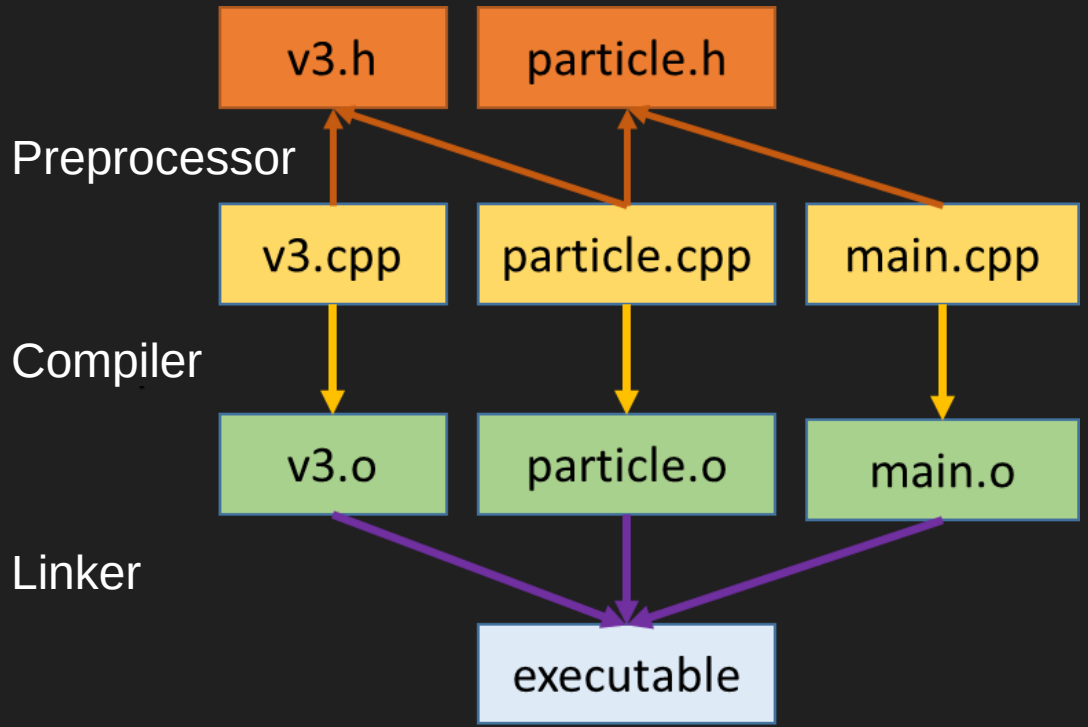
SPECTRAL:
TECHNOLOGIES

WEEK 5

Building programs in C++

Compilation, linking, libraries, cmake

Pipeline


v3.h
particle.h
Preprocessor
v3.cpp
particle.cpp
main.cpp
Compiler
v3.o
particle.o
main.o
Linker
executable

Pipeline

- The compilation itself is only a part of the whole process, but "compilers" usually are the programs, that are capable of executing the entire pipeline (gcc, clang, msvc, icc)
- Usually you don't see these intermediate steps of pipeline and compilers don’t have to implement all of them
  - `ar` may be used for making static libraries from object files
  - `ld` may be used for linking

Preprocessor

- `clang++ -E main.cpp -o main.i`
- Handles `#include` , `#define` , etc

Example

```
// mult.h
#pragma once
double Multiply(double a, double b);
// mult.cpp
#include "mult.h"
double Multiply(double a, double b) { return a * b; }
// dot_product.h
#pragma once
#include <vector>
double DotProduct(const std::vector<double>& a, const std::vector<double>& b);
// dot_product.cpp
#include "mult.h"
#include "dot_product.h"
double DotProduct(const std::vector<double>& a, const std::vector<double>& b) {
    double sum = 0;
    for (size_t i = 0; i < a.size(); ++i) {
        sum += Multiply(a[i], b[i]);
    }
    return sum;
}
```

Example

```
#include "dot_product.h"
int main() {
    std::vector<double> a{1, 2, 3};
    std::vector<double> b{4, 5, 6};
    auto d = DotProduct(a, b);
    return 0;
}
```

SPECTRAL:
TECHNOLOGIES

Object files

```
> clang++-13 -c mult.cpp
> nm mult.o
0000000000000000 T _Z8Multiplydd
> objdump -D mult.o
mult.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <_Z8Multiplydd>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   f2 0f 11 45 f8          movsd  %xmm0,-0x8(%rbp)
   9:   f2 0f 11 4d f0          movsd  %xmm1,-0x10(%rbp)
   e:   f2 0f 10 45 f8          movsd  -0x8(%rbp),%xmm0
  13:   f2 0f 59 45 f0          mulsd  -0x10(%rbp),%xmm0
  18:   5d                      pop    %rbp
  19:   c3                      retq
...
```

SPECTRAL:
TECHNOLOGIES

Object files

```
> nm dot_product.o
0000000000000000     T _Z10DotProductRKSt6vectorIdSaIdEES3_
                     U _Z8Multiplydd
0000000000000000     W _ZNKSt6vectorIdSaIdEE4sizeEv
0000000000000000     W _ZNKSt6vectorIdSaIdEEixEm

> objdump -r dot_product.o
dot_product.o:         file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:

OFFSET               TYPE                 VALUE
000000000000002d     R_X86_64_PLT32       _ZNKSt6vectorIdSaIdEE4sizeEv-0x0000000000000004
000000000000004a     R_X86_64_PLT32       _ZNKSt6vectorIdSaIdEEixEm-0x0000000000000004
0000000000000060     R_X86_64_PLT32       _ZNKSt6vectorIdSaIdEEixEm-0x0000000000000004
000000000000006e     R_X86_64_PLT32       _Z8Multiplydd-0x0000000000000004

> objdump -D dot_product.o
...
  6d:    e8 00 00 00 00            callq 72 <_Z10DotProductRKSt6vectorIdSaIdEES3_+0x72>
...
```

Linking together

```
> clang++ main.o dot_product.o mult.o -o main
> ld *.o # error
> objdump -r main

main:     file format elf64-x86-64
```

C++

- You can have multiple definitions of the same class in your program (all definitions should be identical), but only one definition per a translation unit
  - Thats why there is `#pragma once` in header files
- You can have only one definition of non-inline function in your program
  - Thats why you should mark functions defined in headers as `inline` (functions, that are not actually inlined, usually compiled into weak symbols)
- `extern "C"` can be used to force C-style mangling (basically, no mangling)
- To declare `C* var` or `C& var` you only need a forward declaration `class C;`
- Otherwise you need a definition of class, but don't need methods' definitions
- This doesn't apply to templates

C++-templates

```
// a.h
template<class T>
struct C {
    void f();
}

// a.cpp
definition of f here

// b.cpp
C<int> c;
c.f();
```

- Compiler doesn't know it have to generate `C<int>::f` when processing `a.cpp`
- So you have to put a full class definition in a header. Another solution:

```
// a.cpp
definition of f here
template class C<int>;
```

Symbols' visibility

```
#include "mult.h"

double Abs(double x) { return x < 0 ? -x : x; }
double Multiply(double a, double b) { return Abs(a) * Abs(b); }
```

```
> nm mult.o
0000000000000000 T _Z3Absd
0000000000000050 T _Z8Multiplydd
```

– We want to remove `Abs` from the available symbols

```
static double Abs(double x) { return x < 0 ? -x : x; }
```

```
> nm mult.o
0000000000000000 T _Z8Multiplydd
0000000000000040 t _ZL3Absd
```

Anonymous namespace

– Recommended way

```
namespace {
double Abs(double x) { return x < 0 ? -x : x; }
}
```

```
> nm mult.o
0000000000000000 T _Z8Multiplydd
0000000000000040 t _ZN12_GLOBAL__N_13AbsEd
```

Static library

– Static library is just an archive of object files

```
> ar r libdot_product.a dot_product.o mult.o
> nm -s libdot_product.a

Archive index:
_Z10DotProductRKSt6vectorIdSaIdEES3_ in dot_product.o
_ZNKSt6vectorIdSaIdEE4sizeEv in dot_product.o
_ZNKSt6vectorIdSaIdEEixEm in dot_product.o
_Z8Multiplydd in mult.o

dot_product.o:
0000000000000000  T _Z10DotProductRKSt6vectorIdSaIdEES3_
                  U _Z8Multiplydd  0000000000000000
                  W _ZNKSt6vectorIdSaIdEE4sizeEv
0000000000000000  W _ZNKSt6vectorIdSaIdEEixEm

mult.o:
0000000000000000  T _Z8Multiplydd
```

SPECTRAL:
TECHNOLOGIES

Static library

```
> clang++-13 main.cpp -ldot_product -L. -o main
> objdump -D main
0000000000401e60 <_Z8Multiplydd>:
    401e60:            55                          push   %rbp
    401e61:            48  89   e5                mov    %rsp,%rbp
    401e64:            f2  0f   11  45  f8         movsd  %xmm0,-0x8(%rbp)
    401e69:            f2  0f   11  4d  f0         movsd  %xmm1,-0x10(%rbp)
    401e6e:            f2  0f   10  45  f8         movsd  -0x8(%rbp),%xmm0
    401e73:            f2  0f   59  45  f0         mulsd  -0x10(%rbp),%xmm0
    401e78:            5d                          pop    %rbp
    401e79:            c3                          retq
    401e7a:            66  0f   1f  44  00   00    nopw   0x0(%rax,%rax,1)

...

401ded:           e8 6e 00 00 00                   callq 401e60 <_Z8Multiplydd>
```

Shared library

```
> clang++ -shared dot_product.o mult.o -o libdot_product.so
> clang++ main.cpp -o main -ldot_product -L. -o main
> ldd main
        linux-vdso.so.1 (0x00007fff85be1000)
        libdot_product.so => not found
        libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f75160b6000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f7515f67000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f7515f4c000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7515d5a000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f75162af000)
> ./main
./main: error while loading shared libraries:
libdot_product.so: cannot open shared object file: No such file or directory
> LD_LIBRARY_PATH=. ./main
> sudo cp libdot_product.so /usr/local/lib/
> ldd main
        libdot_product.so => not found
> sudo ldconfig
> ldd main
        libdot_product.so => /usr/local/lib/libdot_product.so (0x00007fd315f29000)
```

Shared libraries

- Much lighter executables
- Can load during runtime with `dlopen` , `dlsym`
- The resulting code might be slightly less effective
- You need all the dynamic dependencies on a target machine to run your executable

Static and shared libraries

- Everything is simple while you have only static libraries
  - `undefined reference` from linker usually means you forgot to add .cpp file to the list of your source files somewhere in cmake or link some external library
  - `multiple definition` usually means you have a complete function in .h, which isn't `inline`
- In real life though you deal with dynamic libraries or a combination of both types mostly:
  - you can merge 2 static libraries together since these are just archives
  - you can link a static library into a dynamic library (but beware of `-fPIC`):

```
clang++ -shared -fPIC -o libtest.so $(OBJS) -lstatic_lib
```

  - you cannot link a dynamic library into a static one
  - you can link a shared library into another shared library

Common issues with dynamic libraries

- If multiple libraries define the same symbol, the first entry is used
- Target machine might have different versions of the libraries your executable is linked with
- You have a dependency on libraries `A` and `B`, which have dependencies on `C-v1` and `C-v2` respectively
- `-fpic` adventures
- And so on

Cmake

- Remember that some cmake targets and commands introduce implicit defaults
  - `RELEASE` target adds `-O3 -DNDEBUG`
  - `DEBUG` adds `-g`
  - `RELWITHDEBINFO` adds `-O2 -g -DNDEBUG`
  - `add_library(... SHARED ...)` adds `-fPIC`
- Prefer target commands to globals. So `target_link_directories`, `target_include_directories`, `target_compile_definitions` and `target_compile_options` instead of `link_directories` and etc.
- `VERBOSE=1` make to make sure you compile with the flags you expect to.
- Make use of `add_subdirectory` and `include`
- Note that `#include "..."` searches the file from the current directory, whereas `<...>` looks for the header in directories specified by `include_directories`