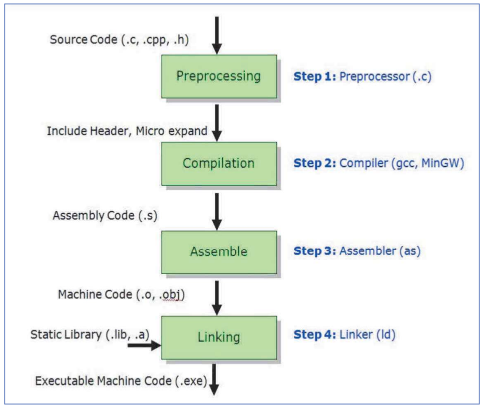
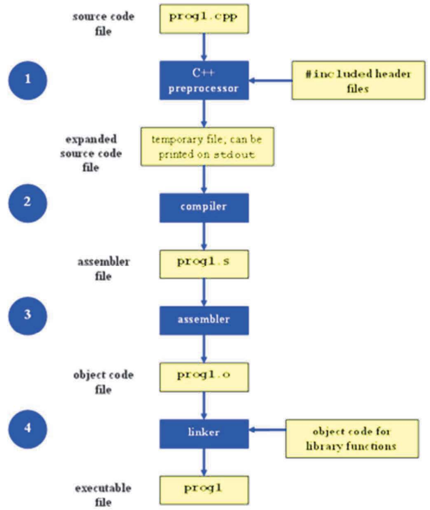
Compilation Process





Compilation Example

- shop.cpp
  - #includes shoppingList.h and item.h
- shoppingList.cpp
  - #includes shoppingList.h
- item.cpp
  - #includes item.h
- How to compile?
  - **g++ -Wall shoppingList.cpp item.cpp shop.cpp –o shop**

What if...

- **We change one of the header or source files?**
  - Rerun command to generate new executable
- **We only made a small change to item.cpp?**
  - not efficient to recompile shoppinglist.cpp and shop.cpp
  - Solution: avoid waste by producing a separate object code file for each source file
    - g++ -Wall –c item.cpp… (for each source file)
    - g++ item.o shoppingList.o shop.o –o shop (combine)
    - Less work for compiler, saves time but more commands

What if...

- We change item.h?
  - Need to recompile every source file that includes it
  - Need to recompile every source file including a header that includes it
    - This means item.cpp and shop.cpp
- Hard to keep track of dependencies ourselves

Solution: Make

- Utility for managing software projects
- Compiles files, ensures they are up to date
- Efficient compilation
  - Only recompiles things that need to be recompiled
- Gets information from file called a “makefile”
- Manual describes makefiles and much more

Example Makefile

```
# Makefile - A Basic Example
all : shop  #usually first
shop : item.o shoppingList.o shop.o
  g++ -g -Wall -o shop item.o shoppingList.o shop.o
item.o : item.cpp item.h
  g++ -g -Wall -c item.cpp
shoppingList.o : shoppingList.cpp shoppingList.h
  g++ -g -Wall -c shoppingList.cpp
shop.o : shop.cpp item.h shoppingList.h
  g++ -g -Wall -c shop.cpp
clean :
  rm -f item.o shoppingList.o shop.o shop
```

- Comments
- Targets
- Prerequisites
- Commands

Dependency Line

Build Process

- **Configure**
  - Script that checks details about the machine before installation
    - Dependencies between packages
  - Creates ‘Makefile’
- **Make**
  - Requires ‘Makefile’ to run
  - Compiles program and creates executables
- **make install**
  - make utility searches for a label named install within the Makefile, and executes only that section of it
  - Copies executables into the final directories (system directories like /usr/bin)

Dynamic Linking

- Allows a process to add, remove, replace or relocate object modules during its execution.
- If shared (dynamic) libraries are used:
  - Only copy a little reference information when the executable file is created
  - Complete the linking during loading time or running time
- Dynamic libraries typically use a “.so” file extension
  - .dll on Windows

Linking and Loading

- Linker collects procedures and links together the object modules into one executable program
- Why isn't everything written as just one big program, saving the necessity of linking?
  - Efficiency: if just one function is changed in a 100K line program, why recompile the whole program? Just recompile the one function and relink.
  - Use a new library without recompiling / redistributing
  - Avoid reading absolutely everything into memory
  - Allow programs to share the memory used by the library

Dynamic linking

- Unix systems: Code is typically compiled as a dynamic shared object (DSO)
- Dynamic vs. static linking resulting size

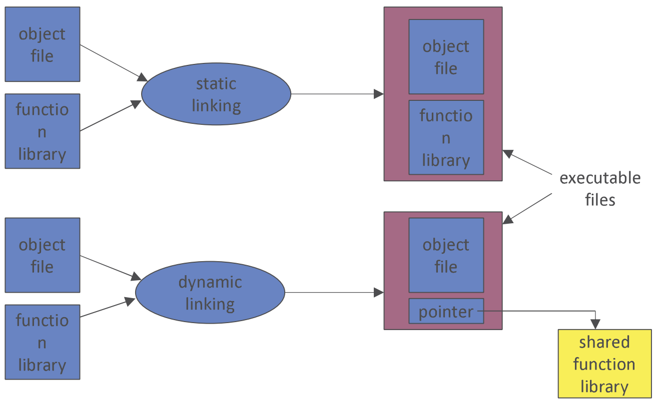
```
$ gcc -static hello.c -o hello-static
$ gcc hello.c -o hello-dynamic
$ ls -l hello
     80 hello.c
  13724 hello-dynamic
1688756 hello-static
```

- Pros and cons?

Advantages of dynamic linking

- The executable is typically smaller
- When the library is changed, the code that references it does not usually need to be recompiled
- The executable accesses the .so at run time; therefore, multiple programs can access the same .so at the same time
  - Memory footprint amortized across all programs using the same .so

Smaller is more efficient



Disadvantages of dynamic linking

- (Slight) performance hit
  - Need to load shared objects (at least once)
  - Need to resolve addresses (once or every time)
  - Remember back to the system call assignment…
- What if the necessary dynamic library is missing?
- What if we have the library, but it is the wrong version?

ldd

- Usage: ldd ./my_program
- Prints out the shared libraries used by a program

```
[tylerd@lnxsrv07 ~]$ ldd ./sfrob
        linux-vdso.so.1 =>  (0x00007ffe749ec000)
        librt.so.1 => /lib64/librt.so.1 (0x00007fe70f395000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007fe70f191000)
        libpthread.so.0 => /lib64/libpthread.so.0
(0x00007fe70ef75000)
        libm.so.6 => /lib64/libm.so.6 (0x00007fe70ec73000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fe70ea5d000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fe70e68f000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe70f59d000)
```

Dynamic Linking

- Allows a process to add, remove, replace or relocate object modules during its execution.
- If shared libraries are called:
  - Only copy a little reference information when the executable file is created
  - Complete the linking during loading time or running time
- Dynamic libraries are typically denoted by the .so (shared object) file extension in Unix system
  - .dll (dynamically linked library) on Windows
- Why we need shared libraries?

Dynamic Loading

- A mechanism to load shared library to memory at runtime
  - E.g. in a C program, use a line of code to load a new library when that line of code is being executed
- You still need to load a shared library
- What're the advantages?
  - Allow start up in the absence of some library
  - Avoid linking unnecessary libraries

Big Picture

- 1. Create a static/shared library
  - Use ar (for static) or gcc (for dynamic)
- 2. Link that library when you compile your program
  - Use gcc (note all the flags)
- 3. If necessary, use dynamic load in your program
  - dlopen(), dlsym(), dlclose()
- 4. Automate the process
  - make

Dynamic Loading

- A sample program
- On success, **dlopen**() returns a non-NULL handle for the loaded library. On error, returns NULL.
- **dlsym**() returns NULL for error
- On success, **dlclose**() returns 0; on error, it returns a nonzero value
- Saves error to **dlerror()**

```c
void *handle;
double (*cosine)(double);
char *error;
handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
if (!handle) {
    // Handle error
    // exit(1);
}
cosine = dlsym(handle, "cos");
if ((error = dlerror()) != NULL)  {
    // Handle error
    // exit(1);
}
printf ("%f\n", (*cosine)(2.0));
dlclose(handle);
```


Makefile Example

```makefile
# Makefile - A Basic Example
all : shop  #usually first
shop : item.o shoppingList.o shop.o
	g++ -g -Wall -o shop item.o shoppingList.o shop.o
item.o : item.cpp item.h
	g++ -g -Wall -c item.cpp
shoppingList.o : shoppingList.cpp shoppingList.h
	g++ -g -Wall -c shoppingList.cpp
shop.o : shop.cpp item.h shoppingList.h
	g++ -g -Wall -c shop.cpp
clean :
	rm -f item.o shoppingList.o shop.o shop
```

Tab here

A Rule

- Comments
- Targets
- Prerequisites
- Commands

Dependency Line

Makefile Structure and Logic

- A *target* is usually the name of a file that is generated by a program
- A *prerequisite* is a file that is used as input to create the target
- A *recipe* is an action that make carries out
- Before make can fully process this rule, it must process the rules for the files that it depends on (the prerequisites)
- The recipe will be carried out if
  - Any file named as prerequisites is more recent than the target file
  - If the target file does not exist at all

Executing Makefile

- When you run `make` command, *the first target* in your Makefile
  - By default, it will look for a file called “Makefile”, use –f to override
- To run a specific target instead of the first, use $ make [target name]
  - Example: make install
- .PHONY target
  - By default, make thinks that the target name is a file
  - Imagine you have a rule `make clean` to delete files, yet you have a file named “clean” in your directory
  - Now `make clean` will tell you nothing to make (why?)
  - Use phony target to enforce `make clean`

Makefile Variable

- [name] = [value]
- To use it later, use $(name)
  - It is conventional to have variables like CC, CFLAGS, etc.
- Automatic variables
  - $@: Target name
  - $<: First prerequisite
  - $?: All prerequisites newer than the target
  - $^: All prerequisites