WL101.1d

Compilation In-depth

Outline

- Scope Resolution Operator
- One Definition Rule
- Global Separated Compilation Model
- Class Separated Compilation Model
- Static Initialization Problem
- Build Settings and Compiler Optimizations
- Static Libraries
- Name Mangling
- Dynamic Libraries

Scope Resolution Operator

::

- Unlike Java where `.` also resolves scope.
- Resolves the scope of identifiers.
- Works for namespace and class members.
- If there's nothing before `::`, it resolves on the ***global*** scope. (Example later)

Trace the Program

```cpp
int v;
void f();

struct S {
 int v;
 static int u;
 void f();
};

void S::f() {
 v = ::v + u;
}
```

```cpp
void f() {
 int u = v;
 v = S::u;
 S::u = u;
}

int main() {
 v = 10;
 f();
 v = 20;
 S s;
 s.f();
 return s.v;
}
```

One Definition Rule

- Throughout the program, a symbol should be defined once and only once.
- There can be multiple declarations as long as they are the same.
- Easy for one file, but harder to track if you have a lot of files (or if you use someone else's code).
- This is why namespaces are created.

ODR Example

- Create 2 files `a.cpp` and `b.cpp`

```cpp
int main() {
 return 0;
}
```

a.cpp

```cpp
int main() {
 return 0;
}
```

b.cpp

- Compile them separately.
- Try compiling them together.

```
g++ -o prog a.cpp b.cpp
```

Separated C.M.

Keeping Things Separated

Global Symbols

- Create and compile `a.cpp`

```
int foo() {
 return 42;
}

int main() {
 return foo();
}
```

Global Symbols

- Split `a.cpp` into the following:

```cpp
int foo() {
 return 42;
}
```

`b.cpp`

```cpp
int main() {
 return foo();
}
```

`a.cpp`

- Try compiling them together.

```
g++ -o prog a.cpp b.cpp
```

WWW (What Went Wrong?)

Global Symbols

- Modify `a.cpp`:

```cpp
int foo() {
 return 42;
}
```

b.cpp

```cpp
int foo();
int main() {
 return foo();
}
```

a.cpp

- Try compiling them together.

```
g++ -o prog a.cpp b.cpp
```

Global Symbols

- Modify and recompile:

```cpp
int foo(int b);

int bar(int b) {
 return b * foo(--b);
}

int foo(int b) {
 if (b > 0) return bar(b);
 return 1;
}
```

b.cpp

```cpp
int foo(int);
int main() {
 return foo(3);
}
```

a.cpp

Global Symbols

- Change a.cpp and recompile:

```
int foo(int b);

int bar(int b) {
 return b * foo(--b);
}

int foo(int b) {
 if (b > 0) return bar(b);
 return 1;
}
```

b.cpp

```
extern int foo
(int);
int main() {
 return foo(3);
}
```

a.cpp

Global Symbols

- The underlying variable, class and function behind an identifier is called a ***symbol***.
- A *Compilation Unit* is a logical unit that a compiler processes. In Java, it is a .java file. In C++, it's a non-header file (.c, .cpp, etc.).
- Symbol visibility don't automatically carry across compilation units (but they still exist).
- Declaring a global symbol as static will strictly limit the visibility to the current compilation unit.

`extern`

- A plain forward declaration states that something exists in the same translation unit that has yet to be defined.
- This may cause some problems in some compilers if the actual symbol is in another translation unit (some are smart enough).
- Mark the declaration as `extern` to state that you're declaring a symbol that's not in the translation unit.

extern variables.

Variables located in another translation unit are declared externally in the same fashion.

```
int myctr = 0;

void myincr() {
 ++myctr;
}
```

b.cpp

```
extern int myctr;
extern void myincr();

int main() {
 myincr();
 return myctr;
}
```

a.cpp

Multiple files.

What if a set of functions from one compilation unit needs to be imported a couple of times?

foo.cpp

```
void f() { /* stuff */ }
void g() { /* stuff */ }
int h;
```

Use a header.

foo.h

```
extern void f();
extern void g();
extern int h;
```

Headers

To use the header just do a `#include`

`main.cpp`

```
#include "foo.h"
int main() {
  f(); g(); return h;
}
```

No need to include the header file in compiling.

```
$ g++ -o test foo.cpp main.cpp
$ cl foo.cpp main.cpp
```

Classes

- Classes are a bit tricker to separate.
- The class ***definition*** should be visible whenever an object of that class is referenced (`extern` won't cut it).
- Definition of member functions is independent of the definition of the class.
- The ODR should always be obeyed.

Class Separation

Let's focus on function-less `struct`s first.

```
struct Vector2 {
 double x, y;
};
```

vec2.cpp

```
void neg(Vector2 &v) {
 v.y = -v.y;
 v.x = -v.x;
}
```

main.cpp

```
#include <iostream>
using namespace std;

extern void neg(Vector2 &);
int main() {
  Vector2 z;
  cin >> z.x >> z.y;
  neg(z);
  cout << z.x << z.y << endl;
}
```

Class Separation

Since `Vector2` is referenced in both files (even if it's not instantiated in `vec2.cpp`), the struct needs to be defined on both files.

`vec2.h`

```
struct Vector2 {
 double x, y;
};
```

And `#include` it on both .cpp files...

What happens?

Woops.

Including `vec2.h` on both files causes an ODR violation.

But we need `Vector2` to be defined on both compilation units... what to do?

Use the preprocessor to guard against double-includes.

`vec2.h`

```
#ifndef __VEC2_H__
#define __VEC2_H__
struct Vector2 {
 double x, y;
};
#endif
```

Try compiling again.

Include Guards

```
#ifndef unique_name
#define unique_name
/* all content go here */
#endif
```

The preprocessor pattern that was just done is called an ***include guard***.

This is used to prevent double-inclusion which usually leads to ODR violations*.

It is usually wise to do this whether or not the file has classes or not as this also speeds up compilation.

*I am not entirely accurate here but this is mostly right.

Member Functions

- Member function definition can be separated from class definitions (WL101.1c)
- Member functions that are defined in the class are said to be ***inline***.
- Separate member function definitions should be on its own compilation unit.
- Putting separate member function definitions in headers may result in ODR violations.

Member Functions

Turn `neg()` into a member function...

vec2.h

```
#ifndef __VEC2_H__
#define __VEC2_H__
struct Vector2 {
 double x, y;
 Vector2 neg() const;
};
#endif
```

vec2.cpp

```
#include “vec2.h”
Vector2 Vector2::neg() const {
 Vector2 r = { -x, -y };
 return r;
}
```

Member Functions

Static member functions need not have their static-ness redeclared.

vec2.h

```
#ifndef __VEC2_H__
#define __VEC2_H__
struct Vector2 {
  double x, y;
  Vector2 neg() const;
  static double distance(const Vector2 &a, const
Vector2 &b);
};
#endif
```

vec2.cpp

```
#include <cmath>
#include "vec2.h"
double Vector2::distance(const Vector2 &a, const
Vector2 &b) {
  const double dx = (a.x - b.x), dy = (a.y - b.y);
  return std::sqrt(dx * dx + dy * dy);
}
```

Checkpoint 4.1

Extend the `Vector2` class in the previous slide to add the following member functions using the separated compilation model:

- `Vector2 operator+` (vector add, in-class)
- `Vector2 operator-` (vector subtract, in-class)
- `Vector2 operator*` (inner product, off-class)
- `Vector2 operator%` (cross product, off-class)
- `double mag()` (inline)

Static Initialization Problem

Static Initializers

- Variables defined at global scope can be initialized upon definition.
- While the order of initialization of variables in the same compilation unit is defined...
- The order across different compilation units is undefined. This is the ***Static Initialization Problem.***

SIP Sample

```cpp
int a = 10;
int b = a;
```

b.cpp

```cpp
#include <iostream>
using namespace std;
extern int b;
int c = b;

int main() {
 cout << c << endl;
}
```

a.cpp

Static Initializers

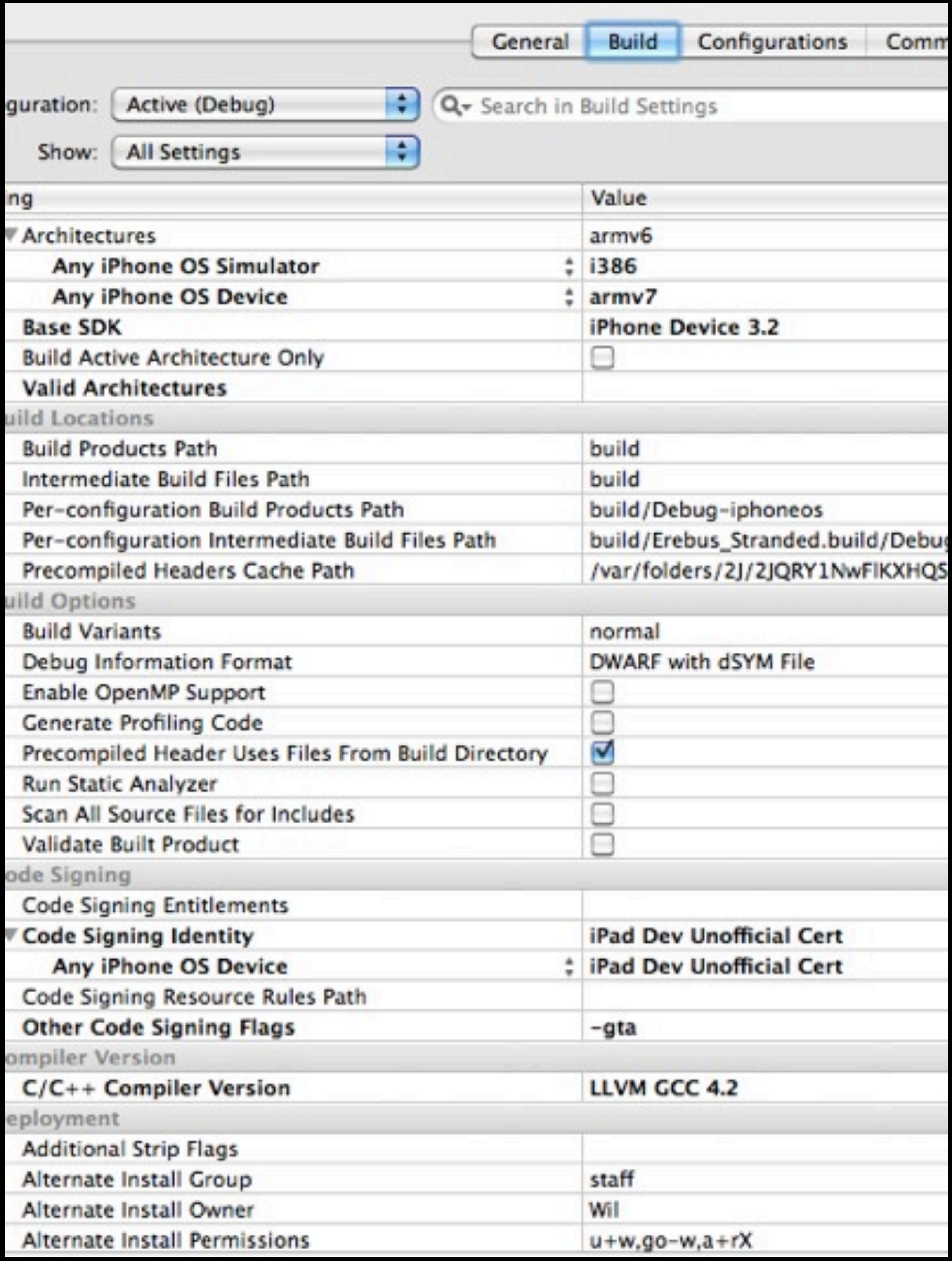
- A possible solution is to use functions that return the variable instead.
- Use the `static` keyword in the function scope to initialize the value the first time.

```
int& getVar() {
 static int var = 10;
 return var;
}
```

Build Settings and Compiler Optimizations

Build Settings

- C++ compilers (together with linkers) have more build settings than the Java compiler.
- Two sets of flags needs to be set: one for the compiler and another for the linker.
- Another two sets of flags are usually noted: Debug and Release

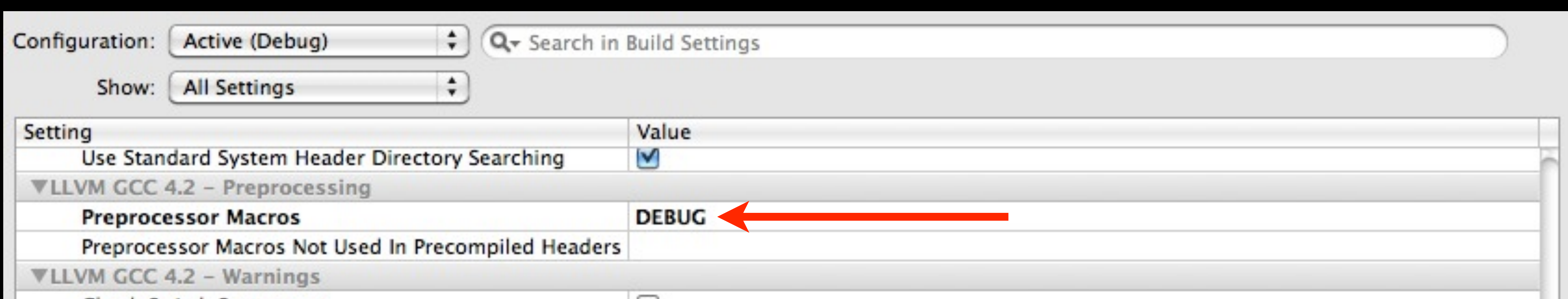


Preproc Symbols

- Predefined Preprocessor Symbols may be defined by the compiler automatically or manually.
- `__cplusplus` is defined whenever a file is being compiled as C++.
- `__LINE__` outputs the current line it's located.
- `__FILE__` outputs a cstring of the file path.

Preproc Symbols

- Check your compiler docs for directions on how to manually set predefined preprocessor symbols.
- For GCC/Clang, it's `-D`*`MacroName`*



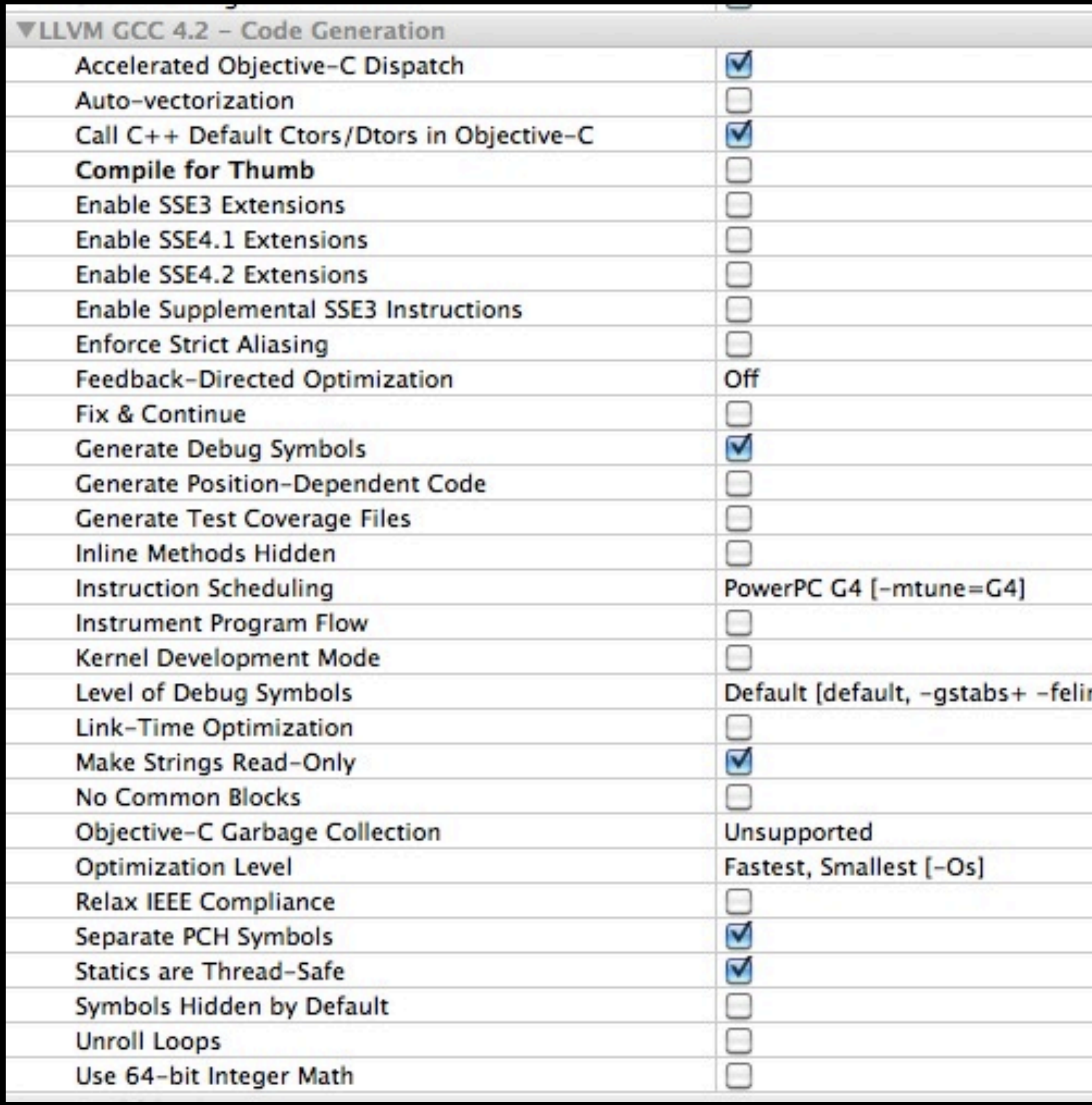
Preproc Symbols

- One use for predefined preprocessor symbols is for conditional compilation.
- You may want to insert possibly slow state checks when debugging but remove them on release for faster execution.

```
  GLenum fbstat = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
  switch(fbstat) {
      case GL_FRAMEBUFFER_COMPLETE_EXT:
          return true;
      case GL_FRAMEBUFFER_UNSUPPORTED_EXT:
      default:
#ifdef DEBUG
          Logger::Log(fbstat, Logger::DOMAIN_GL_FRAMEBUFFER);
#endif
          glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
GL_TEXTURE_2D, 0, 0);
          return false;
  }
```

Optimization

- Optimization flags (usually under "Code Generation") tells the compiler what it can do to make your program faster.
- Usually inversely correlated to debuggability.



Generated Code Example

MetaballController.s:1:1 <No selected symbol>

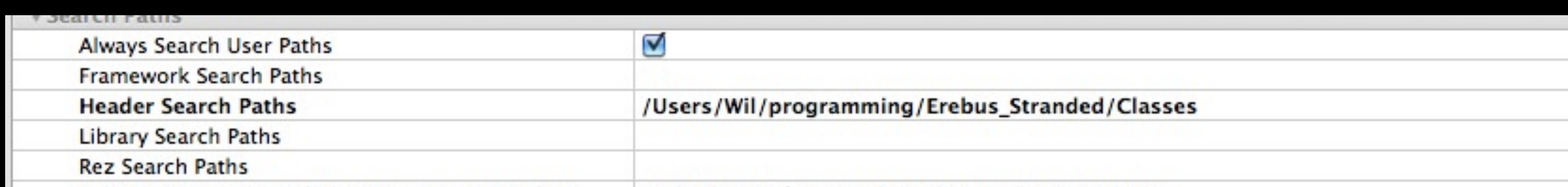
```
	.weak_definition __ZNSt14numeric_limitsIdE7epsilonEv
	.private_extern __ZNSt14numeric_limitsIdE7epsilonEv
__ZNSt14numeric_limitsIdE7epsilonEv:
LFB475:
	.file 4 "/usr/include/c++/4.2.1/limits"
	.loc 4 1054 0
	nop
	nop
	nop
	nop
	nop
	nop
	pushl	%ebp
LCFI12:
	movl	%esp, %ebp
LCFI13:
	subl	$24, %esp
LCFI14:
	call	L11
"L00000000001$pb":
L11:
	popl	%ecx
	.loc 4 1055 0
	leal	LC0-"L00000000001$pb"(%ecx), %eax
	movsd	(%eax), %xmm0
	movsd	%xmm0, -16(%ebp)
	fldl	-16(%ebp)
	leave
	ret
LFE475:
	.align 1
.globl __ZN9GLTexture3getEv
	.weak_definition __ZN9GLTexture3getEv
	.private_extern __ZN9GLTexture3getEv
__ZN9GLTexture3getEv:
LFB2257:
	.file 5 "/Users/Wil/programming/Wil Demo/GLTexture.h"
	.loc 5 30 0
	nop
	nop
	nop
	nop
	nop
	nop
	pushl	%ebp
LCFI15:
	movl	%esp, %ebp
LCFI16:
	subl	$8, %esp
LCFI17:
	.loc 5 30 0
	movl	8(%ebp), %eax
	movl	(%eax), %eax
	movl	(%eax), %eax
	leave
	ret
LFE2257:
	.align 1
.globl __ZN18AbstractController10keyPressedERKN2sf5Event8KeyEventERKNS0_5InputE
	.weak_definition __ZN18AbstractController10keyPressedERKN2sf5Event8KeyEventERKNS0_5InputE
	.private_extern __ZN18AbstractController10keyPressedERKN2sf5Event8KeyEventERKNS0_5InputE
__ZN18AbstractController10keyPressedERKN2sf5Event8KeyEventERKNS0_5InputE:
```

Optimization

- The more optimization the compiler does, the more it "messes up" the code.
- The compiler is free to morph the code (e.g. reorder statements) as long as it does not change the outcome (e.g. replacing "`m + m`" with "`2 * m`").
- The compiler can also strip unused symbols to shrink program size (look out for "debug symbols").
- Optimizations usually makes debugging harder.

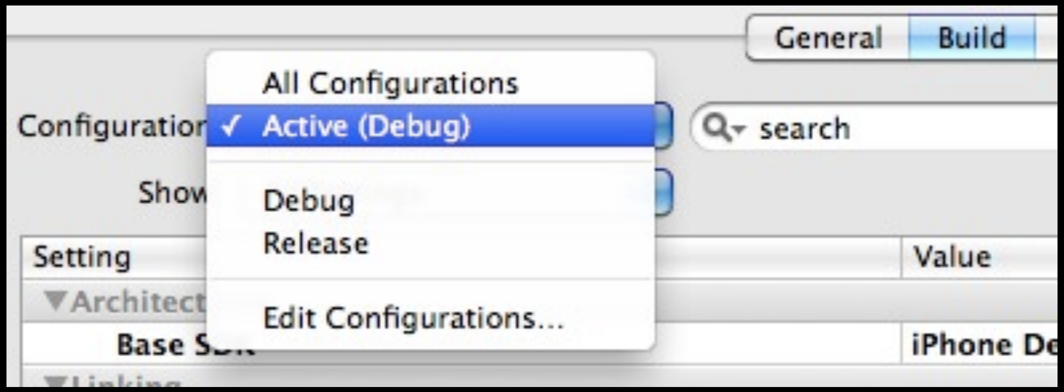
Search Directories

- Remember the difference between `#include <>` and `#include ""` ?
- Where the compiler looks is configurable.
- Search directories for the linker and compiler are separate.

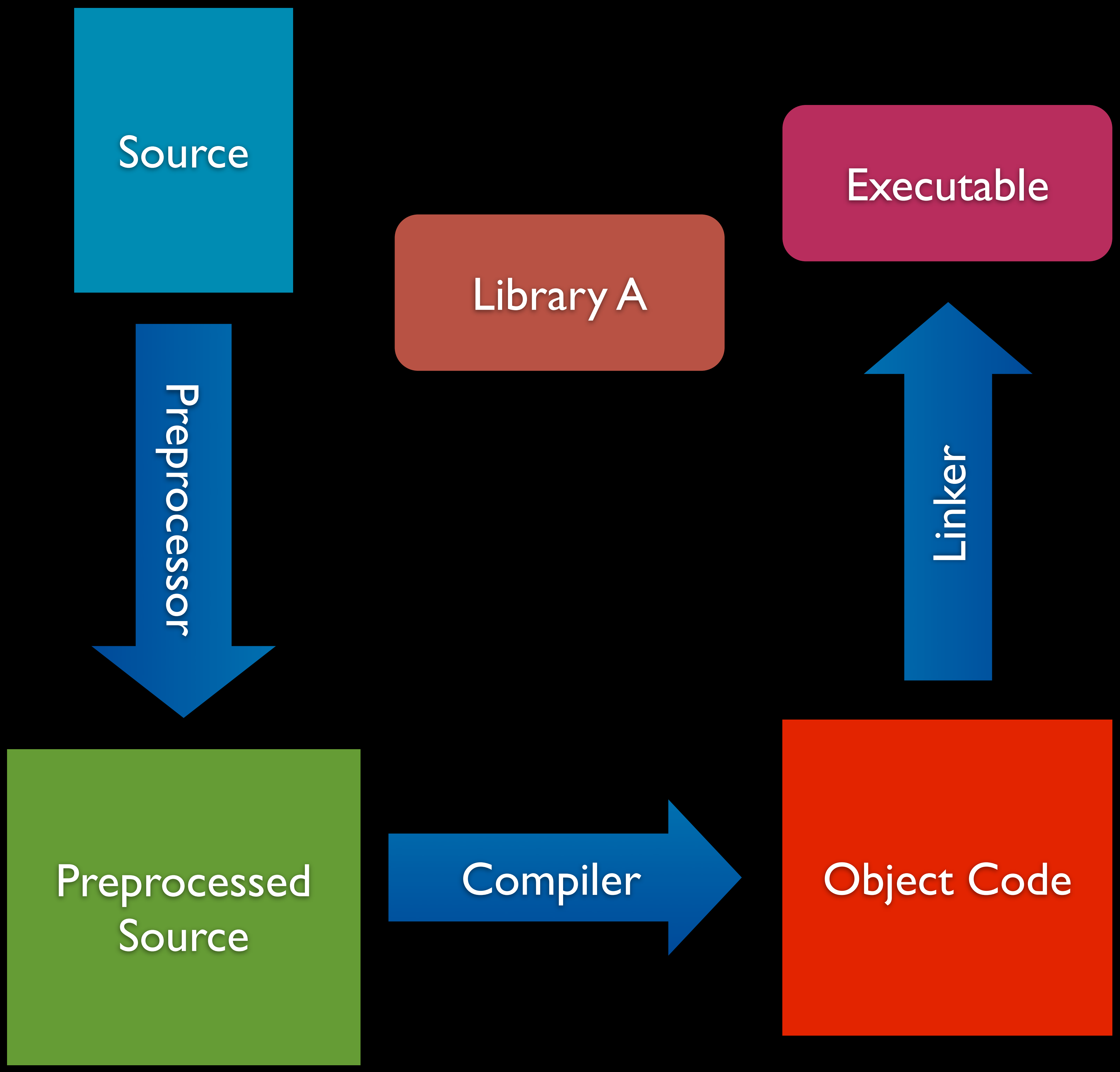


Debug v.s. Release

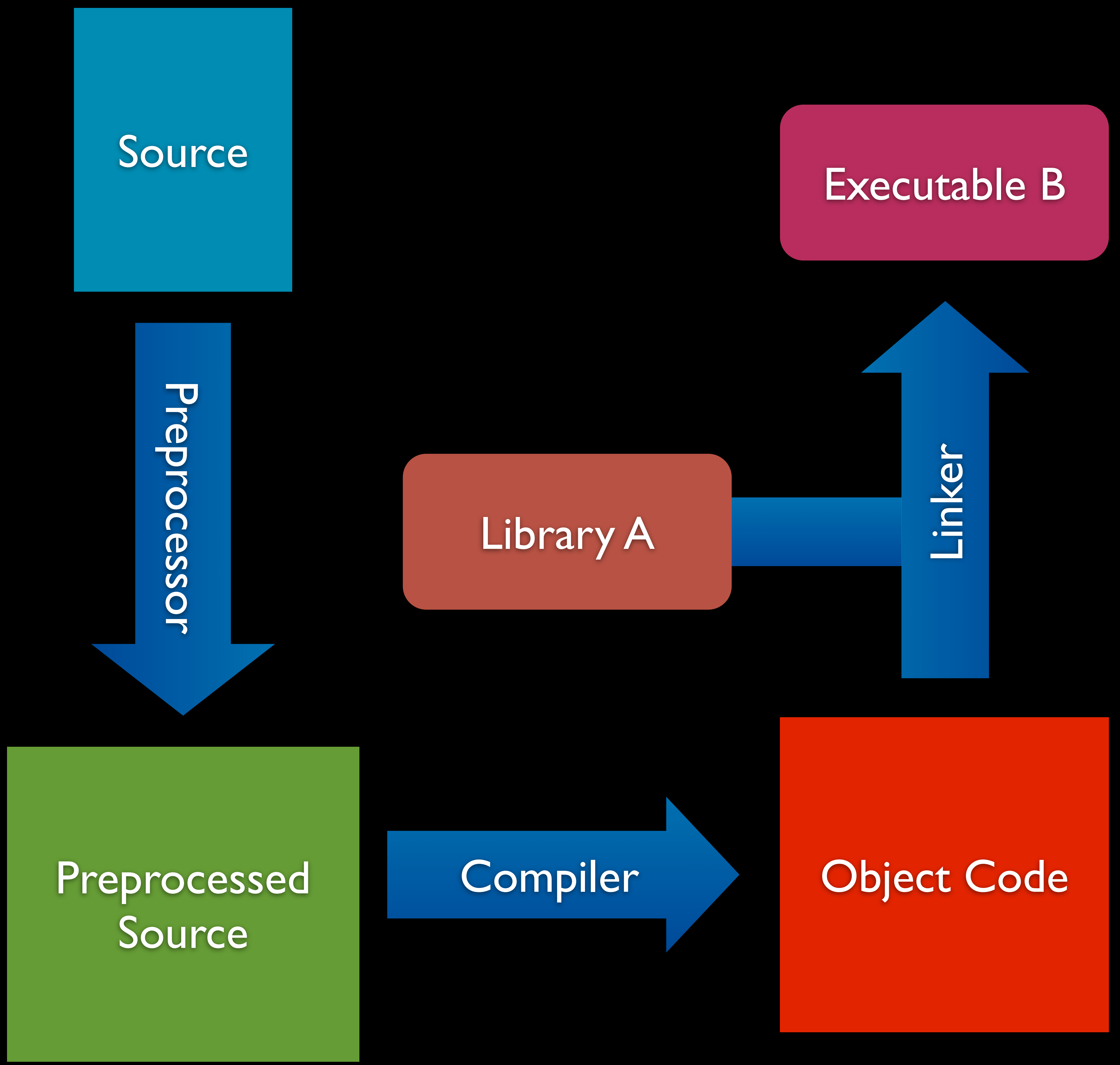
- It is convenient to separate compiler settings between debugging and release.
- Separate search paths, optimization settings, etc.
- All IDEs provide "Build Modes"



Static Libraries

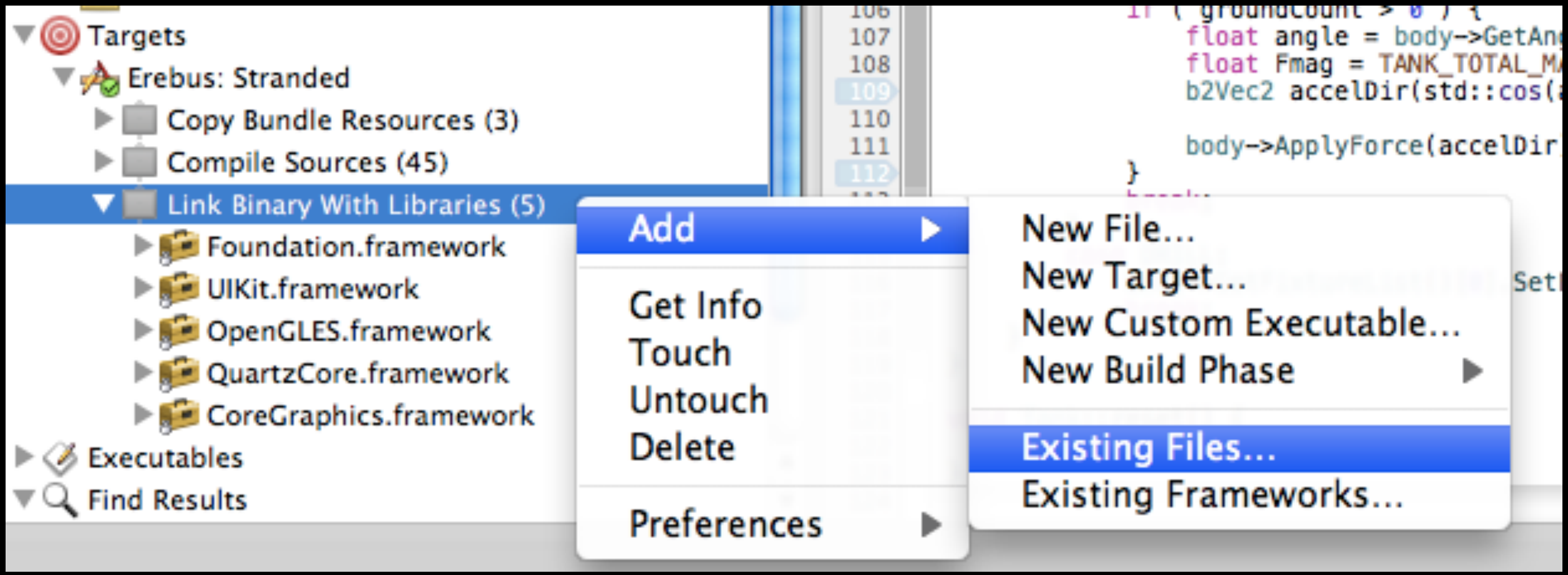


Source
Preprocessor
Preprocessed Source
Compiler
Object Code
Linker
Executable
Library A


Source
Preprocessor
Preprocessed Source
Compiler
Object Code
Linker
Library A
Executable B

Static Library

- Creating static libraries starts from the "create new project" of an IDE.
- To import the library into the project, check the IDE's "linker settings".
- For GCC, it's `-lLibraryName`



Static Library

- Linking to a static library makes its symbols visible to the linker program.
- However, the compiler has no idea on what the library contains (there is no metadata unlike in Java).
- Let the compiler know by declaring the symbols (`extern` and such).
- An easier way is just to reuse the header file of the library.

Static Library

- When distributing static libraries, make sure you bundle the public headers you used to build it.
- When importing static libraries, include the related headers or else you won't see the symbols (like in separated compilation)

Static Library Example

```cpp
void stats(double *v, int len, double &mean, double &var) {
  var = mean = 0;
  for ( int i = 0; i < len; ++i ) {
      mean += v[i];
  }
  mean /= len;
  for ( int i = 0; i < len; ++i ) {
      var += (v[i] - mean) * (v[i] - mean);
  }
  var /= len;
}
```

stat.cpp

```cpp
#ifndef __STAT_HPP__
#define __STAT_HPP__
void stats(double *v, int len, double &mean, double &variance);

#endif
```

stat.hpp

Static Library Example

To make the actual static library (in gcc):

```
$g++ -c stat.cpp
$ar rcs libstat.a stat.o
```

Produces stat.o (object code)

Turns `stat.o` into a static library (libstat.a).

In MSVC++:

```
>cl /c stat.cpp
>lib stat.obj
```

Produces stat.obj (object code)

Turns `stat.obj` into a static library (stat.lib).

Static Library Example

```
#include "stat.hpp"
#include <iostream>
using namespace std;

int main() {
  double v[] = { 1, 2, 3, 4, 5 };
  double mean, variance;
  stats(v, 5, mean, variance);
  cout << "Mean: " << mean << "\n Var: " <<
variance << endl;
}
```

`test.cpp`

```
g++ -o test test.cpp -lstat -L`pwd`
```

"Please link to libstat.a"

"You can find the static libraries here."

Name Mangling

Name Mangling

- The binary (whether it's an executable, static library or dynamic library) contains the names of public symbols.
- How they're represented in the binary is called "Name Mangling".
- C name mangling is different from C++ name mangling (this is where extern linkage comes in).
- Different compilers mangle names differently.
- The bane of creating C++ dynamic libraries.

Name Mangling

`[Wil@Yuushi ~]$ Nm libstat.a` (C++ Version, GCC)

```
libstat.a(stat.o):
0000000000000110 s EH_frame1
0000000000000000 T __Z5statsPdiRdS0_
0000000000000130 S __Z5statsPdiRdS0_.eh
                 U ___gxx_personality_v0
```

`[Wil@Yuushi ~]$ Nm libstatc.a` (C Version, GCC)

```
libstatc.a(statc.o):
0000000000000110 s EH_frame1
0000000000000000 T _stats
0000000000000128 S _stats.eh
```

Note: EH = Exception Handler

Name Mangling

- C++ name mangling has more characters than C (i.e. *decorated*) because it has to encode information for function overloads.
- How will it also account classes and namespaces?
- Different compilers encode function signatures differently (protocols on this are called ABI).
- Not a problem for static libraries because you're usually using the library compiled from the same compiler (dynamic libraries however...)

Linking to a C library.

```c
void stats(double *v, int len, double *mean, double *var) {
  int i;
  *mean = 0;
  for ( i = 0; i < len; ++i ) {
      *mean += v[i];
  }
  *mean /= len;
  *var = 0;
  for ( i = 0; i < len; ++i ) {
      *var += (v[i] - *mean) * (v[i] - *mean);
  }
  *var /= len;
}
```

statc.c

```
$gcc -c statc.c
$ar rcs libstatc.a statc.o
```

Linking to a C Library

```c
#ifndef __STAT_H__
#define __STAT_H__

#ifdef __cplusplus
extern "C" {
#endif

void stats(double *v, int len, double *mean, double *variance);

#ifdef __cplusplus
}
#endif

#endif
```

stat.h

Note the `extern "C"` linkage which tells the compiler to use C-style name mangling when resolving symbols.

There are many ways to do this.

Linking to a C Library

```cpp
#include "stat.h"
#include <iostream>
using namespace std;

int main() {
  double v[] = { 1, 2, 3, 4, 5 };
  double mean, variance;
  stats(v, 5, &mean, &variance);
  cout << "Mean: " << mean << "\n Var: " << variance << endl;
}
```

test.cpp

```
g++ -o test test.cpp -lstatc -L`pwd`
```

Dynamic Libraries

Dynamic Libraries

- Dynamic Libraries are like static libraries except that they are loaded (i.e. linked) *runtime* (i.e. after the entire build procedure).
- Simpler in C but trickier in C++.
- Called "DLLs" in Windows.
- Compiles to its own runnable binary.
- Enables binary sharing between different applications.

Dynamic Libraries

- Depending on the Operating System, when loaded, DLLs may reside in its own memory.
- Be careful on where you allocate memory (do not free a memory that a DLL allocated, have the DLL free it).
- Be prepared to deal with Name Mangling.
- You can optionally tell the compiler to link the library for you (i.e. a dependent library).

C++ Dynamic Libraries

```
#ifndef __CPPDLL_HPP__
#define __CPPDLL_HPP__

class CppDll {
  int i;
public:
  CppDll();
  int incr();
  ~CppDll();
};
#endif
```

cppdll.hpp

```
#include "cppdll.hpp"

CppDll::CppDll() : i(0) {}

int CppDll::incr() {
  return ++i;
}

CppDll::~CppDll() {}
```

cppdll.cpp

```
g++ -o cppdll.dylib -shared cppdll.cpp
cl /LD cppdll.cpp
```

Linking as Dependent Library

```
#include <iostream>
#include "CppDll.hpp"

using namespace std;
int main() {
  CppDll c;
  cout << c.incr() << endl;
}
```

cppdlltest.cpp

```
g++ -o cppdlltest cppdlltest.cpp cppdll.dylib
```

Runtime-loaded Library

For a runtime-loaded library, we needed to somehow call the constructor without accessing the class.

The solution is to use the ***factory pattern***. The function is usually global with C-linkage so that we won't get funky names when we want to look for it

```
extern "C" CppDll* newCppDll();

extern "C" void deleteCppDll(CppDll *);
```

Also, declare the functions virtual just in case (not necessary).

Runtime-loaded Library

```
#ifndef __CPPDLL_HPP__
#define __CPPDLL_HPP__

class CppDll {
  int i;
public:
  CppDll();
  virtual int incr();
  virtual ~CppDll();
};

extern "C" CppDll* newCppDll();

extern "C" void deleteCppDll(CppDll *);
#endif
```

Revised `cppdll.hpp`

Runtime-loaded Library

```cpp
#include "cppdll.hpp"

CppDll::CppDll() : i(0) {}

int CppDll::incr() {
  return ++i;
}

CppDll::~CppDll() {}

extern "C" CppDll* newCppDll() {
  return new CppDll();
}

extern "C" void deleteCppDll(CppDll *ptr) {
  delete ptr;
}
```

Revised `cppdll.cpp`

Runtime-loaded Library

```
[Wil@Yuushi cppdll]$ nm cppdll.dylib
0000000000000dca s  stub helpers
0000000000000cb8 T __ZN6CppDll4incrEv
0000000000000c94 T __ZN6CppDllC1Ev
0000000000000c70 T __ZN6CppDllC2Ev
0000000000000d2c T __ZN6CppDllD0Ev
0000000000000d5a T __ZN6CppDllD1Ev
0000000000000d88 T __ZN6CppDllD2Ev
0000000000001070 S __ZTI6CppDll
0000000000000dc2 S __ZTS6CppDll
0000000000001040 S __ZTV6CppDll
                 U __ZTVN10__cxxabiv117__class_type_infoE
                 U __ZdlPv
                 U __Znwm
                 U ___gxx_personality_v0
0000000000000000 t __mh_dylib_header
0000000000000cda T _deleteCppDll
0000000000000d03 T _newCppDll
                 U dyld_stub_binder
```

Runtime-loaded Library

```cpp
#include <iostream>
#include "CppDll.hpp"
#include <dlfcn.h>

using namespace std;

typedef CppDll* (*CppDll_new)();
typedef void (*CppDll_delete)(CppDll *ptr);

int main() {
   void *dll = dlopen("cppdll.dylib", RTLD_NOW);
   if ( !dll ) {
        cout << "DLL cannot be opened.\n";
        return 0;
   }
   CppDll_new ctor = reinterpret_cast<CppDll_new>(dlsym(dll, "newCppDll"));
   CppDll_delete dtor = reinterpret_cast<CppDll_delete>(dlsym(dll, "deleteCppDll"));

   if ( !ctor ) {
        cout << "Ctor not found\n";
        return 0;
   }
   if ( !dtor ) {
        cout << "Dtor not found\n";
        return 0;
   }

   CppDll *p = ctor();
   cout << "p->incr() = " << p->incr() << '\n';
   dtor(p);
   dlclose(dll);
}
```

cppdlldytest.cpp

Runtime-loaded Library

To compile:

```
g++ -o cppdlldytest cppdlldytest.cpp
```

Note that we didn't include the dynamic library when compiling as we resolve them at runtime. With this:

```
dlopen("cppdll.dylib", RTLD_NOW);
```

And this:

```
CppDll_new ctor = reinterpret_cast<CppDll_new>(dlsym(dll, "newCppDll"));
CppDll_delete dtor = reinterpret_cast<CppDll_delete>(dlsym(dll, "deleteCppDll"));
```

Note that the names we used are "simple" (and no underscore needed). If we used C++ name mangling...

Runtime-loaded Library

The function

```
dlsym(dll_handle, "SymbolName")
```

Returns a void pointer to the corresponding symbol (`0` if the symbol cannot be found).

The type of the symbol has to be known in advance for it to be cast to the proper pointer type.

For function pointers, it would be especially helpful to do `typedef`s to ease casting.

Runtime-loaded Library

UNIX	Windows
dlopen	LoadLibrary
dlsym	GetProcAddress
dlclose	FreeLibrary

Resources:

http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/DynamicLibraries
http://msdn.microsoft.com/en-us/library/ms682599(VS.85).aspx
UNIX Manpages (`man dlopen`, etc.)

Changing Dynamic Libraries

Try changing the `CppDll::incr` in `cppdll.cpp`.

```
int CppDll::incr() {
 return i += 2;
}
```

Recompile `cppdll.cpp` ONLY (not the test programs) and see what happpens on both programs.

Also see what happens when to both programs when you delete `cppdll.dylib`/`dll`.

Additional Notes

- Some compilers have an option of toggling symbol visibility to hide symbols (cannot be looked-up via `dlsym`).
- MSVC/Windows has something called `dllimport` and `dllexport` which complicates things a bit.
- Templates, being “class blueprints” cannot be exported as libraries. They have to distributed in source form.

DONE!

Thank you for your kind cooperation.