



Behind the Scenes of the Projucer C++ Live-Build Engine

ADC'16 London, November 2016
Stefan Gränitz, Consultant Software Development JUCE



Outline



Part 1: Conceptual

- Demo
- How it works
- C++ for Live Coding?

Part 2: Technical

- Linking: Static vs. Live



Live builds – High-level overview



Live builds – High-level overview

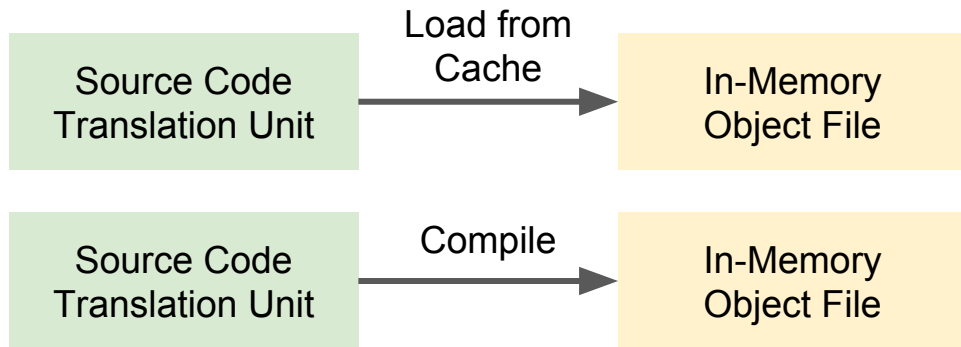
Source Code
Translation Unit

Source Code
Translation Unit



Behind the Scenes of the Projucer C++ Live-Build Engine

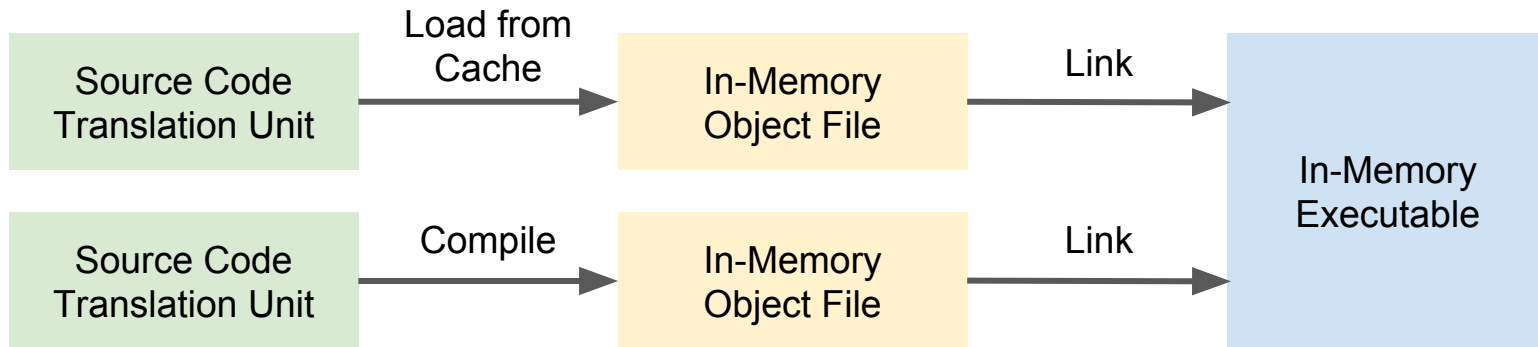
Live builds – High-level overview





Behind the Scenes of the Projucer C++ Live-Build Engine

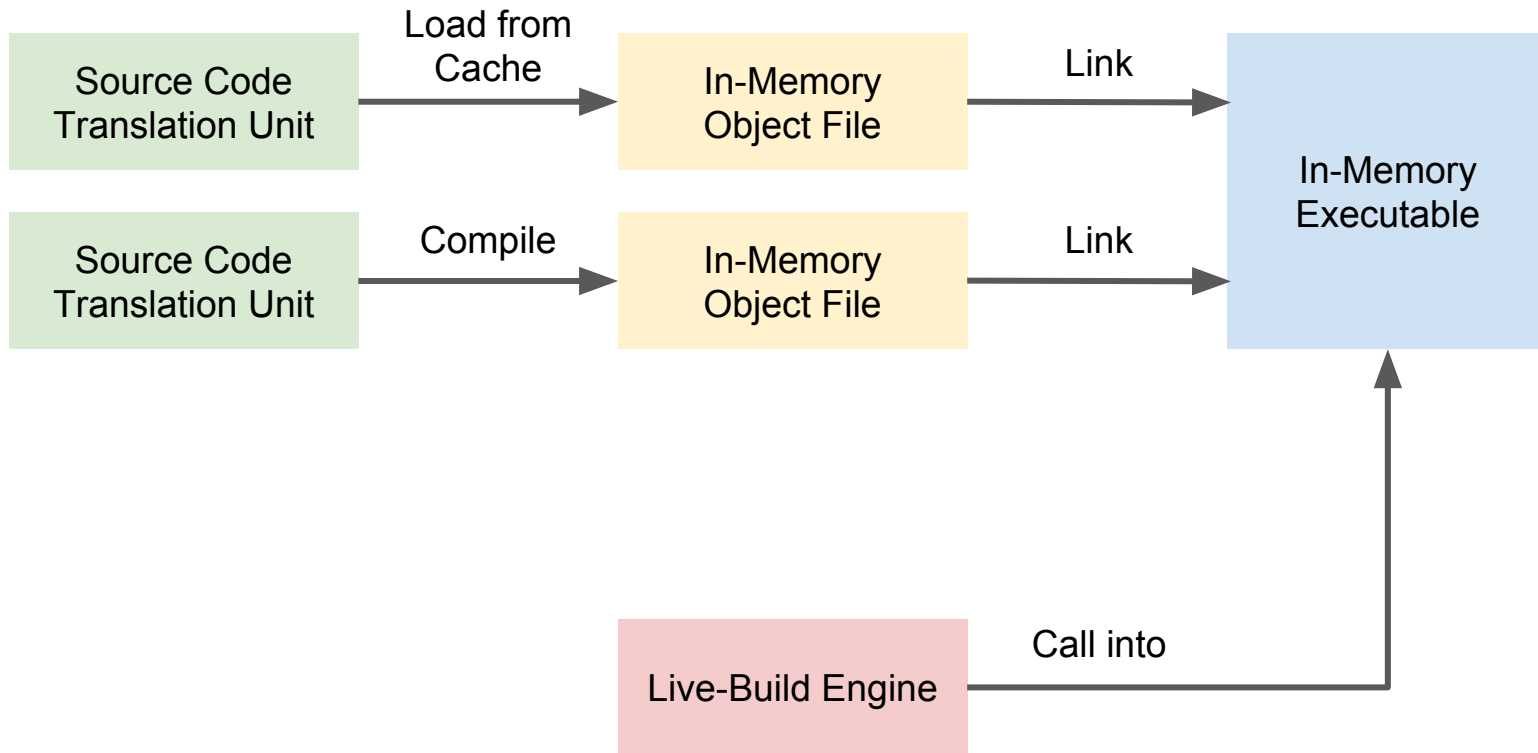
Live builds – High-level overview





Behind the Scenes of the Projucer C++ Live-Build Engine

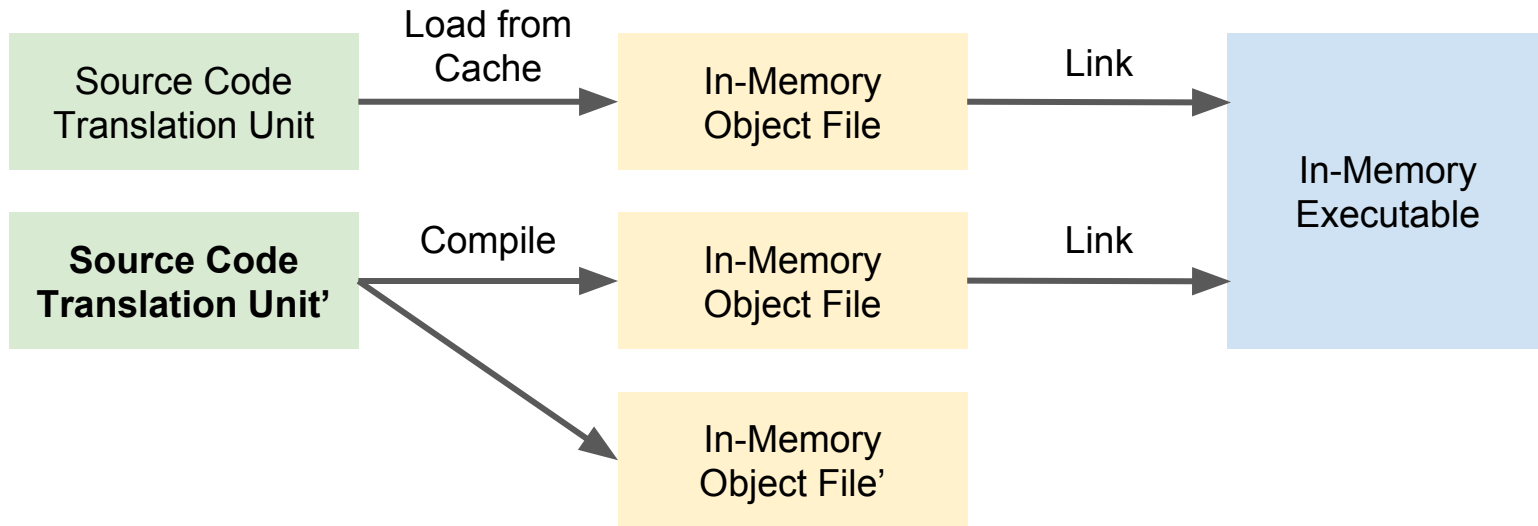
Live builds – High-level overview





Behind the Scenes of the Projucer C++ Live-Build Engine

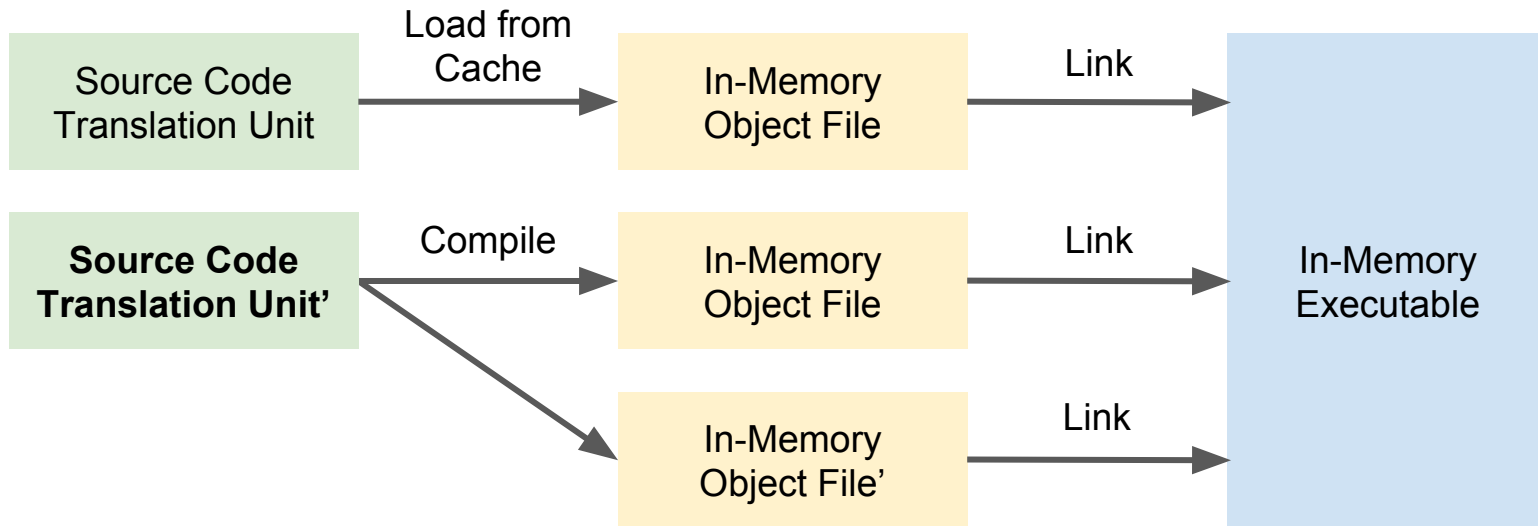
Live builds – High-level overview





Behind the Scenes of the Projucer C++ Live-Build Engine

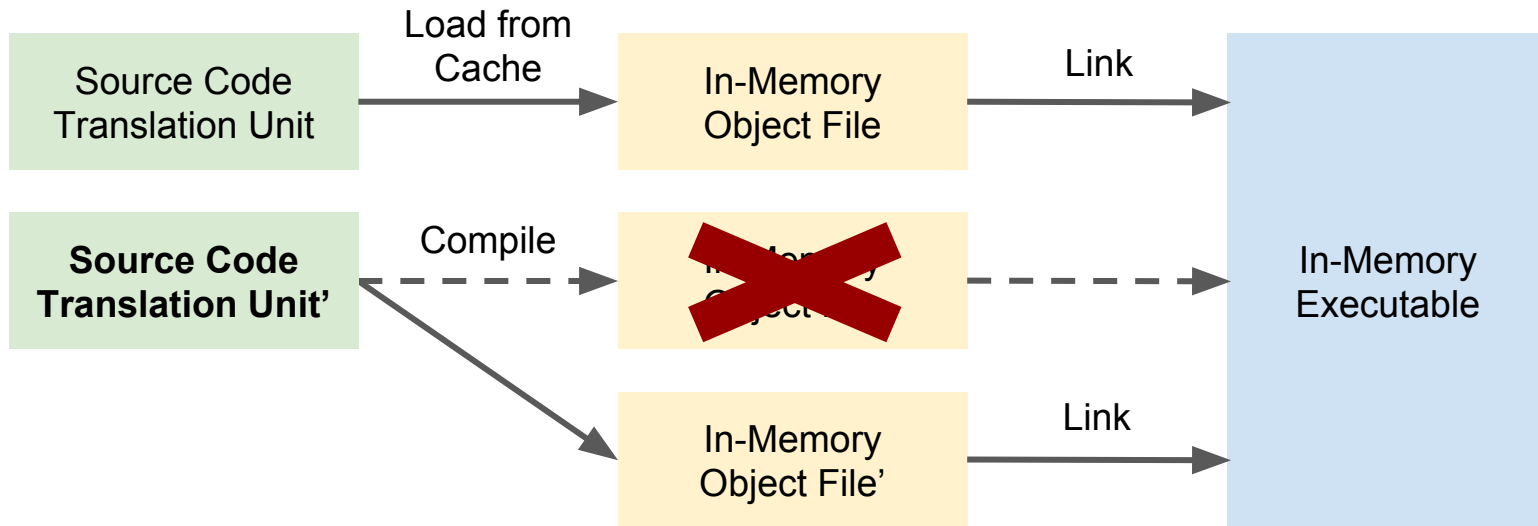
Live builds – High-level overview





Behind the Scenes of the Projucer C++ Live-Build Engine

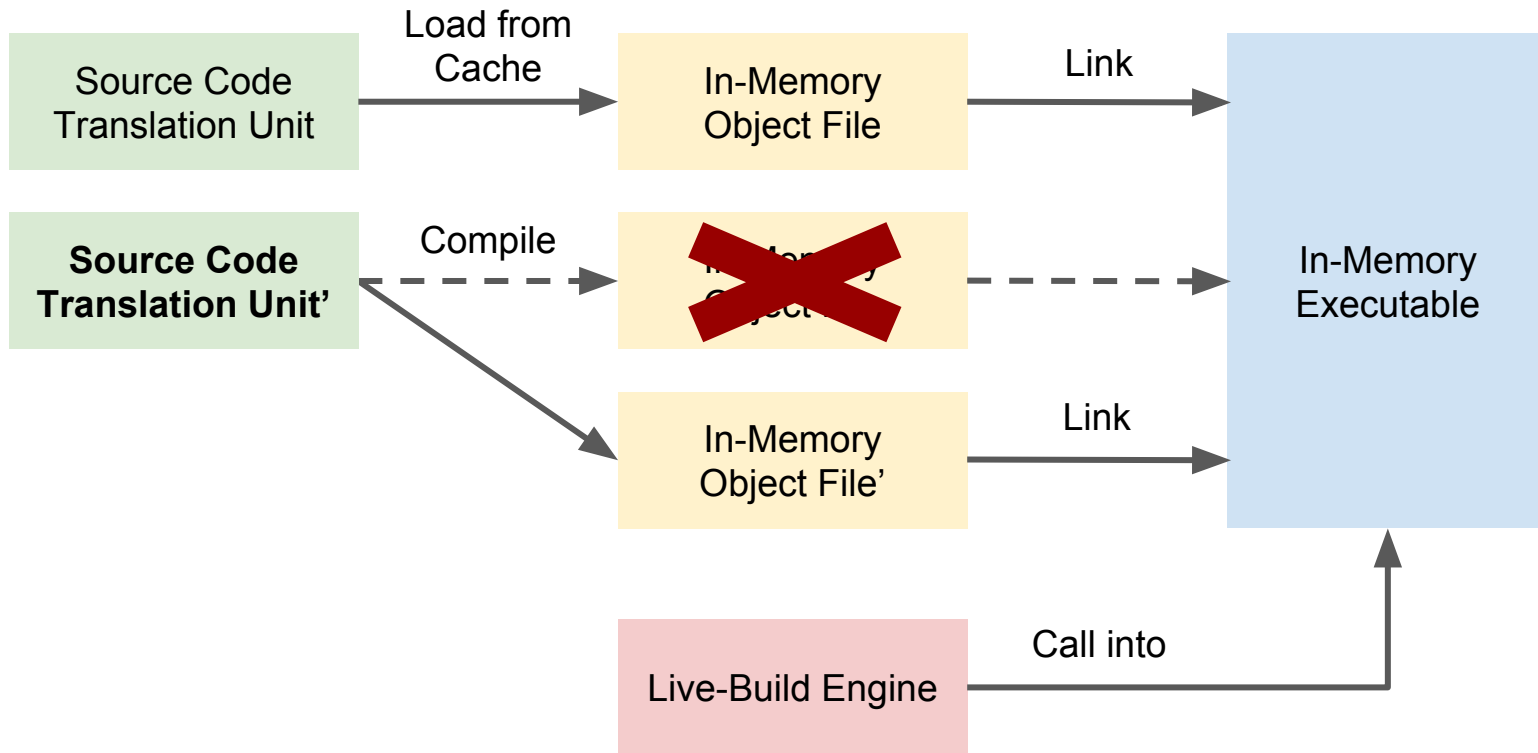
Live builds – High-level overview





Behind the Scenes of the Projucer C++ Live-Build Engine

Live builds – High-level overview





C++ vs. a Language Constructed for Live Coding



C++ vs. a Language Constructed for Live Coding

Ideal	guaranteed crash-freeness → declarative or functional
C++	imperative → inherently not crash-safe

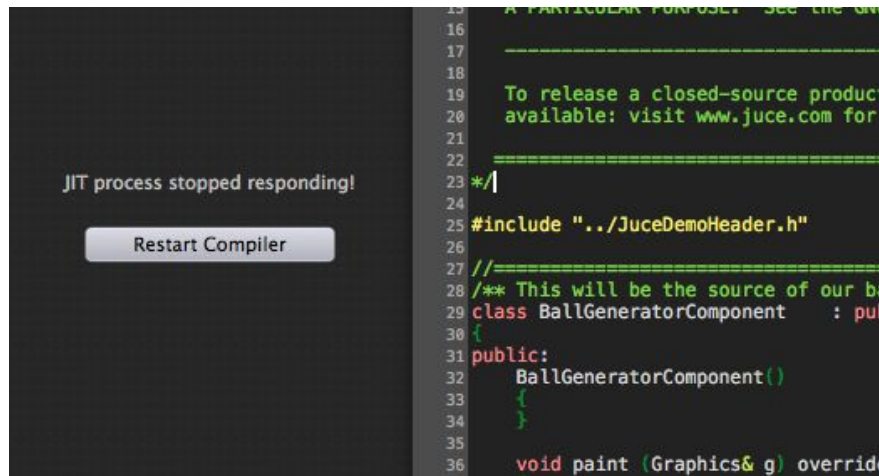


Behind the Scenes of the Projucer C++ Live-Build Engine

C++ vs. a Language Constructed for Live Coding

Ideal guaranteed crash-freeness → declarative or functional
C++ imperative → inherently **not** crash-safe

Projucer: live code runs in a
separate process





C++ vs. a Language Constructed for Live Coding

Ideal	unified runtime environment → avoid arbitrary external dependencies
C++	“There should be no language beneath C++” → handle all quirks of supported platforms manually



C++ vs. a Language Constructed for Live Coding

Ideal unified runtime environment → avoid arbitrary external dependencies

C++ “There should be no language beneath C++”
 → handle all quirks of supported platforms manually

- Object File formats: PE/COFF on Windows, Mach Object on MacOSX x64, ELF on Linux



C++ vs. a Language Constructed for Live Coding

Ideal unified runtime environment → avoid arbitrary external dependencies

C++ “There should be no language beneath C++”
 → handle all quirks of supported platforms manually

- Object File formats: PE/COFF on Windows, Mach Object on MacOSX x64, ELF on Linux
- No standardized ABI: name mangling



C++ vs. a Language Constructed for Live Coding

Ideal unified runtime environment → avoid arbitrary external dependencies

C++ “There should be no language beneath C++”

→ handle all quirks of supported platforms manually

- Object File formats: PE/COFF on Windows, Mach Object on MacOSX x64, ELF on Linux
- No standardized ABI: name mangling

```
void foo(int bar)
```

→ unix-like style: `__Z3fooi`

→ msvc style: `?foo@@YAPEAHH@Z`



C++ vs. a Language Constructed for Live Coding

Ideal unified runtime environment → avoid arbitrary external dependencies

C++ “There should be no language beneath C++”
→ handle all quirks of supported platforms manually

- Object File formats: PE/COFF on Windows, Mach Object on MacOSX x64, ELF on Linux
- No standardized ABI: name mangling

```
// We should never ever see a FunctionNoProtoType at this point.  
// We don't even know how to mangle their types anyway :).
```

[MicrosoftMangle.cpp](#)



C++ vs. a Language Constructed for Live Coding

Ideal unified runtime environment → avoid arbitrary external dependencies

C++ “There should be no language beneath C++”

→ handle all quirks of supported platforms manually

- Object File formats: PE/COFF on Windows, Mach Object on MacOSX x64, ELF on Linux
- No standardized ABI: name mangling, exception handling (Unix signals, Windows SEH/VEH/C++ exceptions)



C++ vs. a Language Constructed for Live Coding

Ideal unified runtime environment → avoid arbitrary external dependencies

C++ “There should be no language beneath C++”

→ handle all quirks of supported platforms manually

- Object File formats: PE/COFF on Windows, Mach Object on MacOSX x64, ELF on Linux
- No standardized ABI: name mangling, exception handling (Unix signals, Windows SEH/VEH/C++ exceptions), calling conventions (Windows `__declspec(dllimport)`)



C++ vs. a Language Constructed for Live Coding

Ideal unified runtime environment → avoid arbitrary external dependencies

C++ “There should be no language beneath C++”
→ handle all quirks of supported platforms manually

- Object File formats: PE/COFF on Windows, Mach Object on MacOSX x64, ELF on Linux
- No standardized ABI: name mangling, exception handling (Unix signals, Windows SEH/VEH/C++ exceptions), calling conventions (Windows `__declspec(dllexport)`, RTTI



C++ vs. a Language Constructed for Live Coding

Ideal unified runtime environment → avoid arbitrary external dependencies

C++ “There should be no language beneath C++”

→ handle all quirks of supported platforms manually

- Object File formats: PE/COFF on Windows, Mach Object on MacOSX x64, ELF on Linux
- No standardized ABI: name mangling, exception handling (Unix signals, Windows SEH/VEH/C++ exceptions), calling conventions (Windows `__stdcall`), RTTI
- Intrinsics for extended instruction sets (SSE, HLSL, etc.)



C++ vs. a Language Constructed for Live Coding

Ideal unified runtime environment → avoid arbitrary external dependencies

C++ “There should be no language beneath C++”

→ handle all quirks of supported platforms manually

- Object File formats: PE/COFF on Windows, Mach Object on MacOSX x64, ELF on Linux
- No standardized ABI: name mangling, exception handling (Unix signals, Windows SEH/VEH/C++ exceptions), calling conventions (Windows `__stdcall`), RTTI
- Intrinsics for extended instruction sets (SSE, HLSL, etc.)
Projucer: ships intrinsics headers for its specific version of Clang



C++ vs. a Language Constructed for Live Coding

Ideal optimize for simplicity, uniformity, portability

C++ optimize for efficiency: “Only pay for what you use”
→ standard encourages compiler vendors to implement
system-specific optimizations for performance reasons



C++ vs. a Language Constructed for Live Coding

Ideal optimize for simplicity, uniformity, portability

C++ optimize for efficiency: “Only pay for what you use”
→ standard encourages compiler vendors to implement
system-specific optimizations for performance reasons

C++ Standard Library:

- standardized syntax & semantics
- implementations and **headers** vary between compilers & platforms



C++ vs. a Language Constructed for Live Coding

Ideal optimize for simplicity, uniformity, portability

C++ optimize for efficiency: “Only pay for what you use”
→ standard encourages compiler vendors to implement
system-specific optimizations for performance reasons

C++ Standard Library:

- standardized syntax & semantics
- implementations and **headers** vary between compilers & platforms

Projucer: Xcode must be installed on Mac

Visual Studio 2015 Update 3 on Windows



C++ vs. a Language Constructed for Live Coding

Ideal strong static typing → allow runtime state restoration

C++ “No implicit violations of the type system, but allow explicit violations”



C++ vs. a Language Constructed for Live Coding

Ideal strong static typing → allow runtime state restoration

C++ “No implicit violations of the type system, but allow explicit violations”

Projucer: no runtime state restoration



C++ vs. a Language Constructed for Live Coding

Ideal will (most likely) never be adapted

C++ used a lot by real people today and for a long time to come



All these quirks make it hard!

hard \neq impossible



All these quirks make it hard!

hard \neq impossible

It needs some really good tools.



LLVM and Clang come to rescue!

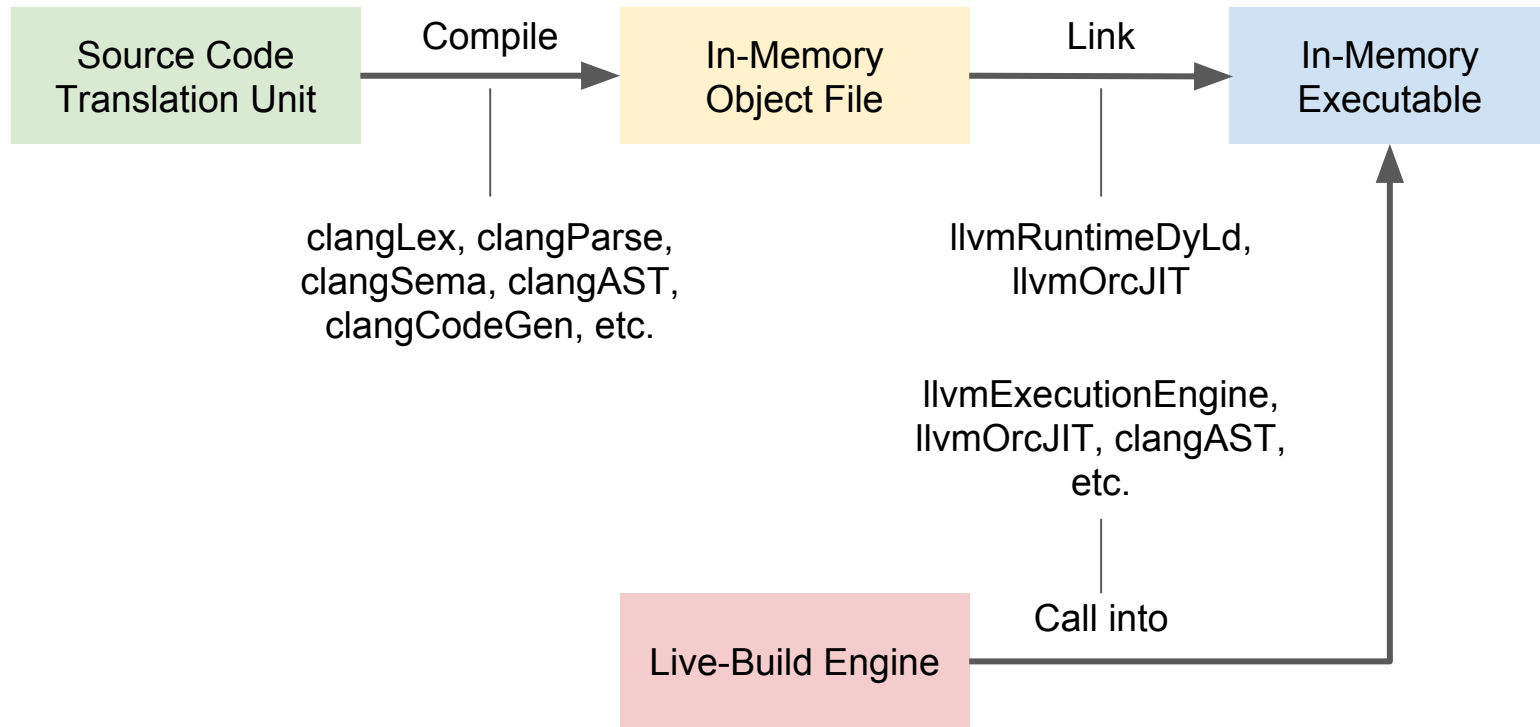
- LLVM: compiler infrastructure
 - Intermediate Representation (LLVM IR)
 - set of tools and libraries to work with it
- Clang: C-language family frontend for LLVM





Behind the Scenes of the Projucer C++ Live-Build Engine

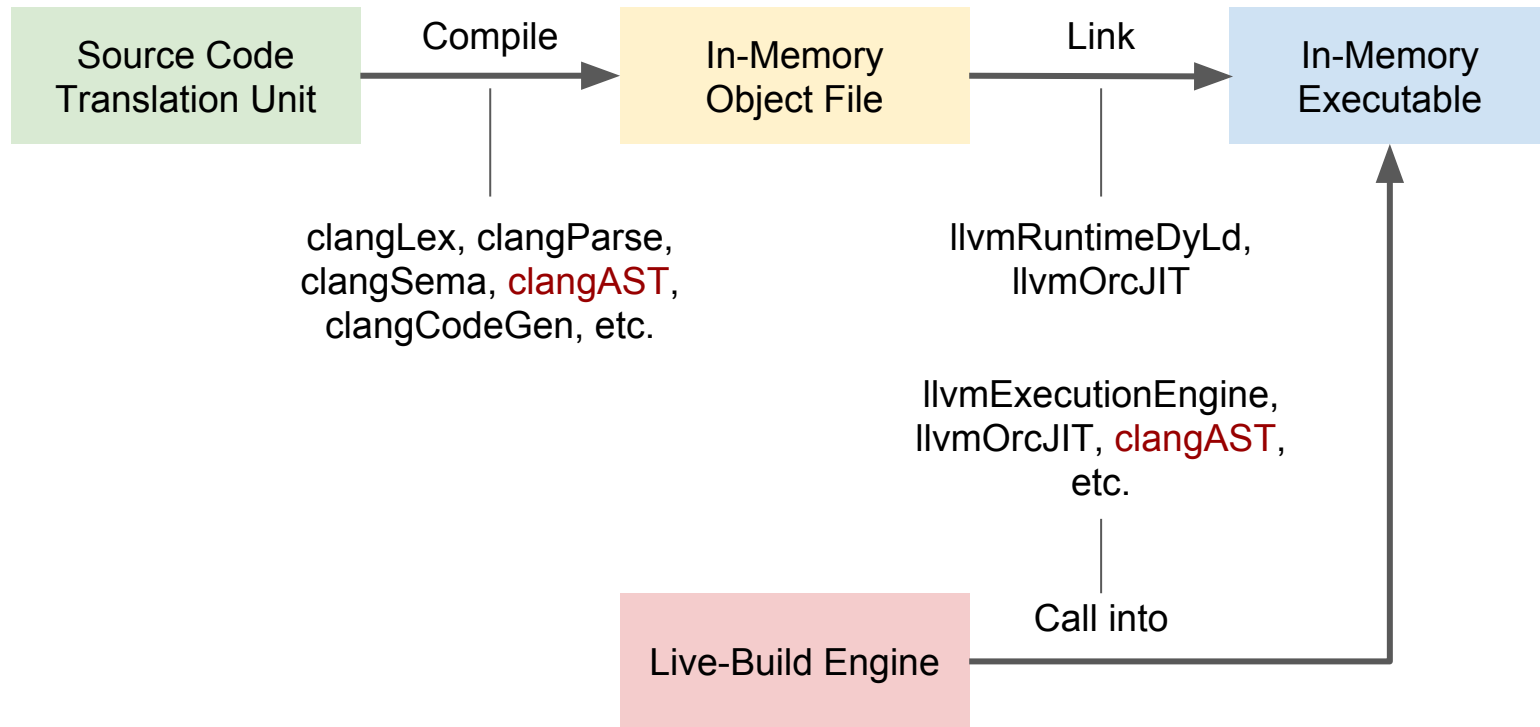
Live builds – High-level overview





Behind the Scenes of the Projucer C++ Live-Build Engine

Live builds – High-level overview





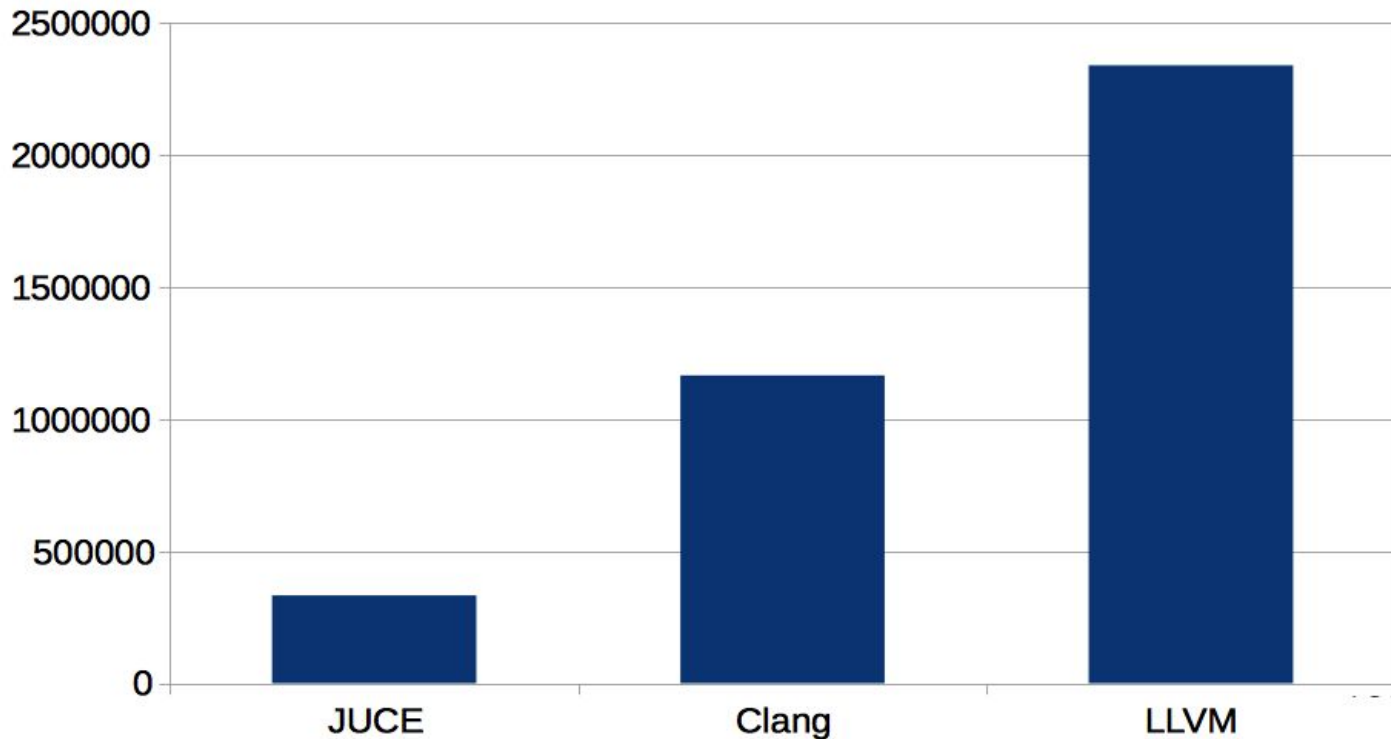
Still: There be dragons

- LLVM & Clang are moving targets
- Upstream LLVM in numbers:
 - 30 commits per day on average
 - contributions from ~100 developers each month
 - Google, Intel, Microsoft, Qualcomm, AMD, ...
- Maintenance cost for customizations is enormous
- 2015 DevMeeting: [Living Downstream Without Drowning](#)



Behind the Scenes of the Projucer C++ Live-Build Engine

lines of code (measured with [cloc](#))

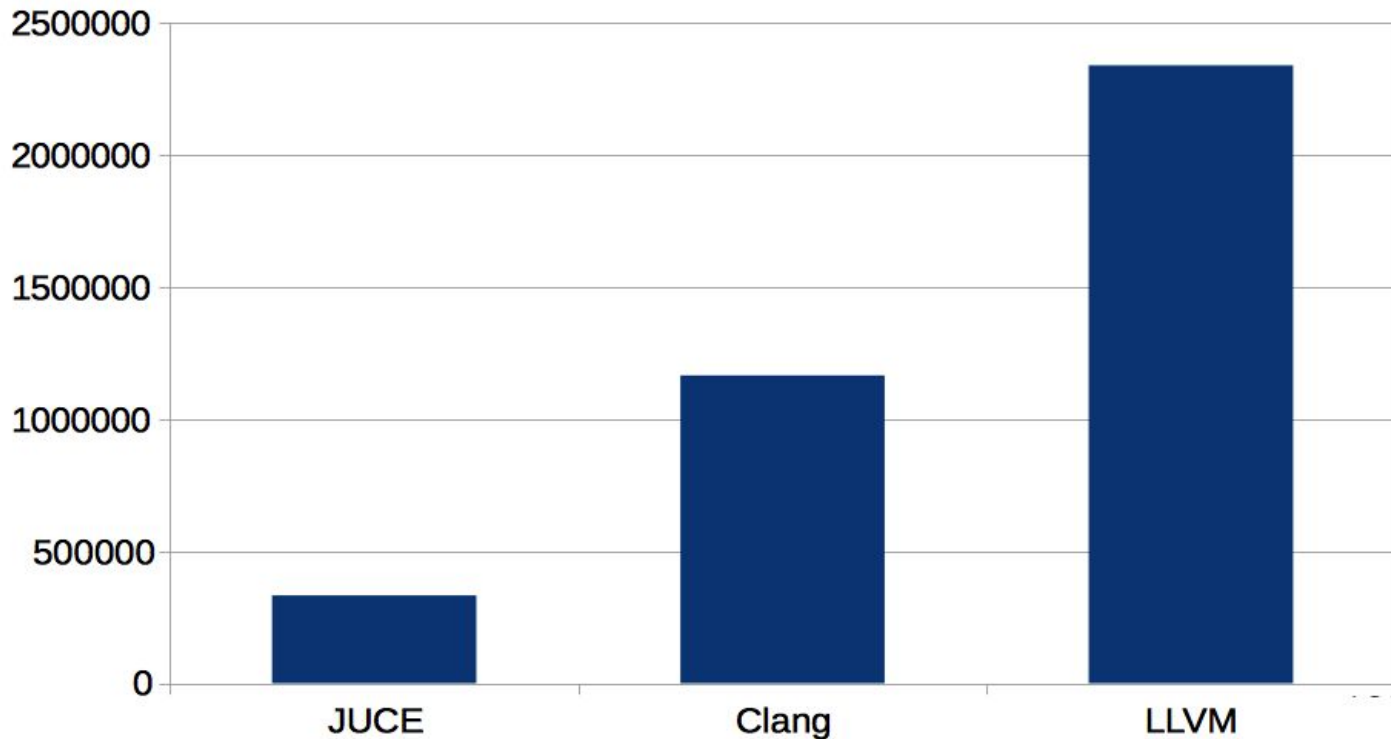




Behind the Scenes of the Projucer C++ Live-Build Engine

Where's the lines that cause our bug?

lines of code





Blocker bug in the live build engine on Windows x64:

- Symptom: immediate crash when launching any live component preview
- Assertion failed while linking:
`((int64_t)Result <= INT32_MAX) && "Relocation overflow"`
file: RuntimeDyldCOFFX86_64.h, line 81



Behind the Scenes of the Projucer C++ Live-Build Engine

A brief discourse on linking



A brief discourse on linking

Source Code Translation Units

foo.cpp

```
#include "bar.h"  
void foo() { bar(25); }
```

bar.cpp

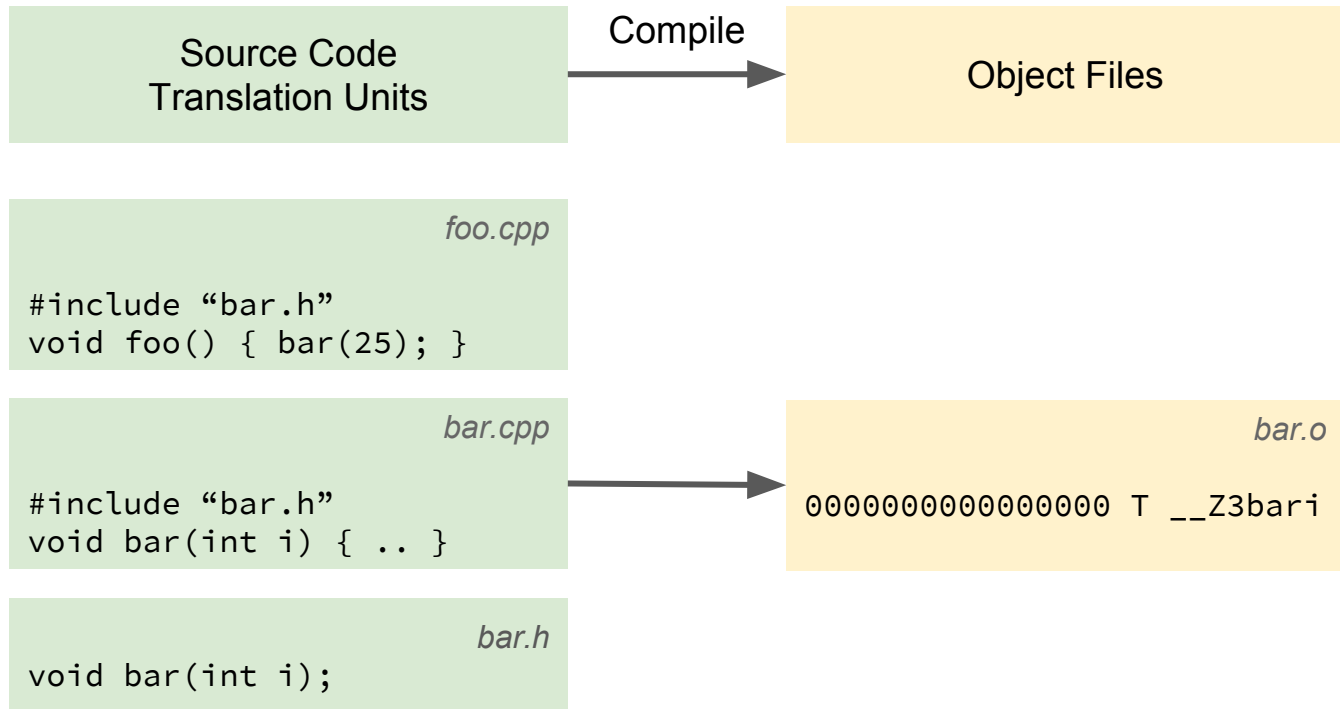
```
#include "bar.h"  
void bar(int i) { .. }
```

bar.h

```
void bar(int i);
```



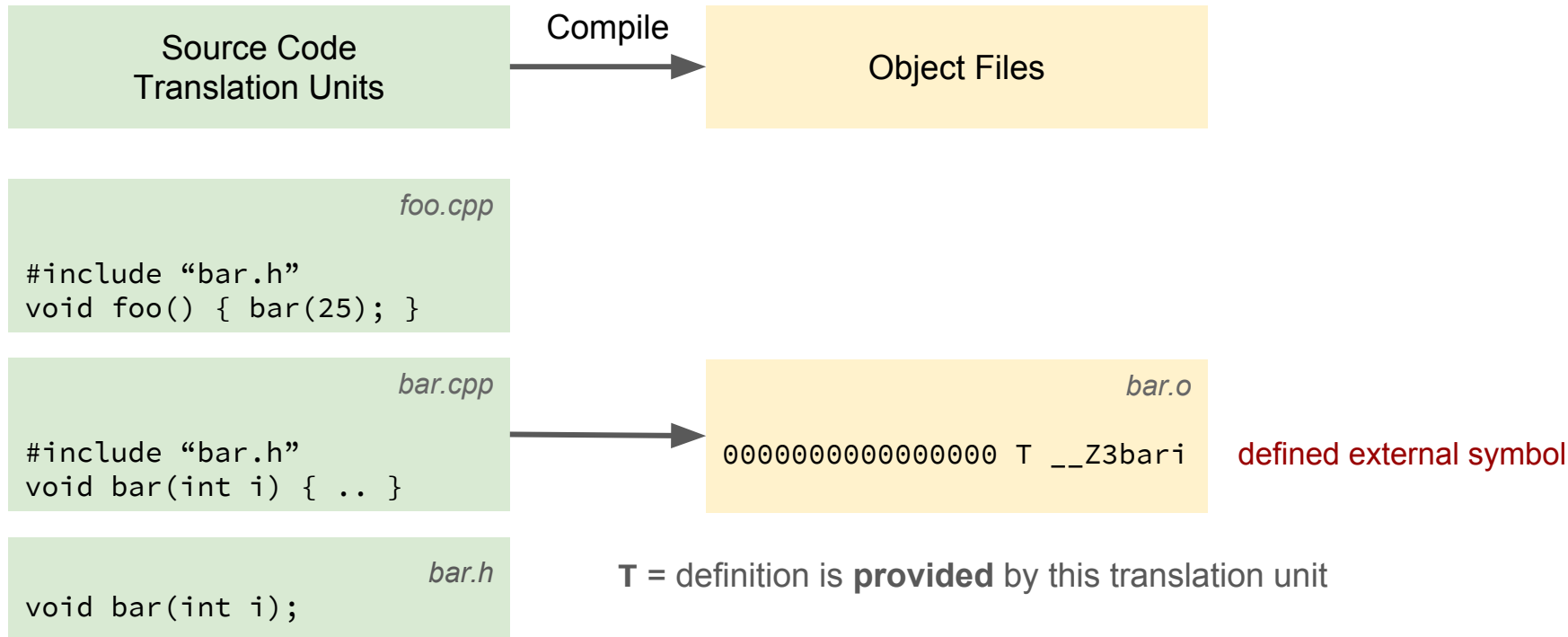
A brief discourse on linking





Behind the Scenes of the Projucer C++ Live-Build Engine

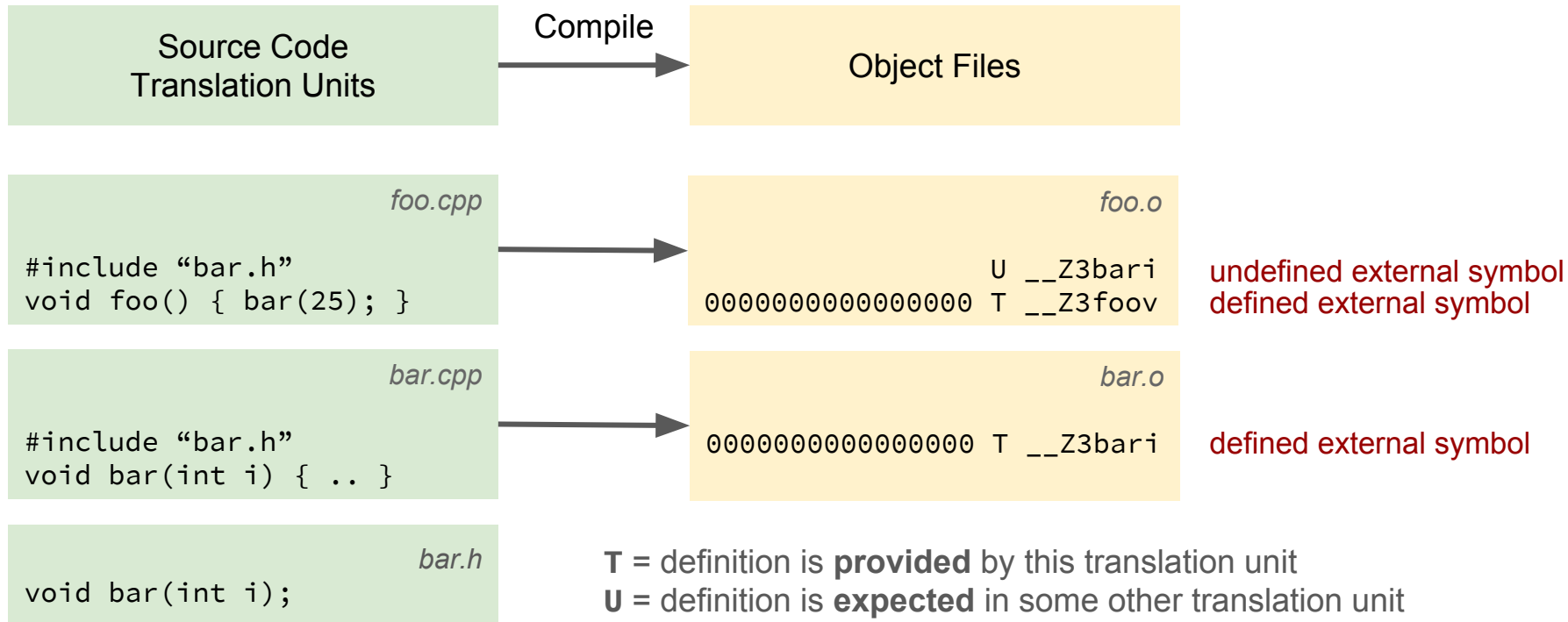
A brief discourse on linking





Behind the Scenes of the Projucer C++ Live-Build Engine

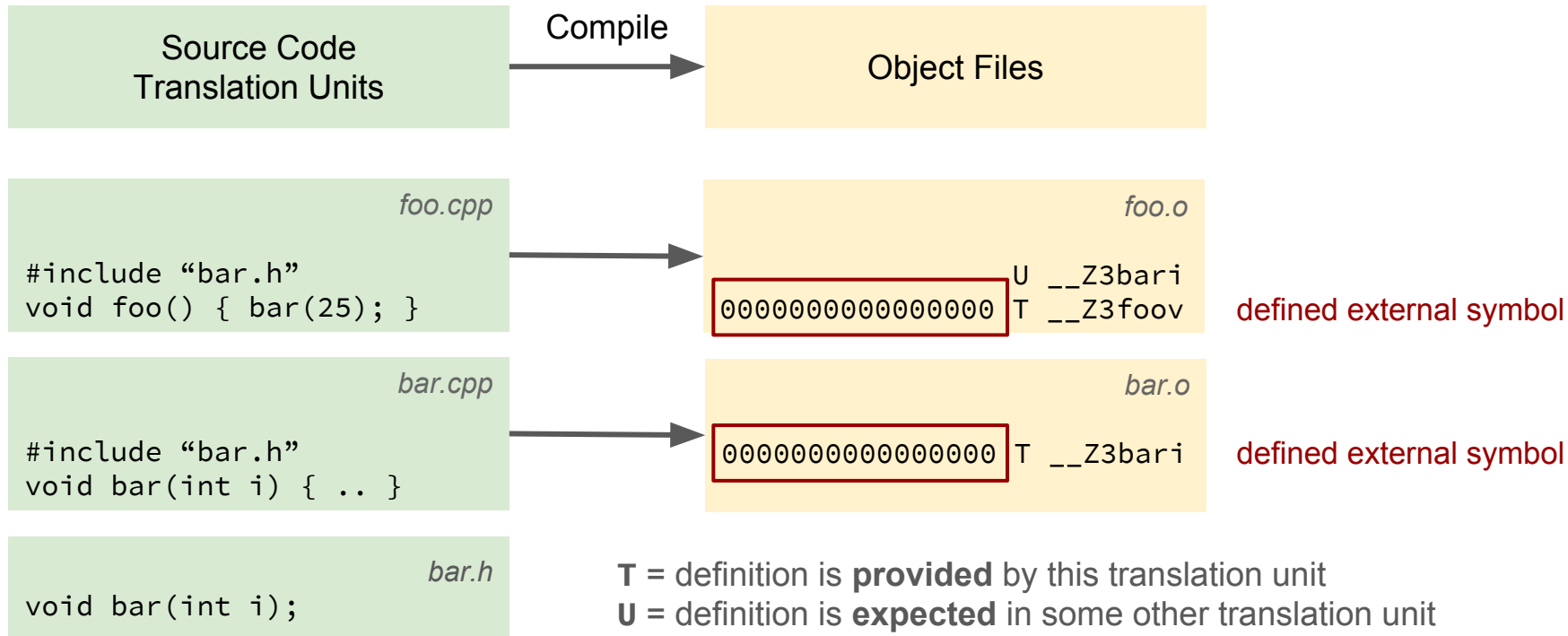
A brief discourse on linking





Behind the Scenes of the Projucer C++ Live-Build Engine

A brief discourse on linking





A brief discourse on linking

Object Files

foo.o

```
U __Z3bari
000000000000000000 T __Z3foov
```

bar.o

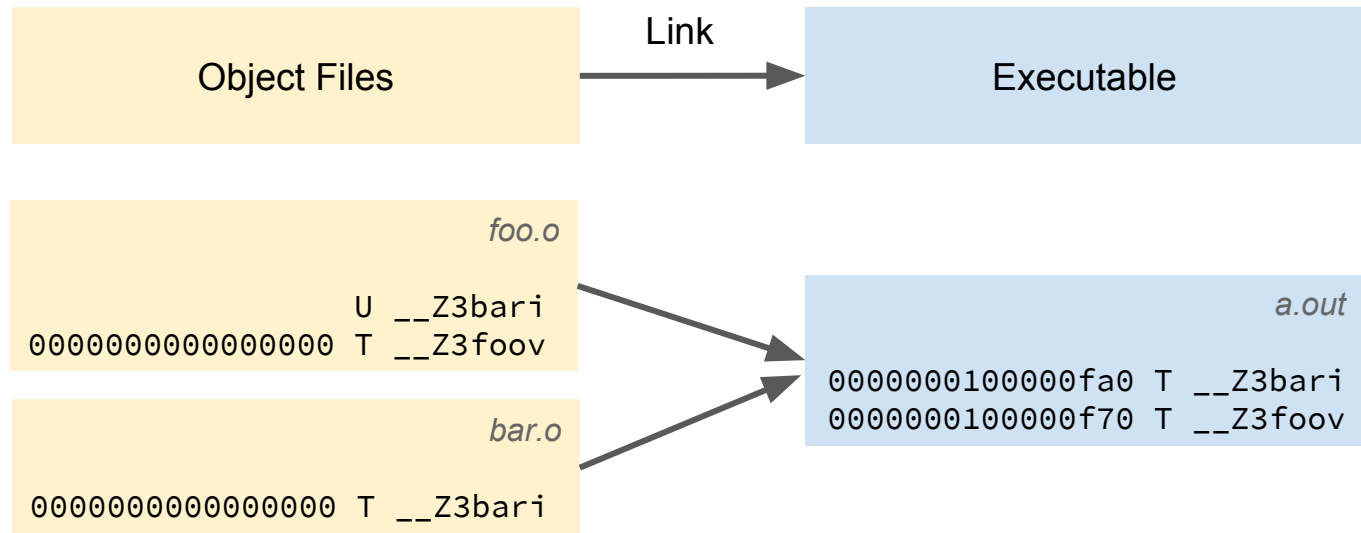
```
000000000000000000 T __Z3bari
```

T = definition is **provided** by this translation unit

U = definition is **expected** in some other translation unit



A brief discourse on linking



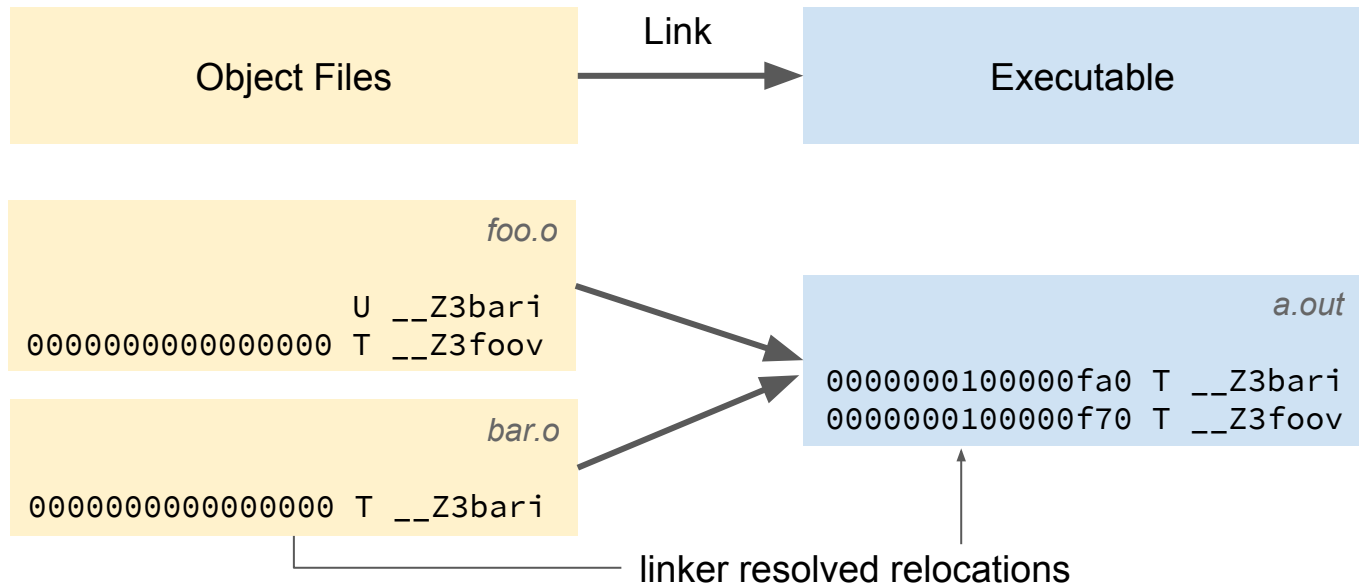
T = definition is **provided** by this translation unit

U = definition is **expected** in some other translation unit



Behind the Scenes of the Projucer C++ Live-Build Engine

A brief discourse on linking



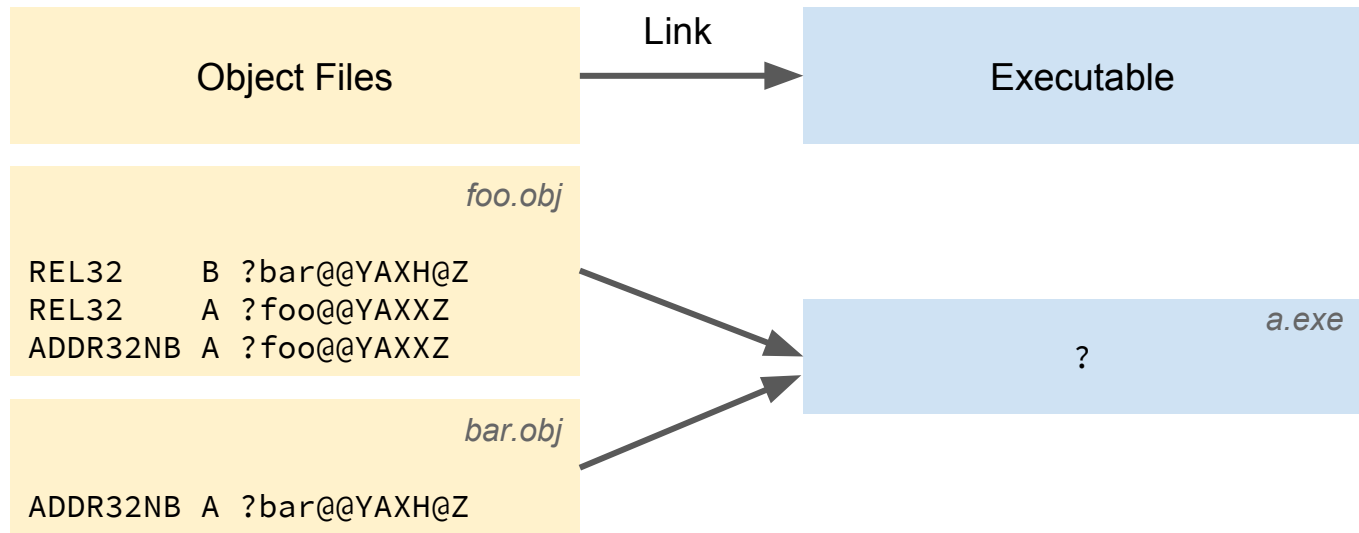
T = definition is **provided** by this translation unit

U = definition is **expected** in some other translation unit



Behind the Scenes of the Projucer C++ Live-Build Engine

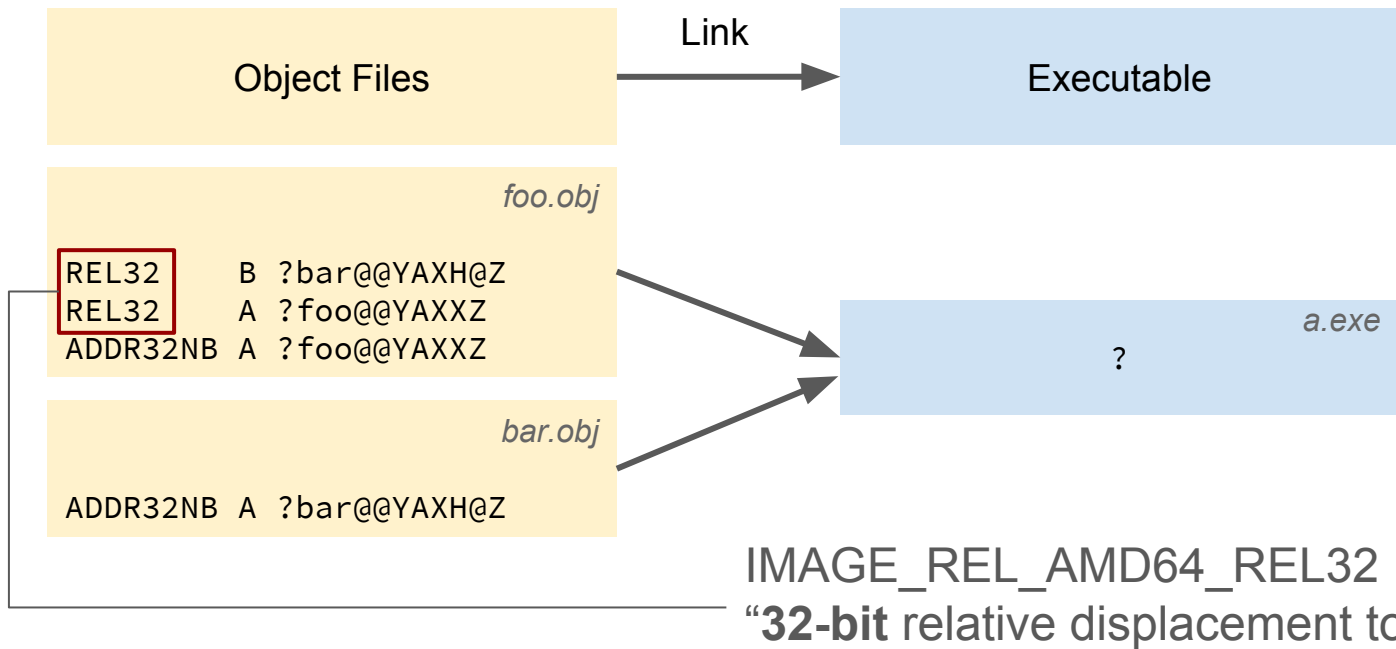
A brief discourse on linking – Clang COFF Objects Windows x64





Behind the Scenes of the Projucer C++ Live-Build Engine

A brief discourse on linking – Clang COFF Objects Windows x64



[Microsoft PE/COFF Specification](#)



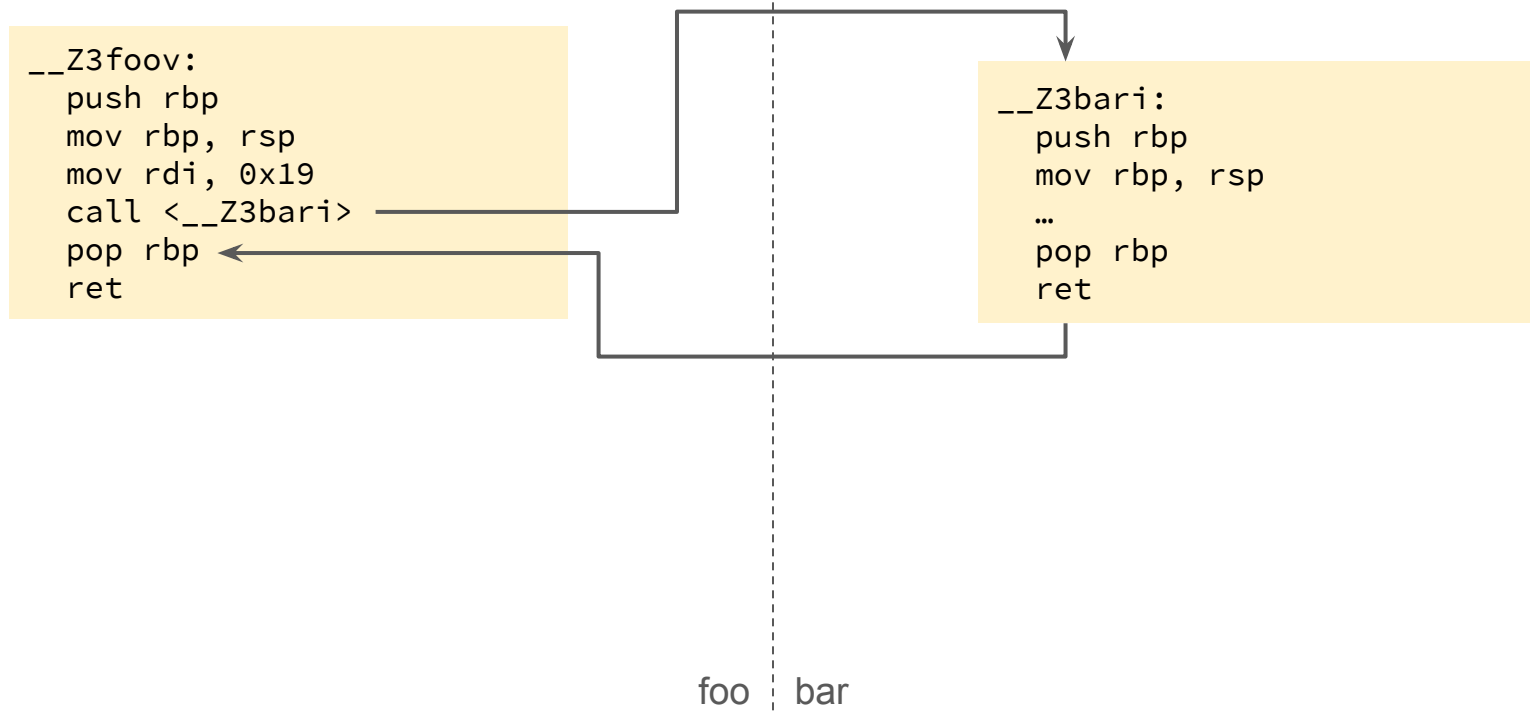
A brief discourse on linking

```
__Z3foov:  
  push rbp  
  mov rbp, rsp  
  mov rdi, 0x19  
  call <__Z3bari>  
  pop rbp  
  ret
```



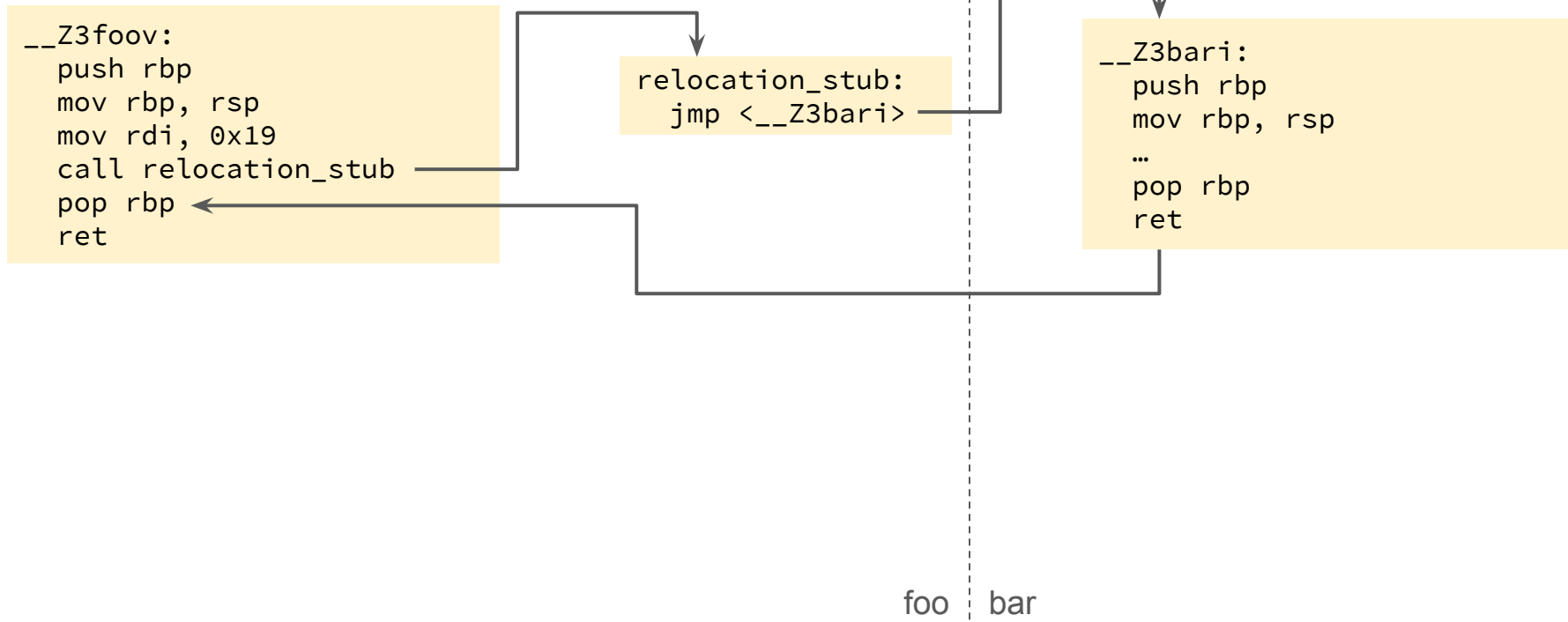
Behind the Scenes of the Projucer C++ Live-Build Engine

A brief discourse on linking (no PIC)



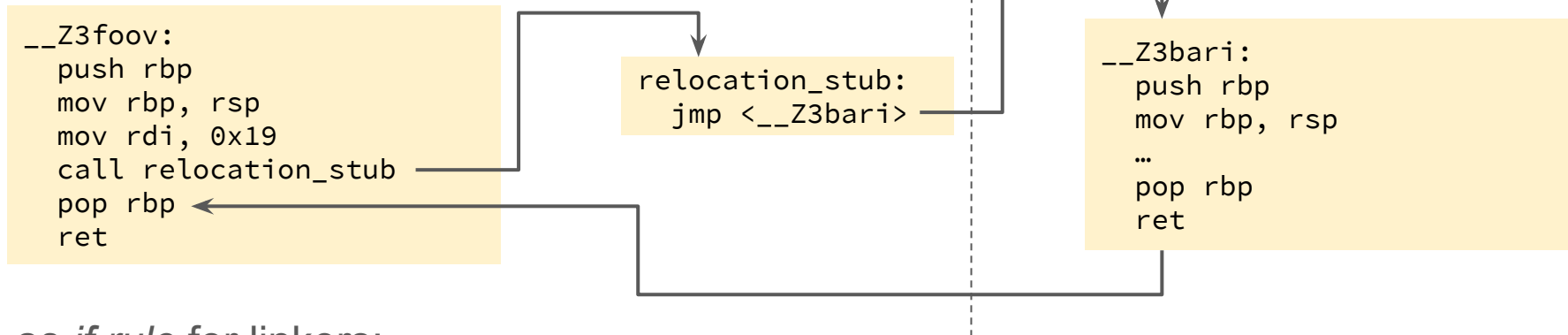


A brief discourse on linking (PIC)





A brief discourse on linking (PIC)

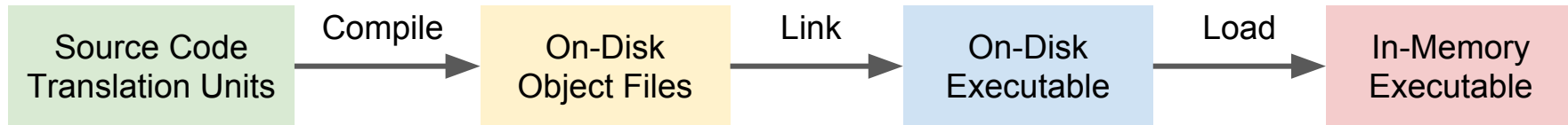


as-if rule for linkers:

“The linker is responsible for creating [...] stub functions and lazy pointers [...] for calls to another linkage unit. Since **the linker** must create these entries, it **can also choose not to create them when it sees the opportunity.**”

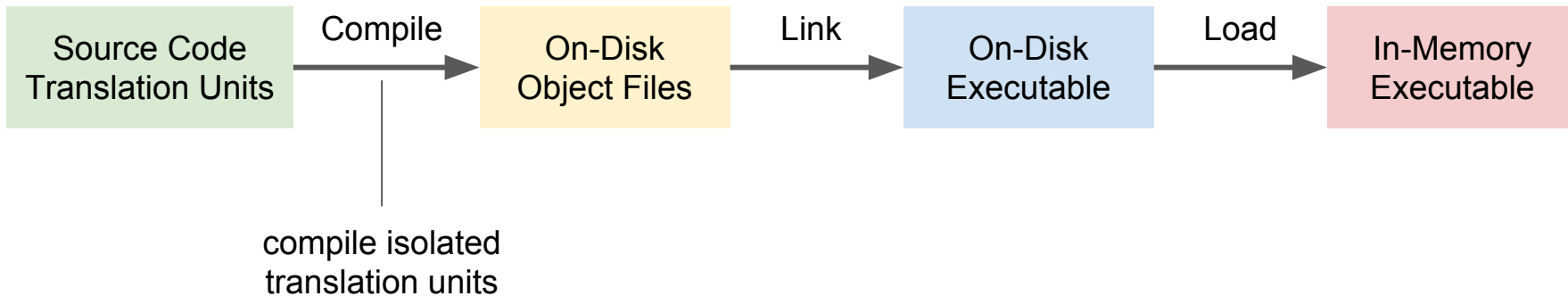


Static Builds – Phases



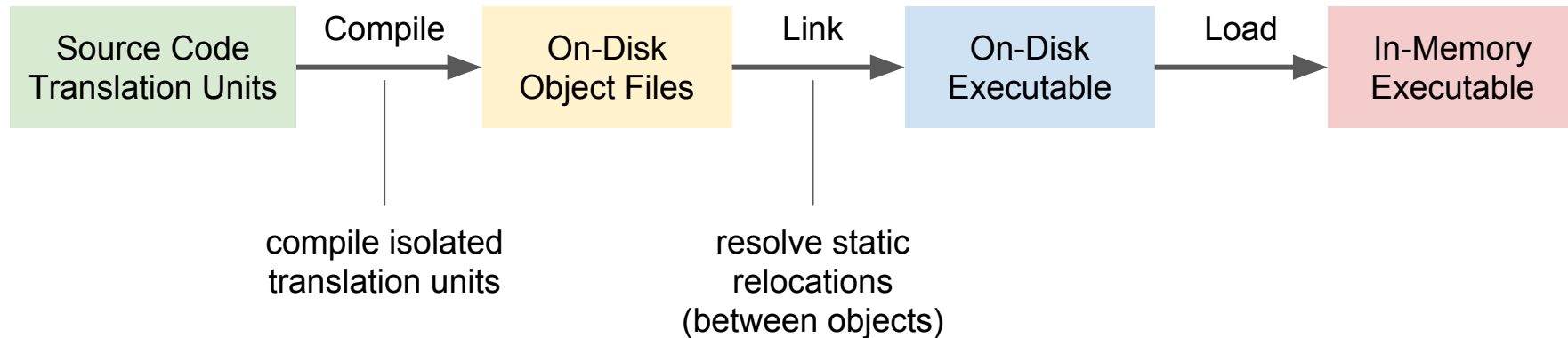


Static Builds – Phases



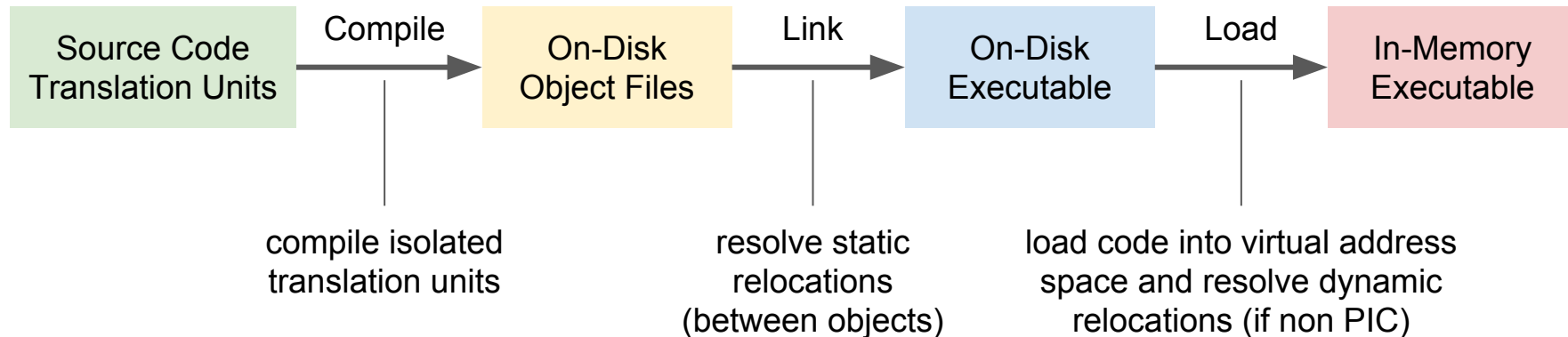


Static Builds – Phases



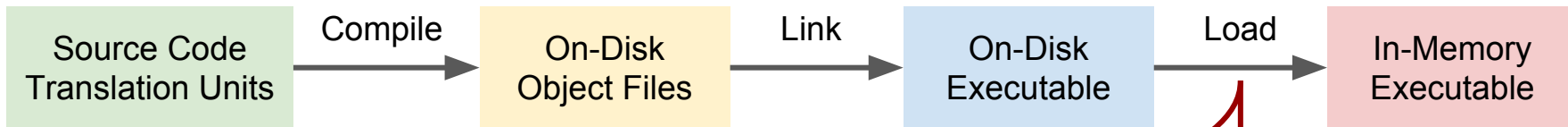


Static Builds – Phases

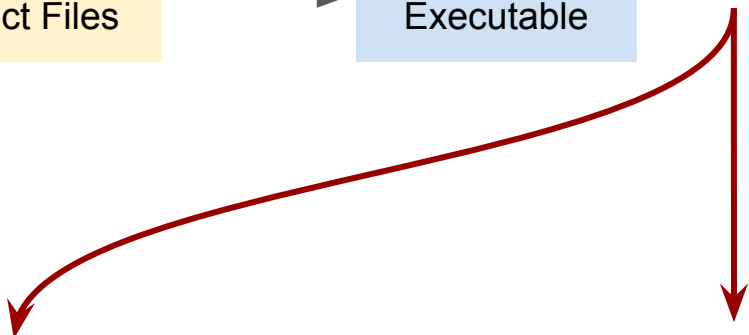




Static Builds – Phases

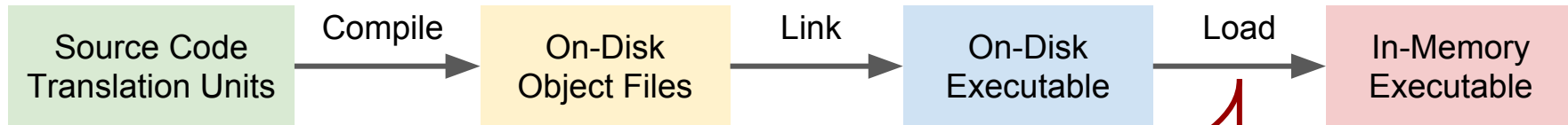


Live Builds – Phases





Static Builds – Phases



Live Builds – Phases

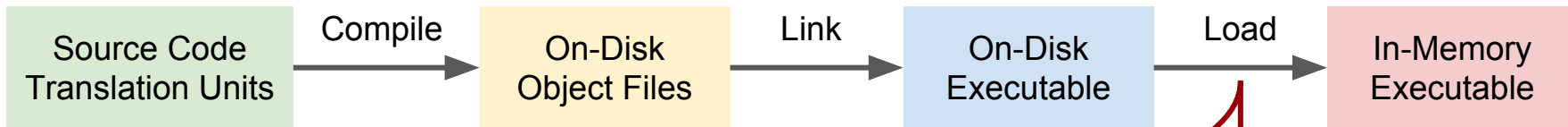


compile isolated translation
units in memory

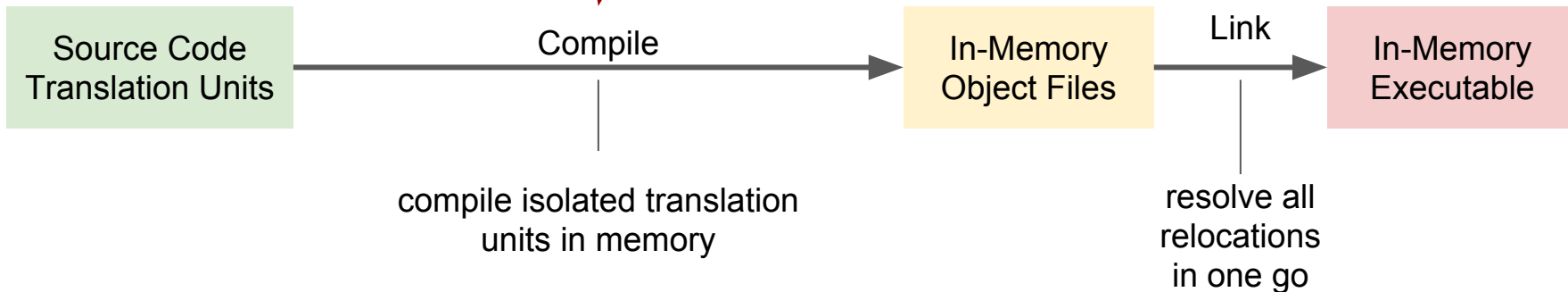


Behind the Scenes of the Projucer C++ Live-Build Engine

Static Builds – Phases



Live Builds – Phases





Static Builds: Link-Time Virtual Memory x64





Static Builds: Link-Time Virtual Memory x64





Static Builds: Link-Time Virtual Memory x64



Live Builds: Link-Time Virtual Memory x64





Static Builds: Link-Time Virtual Memory x64

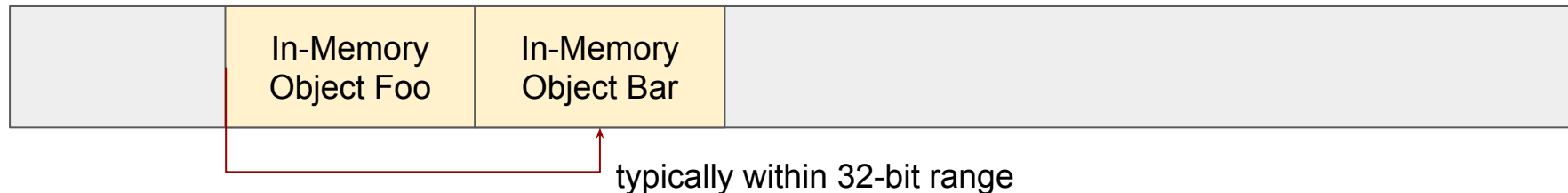


Live Builds: Link-Time Virtual Memory x64





Static Builds: Link-Time Virtual Memory x64

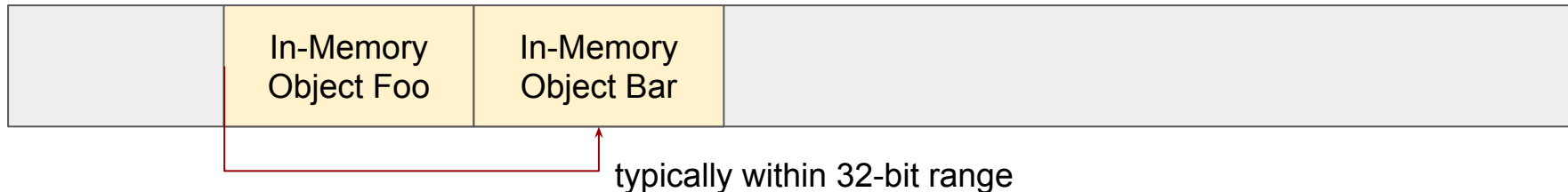


Live Builds: Link-Time Virtual Memory x64

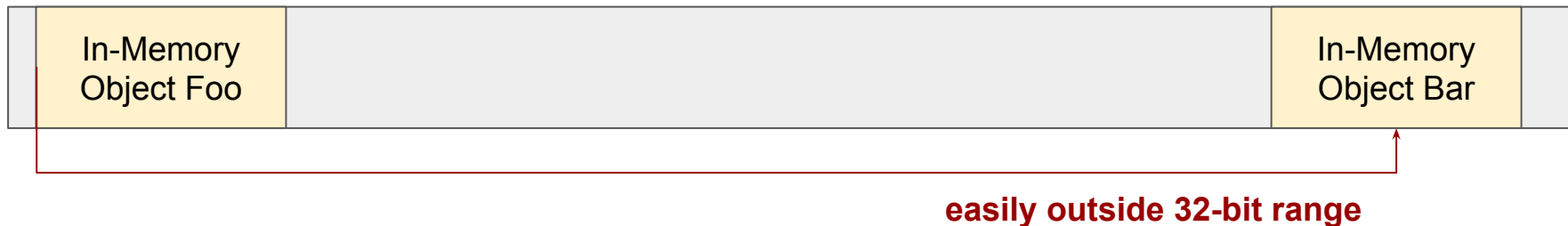




Static Builds: Link-Time Virtual Memory x64

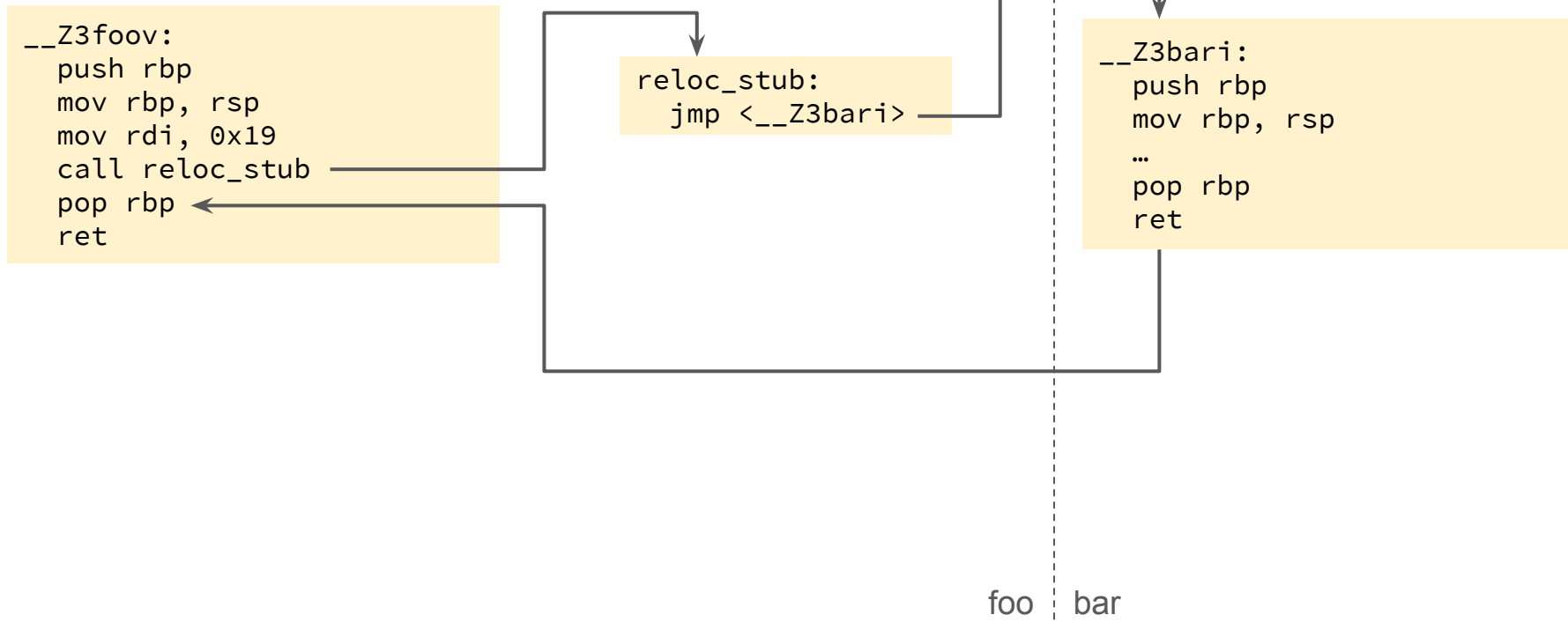


Live Builds: Link-Time Virtual Memory x64



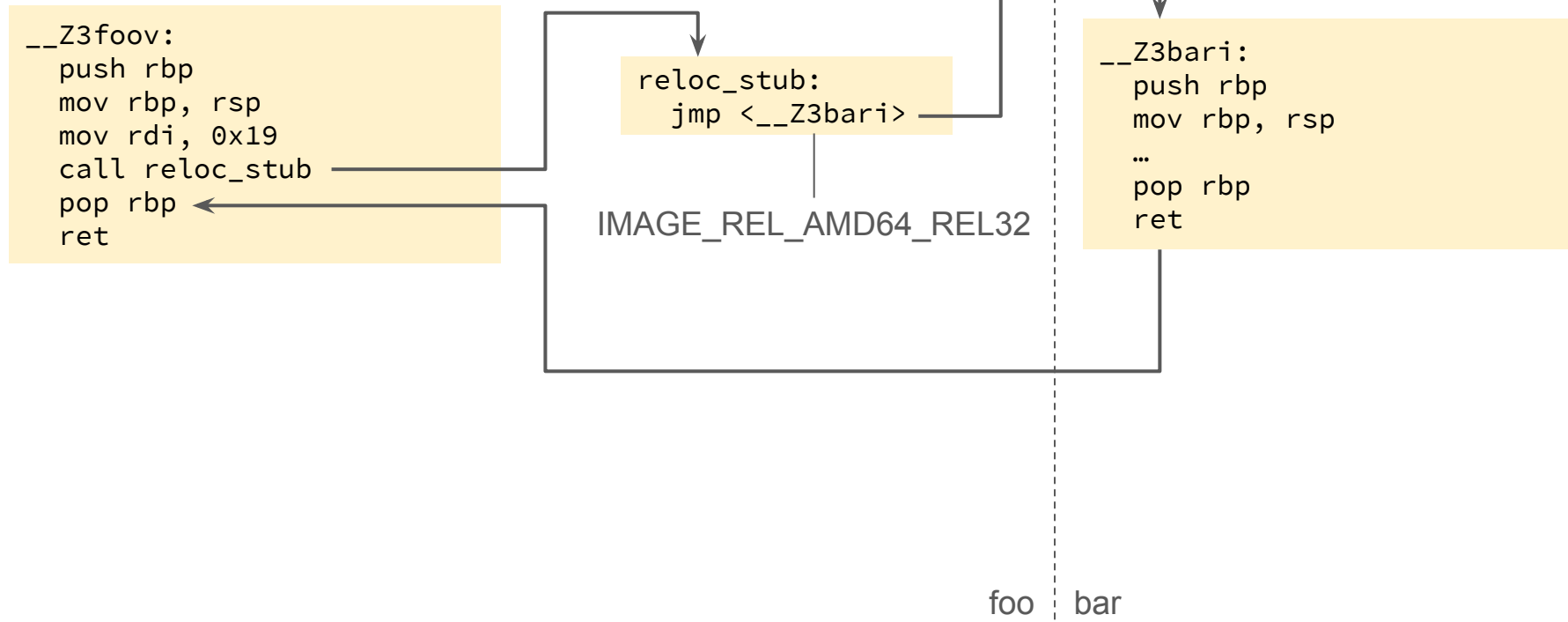


Fixing the Relocation Bug



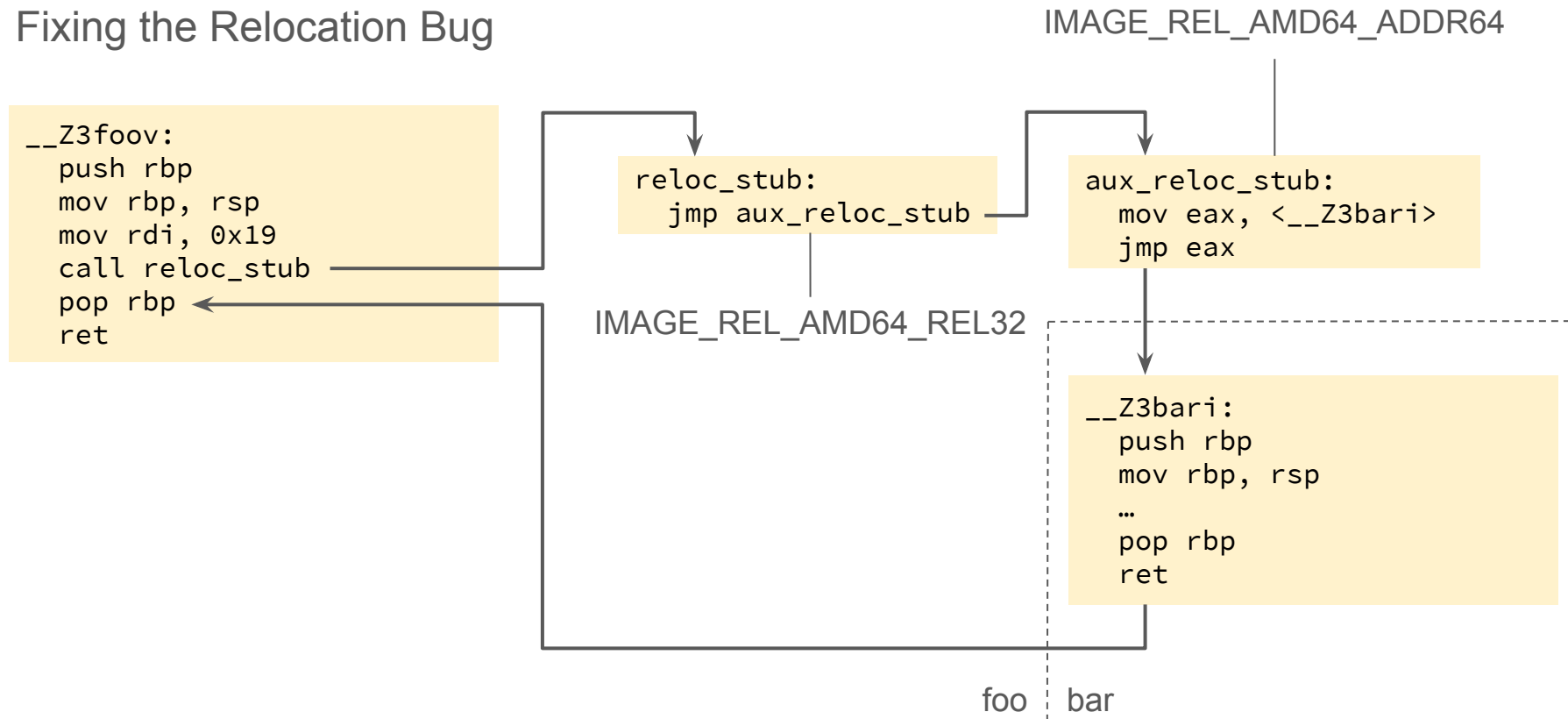


Fixing the Relocation Bug



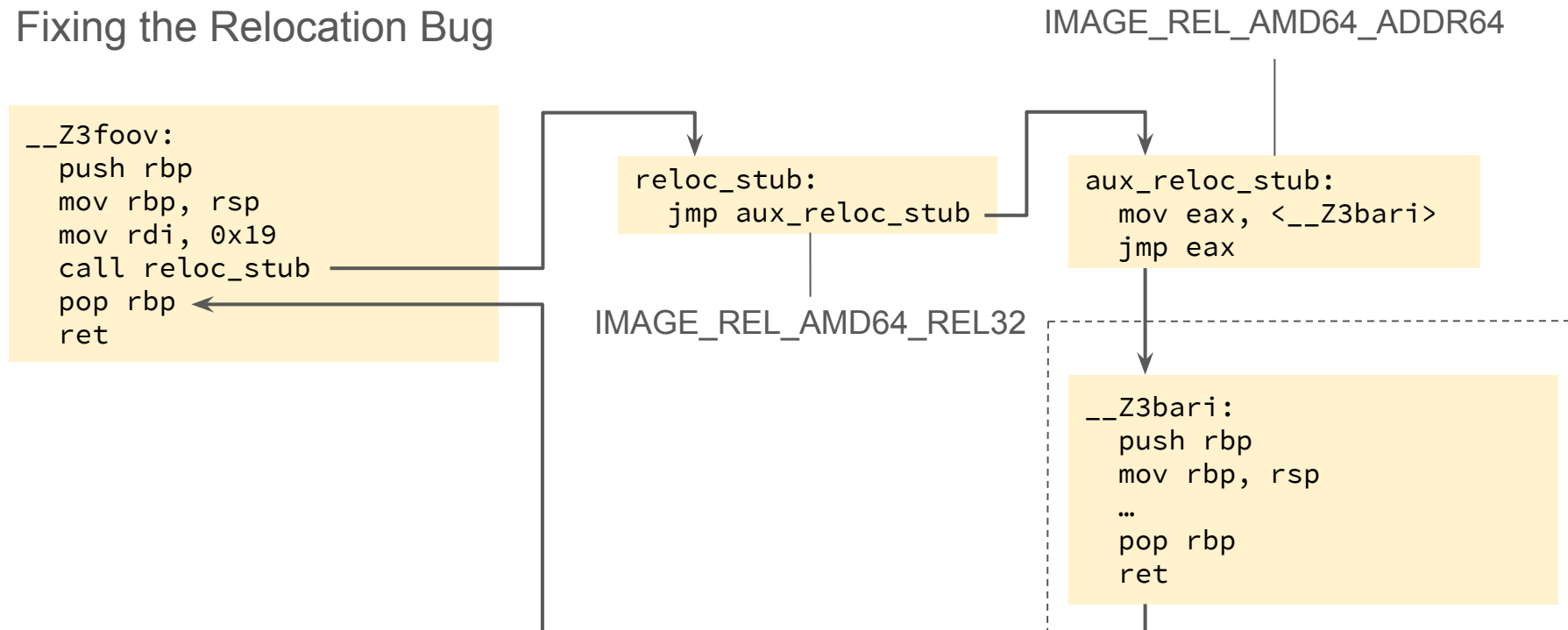


Fixing the Relocation Bug





Fixing the Relocation Bug



Patch Proposal

<https://github.com/weliveindetail/pj-llvm/commit/f9f26dc8bf511dde02142dc2cf361f67b4964985>



Eli Bendersky about ELF:

- Position Independent Code (PIC) in shared libraries
<http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/>
- Load-time relocation of shared libraries
<http://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries/>

Mach-O Programming Topics:

- Position-Independent Code
https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/dynamic_code.html
- x86-64 Code Model
https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/x86_64_code.html

Microsoft PE/COFF:

- A Tour of the Win32 Portable Executable File Format (1994)
<https://msdn.microsoft.com/en-us/library/ms809762.aspx>
- Microsoft PE/COFF Specification (2015)
http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff_v83.docx



Thanks for your attention.

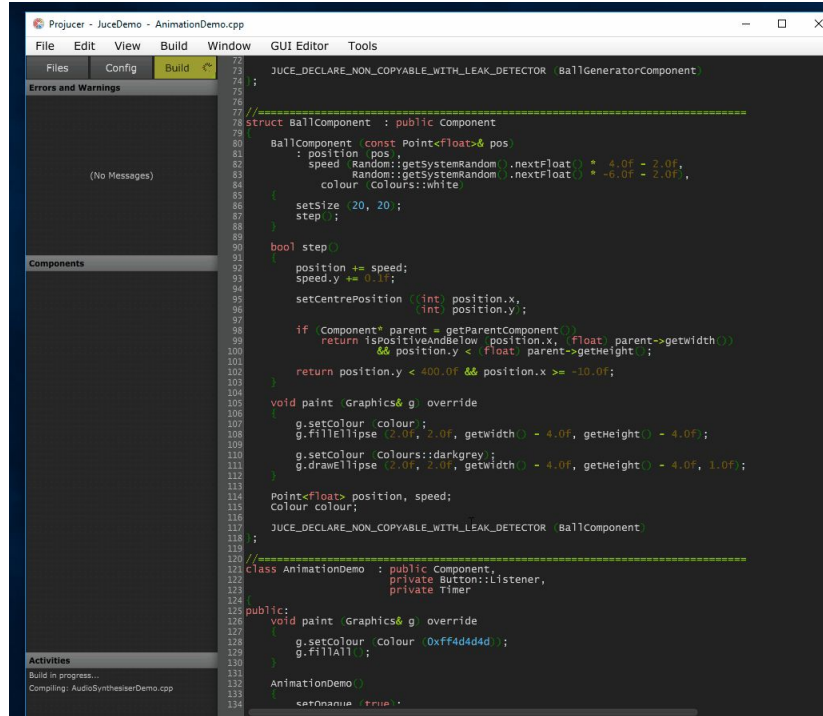


Questions?



Recent additions

Platform support: Windows



```
72
73
74 ; JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR BallGeneratorComponent;
75
76
77 //-----
78 struct BallComponent : public Component
79 {
80     BallComponent (const Point<float>& pos)
81     : position (pos),
82       speed (Random::getSystemRandom().nextFloat() * 4.0f - 2.0f,
83            Random::getSystemRandom().nextFloat() * -6.0f - 2.0f),
84       colour (Colours::white)
85     {
86         setSize (20, 20);
87         step();
88     }
89
90     bool step()
91     {
92         position += speed;
93         speed.y += 0.1f;
94
95         setCentrePosition ((int) position.x,
96                           (int) position.y);
97
98         if (Component* parent = getParentComponent())
99             return isPositiveAndBelow (position.x, float (parent->getWidth()),
100                                     && position.y < float (parent->getHeight()));
101         return position.y < 400.0f && position.x >= -10.0f;
102     }
103
104     void paint (Graphics& g) override
105     {
106         g.setColour (colour);
107         g.fillRect (2.0f, 2.0f, getWidth() - 4.0f, getHeight() - 4.0f);
108         g.setColour (Colours::darkgrey);
109         g.drawEllipse (2.0f, 2.0f, getWidth() - 4.0f, getHeight() - 4.0f, 1.0f);
110     }
111
112     Point<float> position, speed;
113     Colour colour;
114 };
115
116 JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR BallComponent;
117
118 //-----
119
120
121 class AnimationDemo : public Component,
122                      private Button::Listener,
123                      private Timer
124 {
125 public:
126     void paint (Graphics& g) override
127     {
128         g.setColour (Colour (0xFF4d4d));
129         g.fillRect (0, 0, getWidth(), getHeight());
130     }
131
132     AnimationDemo()
133     {
134         setLookAndFeel (*true);
135     }
136 }
```





Recent additions

Usability feature: PCH error recovery

The screenshot shows the Projucer IDE window titled "Projucer - PluckedStringsDemo - StringDemoComponent.h". The interface includes a top bar with "Files", "Config", and "Build" tabs. On the left, there are three panels: "Errors and Warnings", "Components", and "Activities".

The "Errors and Warnings" panel displays a yellow warning icon and the message: "Detected corrupt PCH file. Trying to regenerate it for you. If this issue remains please try 'Clean all'".

The "Components" panel is currently empty.

The "Activities" panel shows "Build in progress..." and "Compiling: Main.cpp".

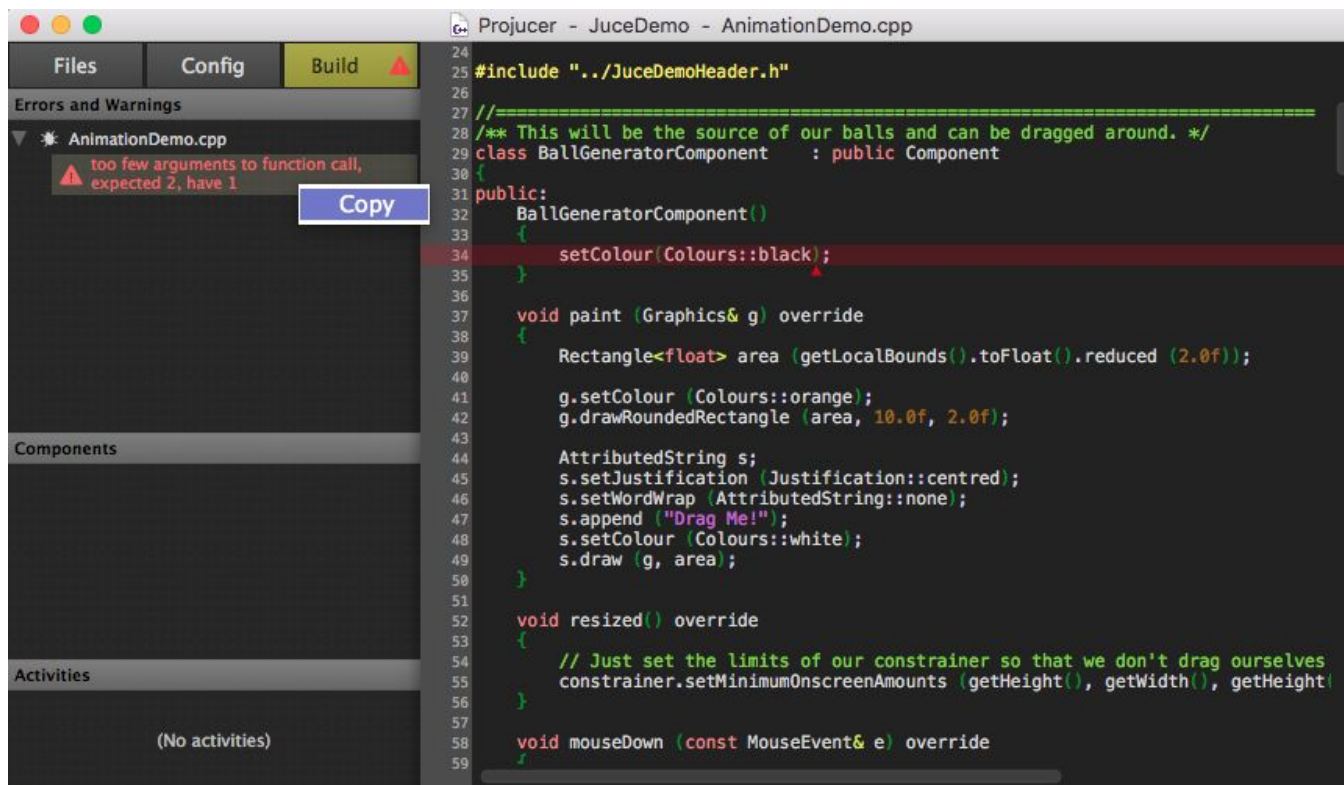
The main editor area displays C++ code from "StringDemoComponent.h". The code includes a function definition for `StringParameters` and a `createStringComponents()` function. The code is as follows:

```
97     {
98         float position = (e.position.x - stringLine->getX()) / stringLine->getLength();
99
100         stringLine->stringPlucked (position);
101         stringSynths.getUnchecked(i)->stringPlucked (position);
102     }
103 }
104
105 //=====
106 struct StringParameters
107 {
108     StringParameters (int midiNote)
109         : frequencyInHz (MidiMessage::getMidiNoteInHertz (midiNote)),
110           lengthInPixels ((int) (760 / (frequencyInHz / MidiMessage::getMidiNoteInHertz (midiNote))))
111     {
112     }
113
114     double frequencyInHz;
115     int lengthInPixels;
116 };
117
118 static std::vector<StringParameters> getDefaultStringParameters()
119 {
120     return { 42, 44, 46, 49, 51, 54, 56, 58, 61, 63, 66, 68, 70 };
121 }
122
123 void createStringComponents()
124 {
125     for (auto stringParams : getDefaultStringParameters())
126     {
127         stringLines.add (new StringComponent (stringParams.lengthInPixels,
128                                             Colour::fromHSV (Random().nextFloat(),
129                                                             1, 1, 1)));
130     }
131 }
132
```



Recent additions

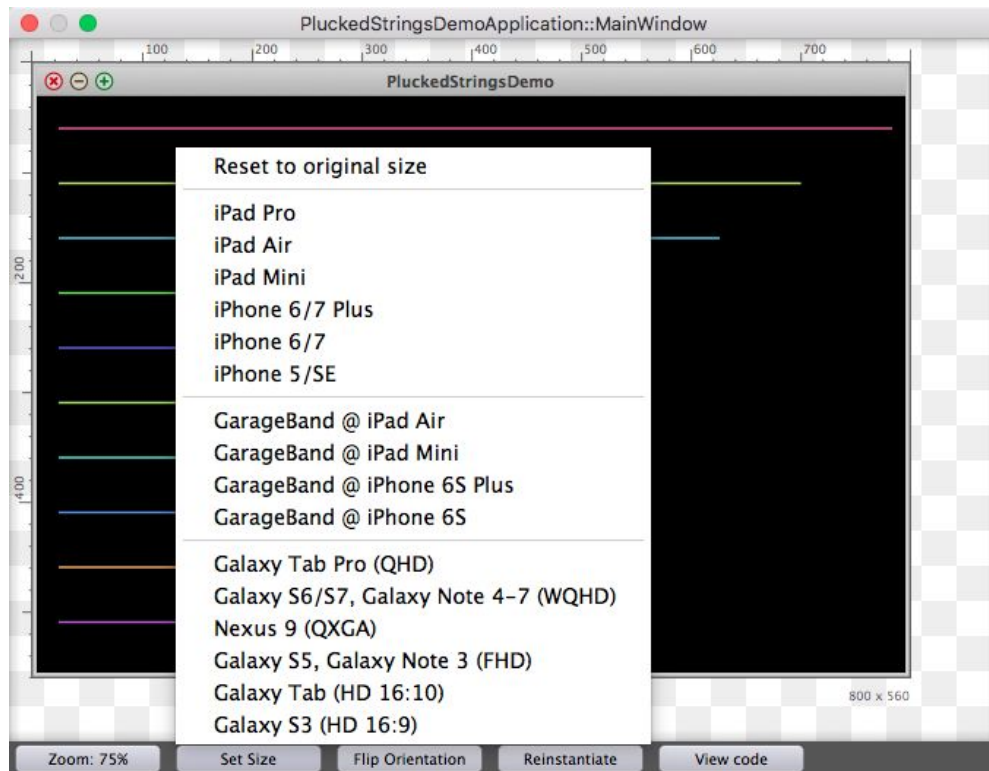
Usability feature: copy diagnostics





Recent additions

Usability feature: common device sizes





And a lot more:

- live preview windows pop-up in foreground
- check for code changes on focus preview window
- fixed drag'n'drop in live previews on Mac
- fixed keyboard input in live previews on Mac
- finally fixed “Open recent” menu on Mac
- symbol resolution entirely from linked libraries and frameworks
- use Clang mangler for all C++ ABI calls
- improved link-times
- ...