microsoft Visual Studio compiler. If your code is cross platform and if any of the platform compiler do not support #pragma directive, it is better to create header guards by hand.



May not always be supported. Cross platform code may need yout to write header guards by hand. Like so:
#ifndef UNIQUE_NAME
#define UNIQUE_NAME
...
#endif

Single compilation unit

This approach is quite controversial but we still have seen this practiced to reduce compilation times of C++ builds. Although we don't recommend it as it goes against the modular nature of compilation units and can also have significant penality on small incremental builds.

In this approach to improve build times, multiple compilation units (the CPP files) are combined into one single but a larger file. This improves the build times as duplicate effort in parsing multiple headers included in different CPP files is eliminated. The number of object files created during this technique is also reduced thereby reducing the link times. One disadvantage of using single compilation unit builds (unity builds) is that incremental builds are no longer possible when using this approach. It is also worth noting here that although header-only libraries have many benefits it increases the compilation time of your C++ builds. You can checkout the blog (https://onqtam.com/programming/2018-07-07-unity-builds/) for the pros and cons of unity builds.

Turn off compiler optimizations

We strongly advise against taking this route. Compilers are many times cleverer than the programmer and can greatly improve the run time performance of the code. It is not advisable to turn off compiler optimizations in the guise of improving build times. Usually, during debug builds, aggressive compiler optimizations are automatically turned off. This is to make sure that the debugged binary matches the source. Unless you are certain of what you are doing, we don't recommend turning off compiler optimizations for improving compilation times.

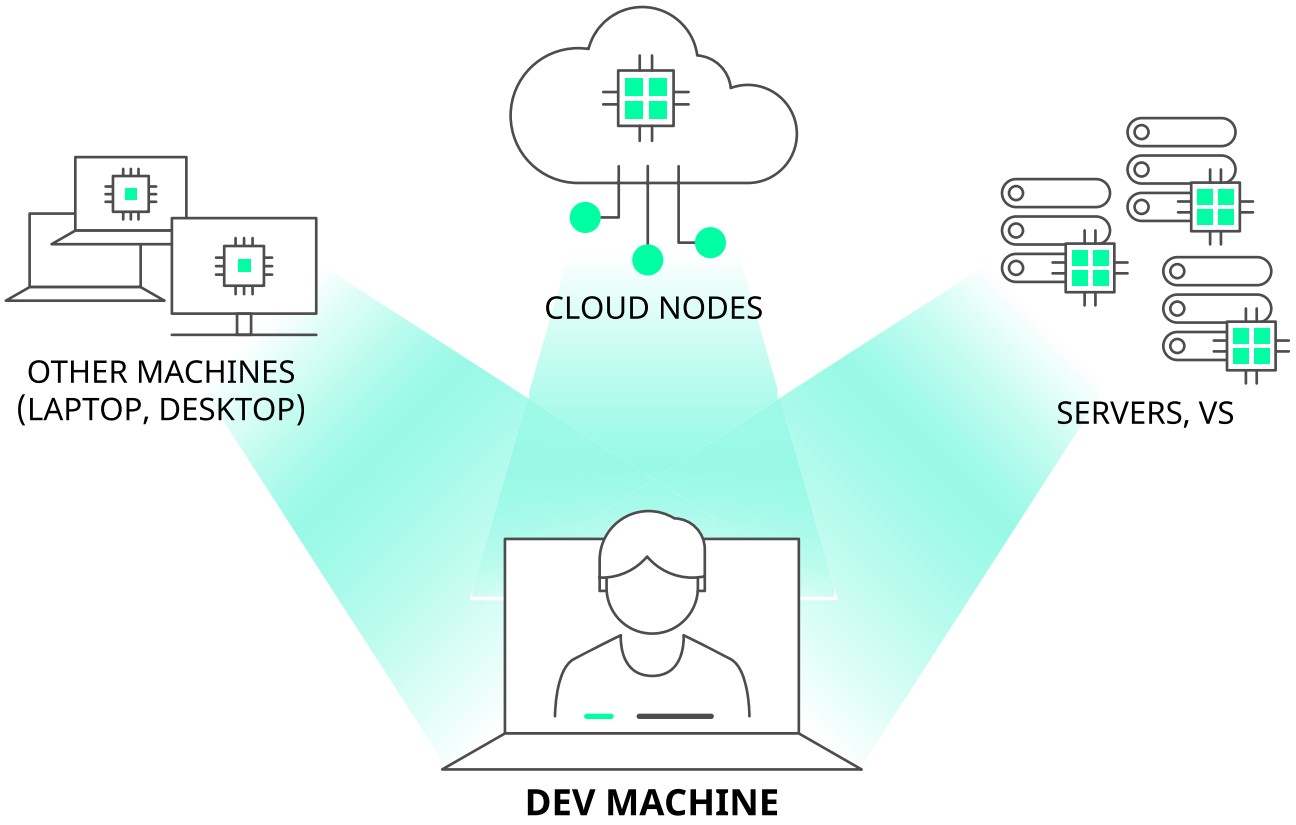
Distributed compilation – the incredibuild solution

We strongly advise you to take this approach, of course. 🙂

Incredibuild was created specifically to tackle the long compilation times of C++ builds. No wonder we are considered world leaders in build accelerators.

Our Virtualized Distributed Processing™ technology harvests idle CPU across your network and the cloud, emulates your local environment on remote machines, and seamlessly turns every host into a supercomputer with hundreds—even thousands—of cores. This greatly improves build performance.

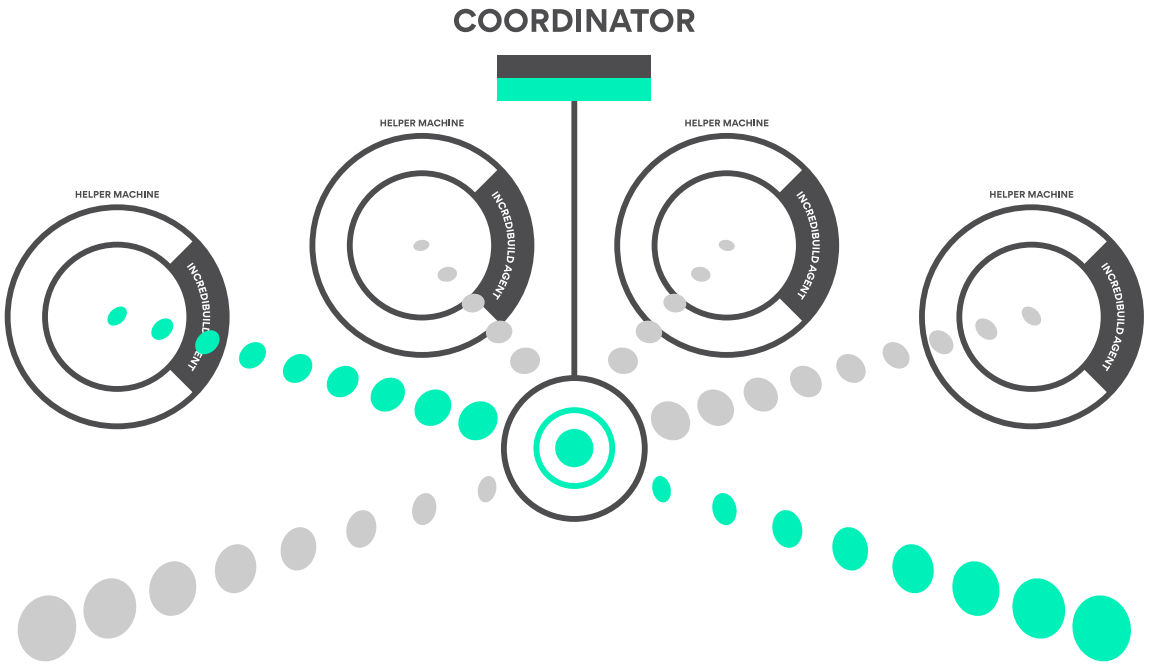
Here is how we do it:



Agents installed on each host are connected to a centralized coordinator. Each host machine with an agent can use the idle power of all the other machines in the Incredibuild environment.

In organization environments, the aggregated number of idle CPUs can easily be in the thousands. The processing power of these wasted cores is effectively used to get faster builds.

From the user's perspective here is what happens during an Incredibuild.



It is as though the host machine is a supercomputer with hundreds of cores and the compilation workload is executed dramatically faster.

Incredibuild runs processes on remote machines in a secure sandbox. Everything each process requires to run properly is dynamically emulated from the local host to the remote machine. We are safe and secure.

To learn more visit our website or download our free license.