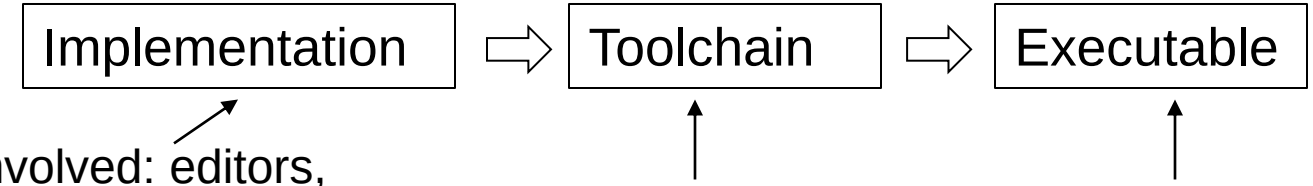
CS601: Software Development for Scientific Computing

Autumn 2024

Week4: Programming Environment, Makefile

Creating a Program (Program Development Environment)

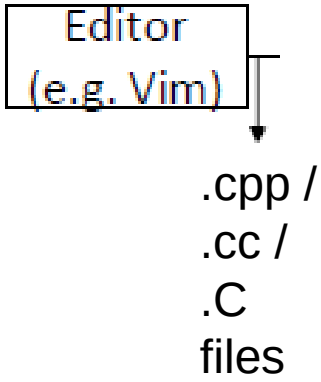


- Tools involved: editors, IDEs, documentation tools
- Tools that are involved: preprocessor, compiler, assembler, loader, linker
- Tools that are involved: Profilers, debuggers, code coverage tools, testing harnesses etc.

- How to create a program and execute?
- What is the entry point of execution?
  - How to pass arguments from command line?
- How is the program laid out in memory?

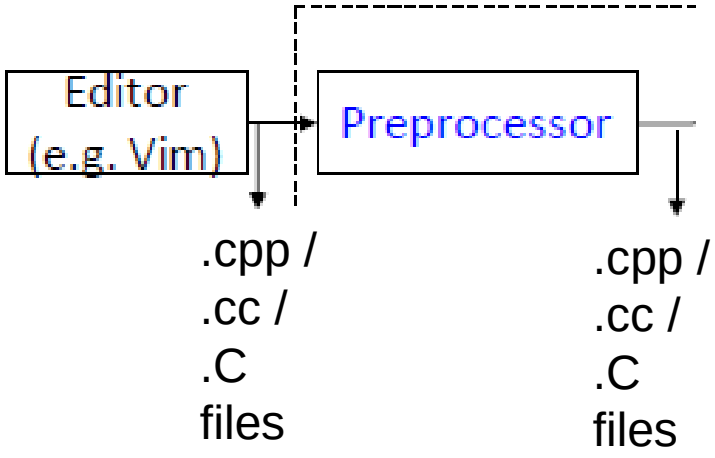
Creating a Program

- Create your c++ program file



Creating a Program

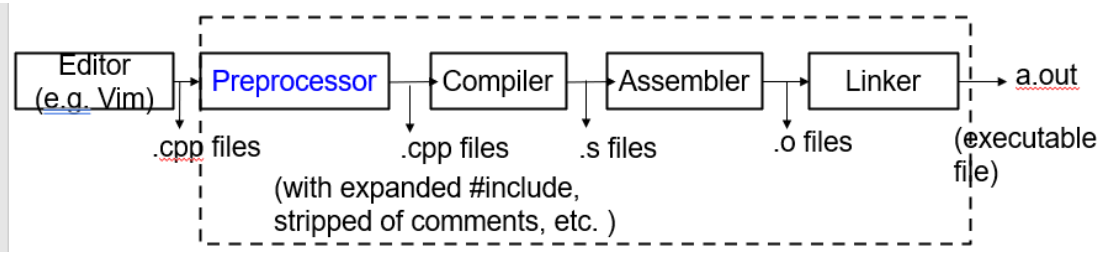
- Preprocess your c++ program file



- removes comments from your program,
- expands #include statements

Detour - Conditional Compilation

- Set of 6 preprocessor directives and an operator.
  - #if
  - #ifdef
  - #ifndef
  - #elif
  - #else
  - #endif
- Operator ‘defined’



#if

```
#if <constant-expression>
cout<<“CS601”;
#endif
```

//This line is compiled only if <constant-expression> evaluates to a value > 0 while preprocessing

```
#define COMP 0
#if COMP
cout<<“CS601”
#endif
```

No compiler error

```
#define COMP 2
#if COMP
cout<<“CS601”
#endif
```

Compiler throws error about missing semicolon

#ifdef

```
#ifdef identifier
cout<<“CS601”;
#endif
```

//This line is compiled only if identifier is defined before the previous line is seen while preprocessing.

identifier does not require a value to be set. Even if set, does not care about 0 or > 0.

```
#define COMP
#ifdef COMP
cout<<“CS601”
#endif
```

```
#define COMP 0
#ifdef COMP
cout<<“CS601”
#endif
```

```
#define COMP 2
#ifdef COMP
cout<<“CS601”
#endif
```

All three snippets throw compiler error about missing semicolon

#else and #elif

```
1. #ifdef identifier1
2. cout<<“Summer”
3. #elif identifier2
4. cout<<“Fall”;
5. #else
6. cout<<“Spring”;
7. #endif
```

//preprocessor checks if identifier1 is defined. if so, line 2 is compiled. If not, checks if identifier2 is defined. If identifier2 is defined, line 4 is compiled. Otherwise, line 6 is compiled.

defined operator

Example:

```
#if defined(COMP)
cout<<“Spring”;
#endif
```

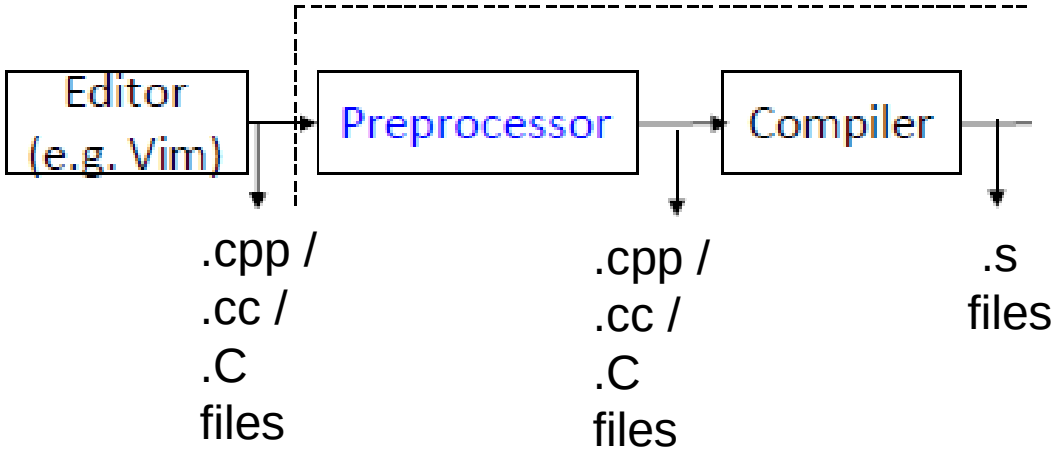
//same as if #ifdef COMP

```
#if defined(COMP1) || defined(COMP2)
cout<<“Spring”;
#endif
```

//if either COMP1 or COMP2 is defined, the printf statement is compiled. As with #ifdef, COMP1 or COMP2 values are irrelevant.

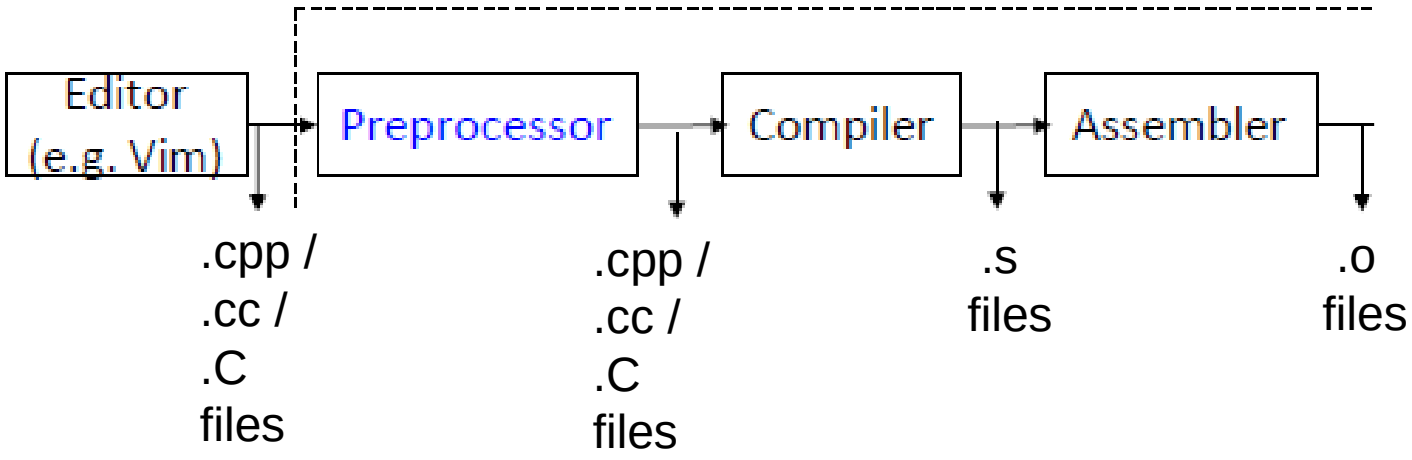
Creating a Program

- Translate your source code to assembly language



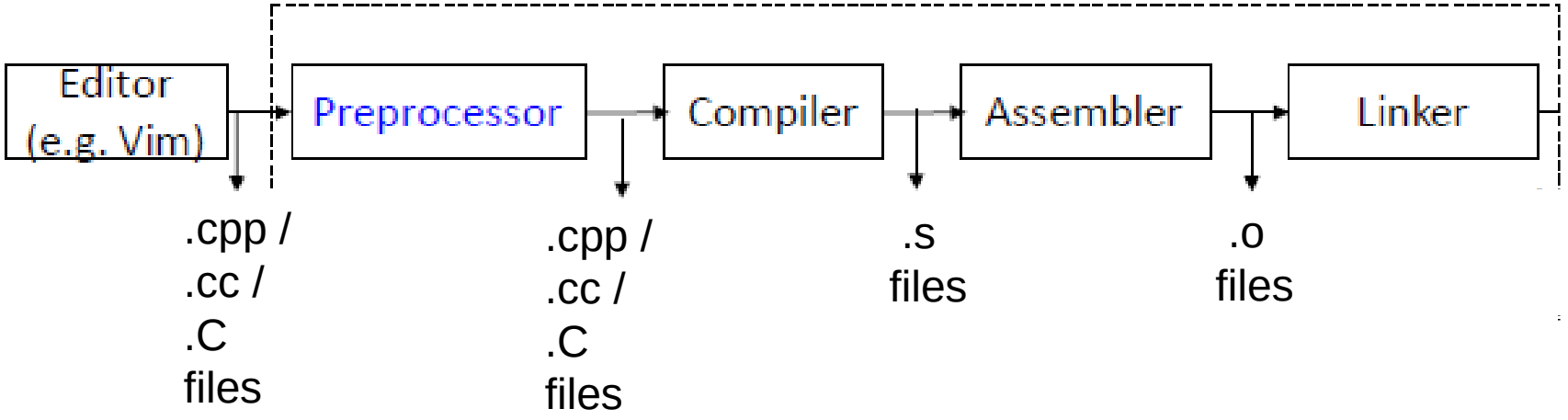
Creating a Program

- Translate your assembly code to machine code



Creating a Program

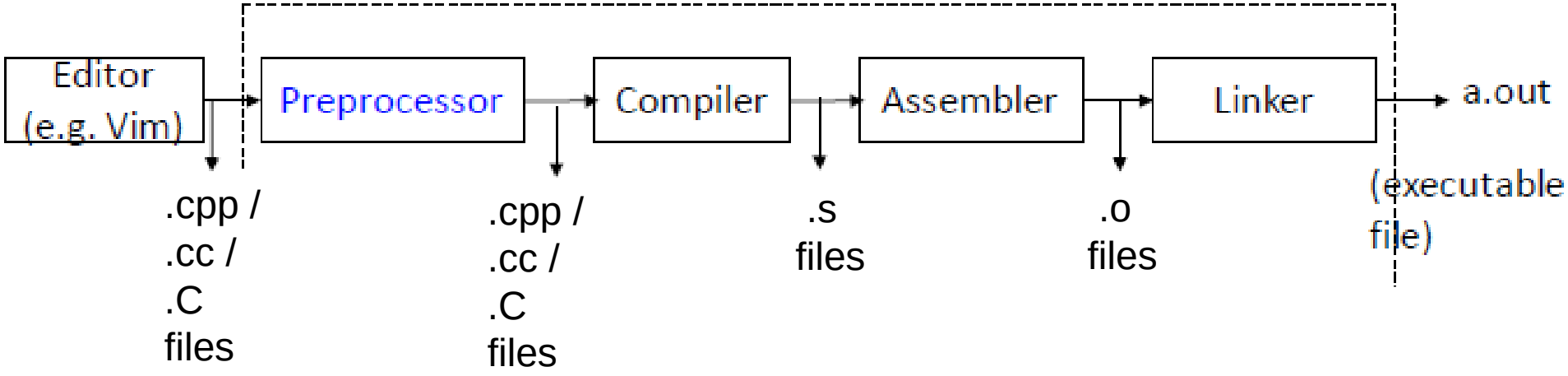
- Get machine code that is part of libraries*



* Depending upon how you get the library code, *linker* or *loader* may be involved.

Creating a Program

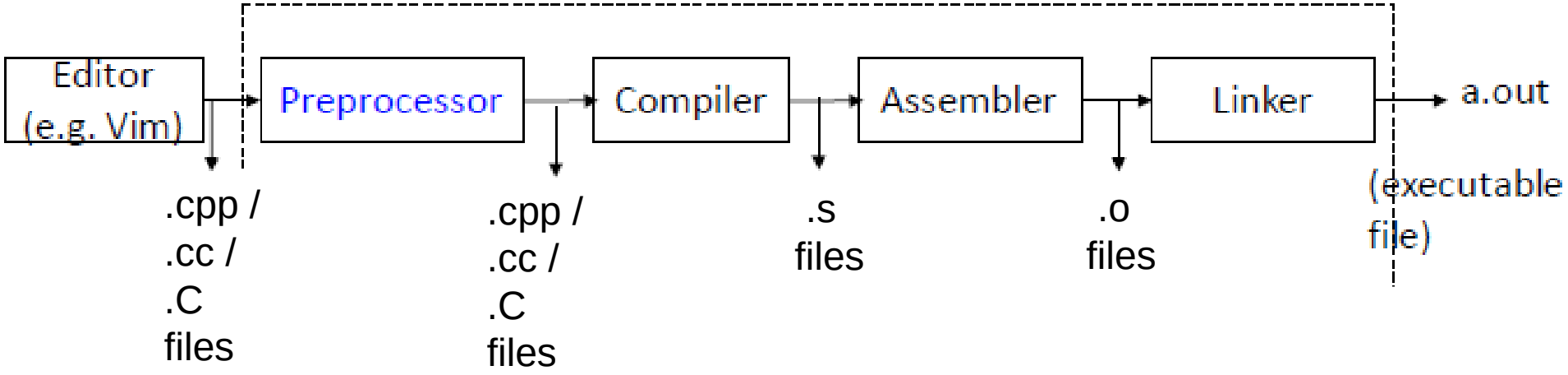
- Create executable



1. Either copy the corresponding machine code OR
2. Insert a ‘stub’ code to execute the machine code directly from within the library module

Creating a Program

- `g++ 4_8_1.cpp -lm`



  - g++ is a command to translate your source code (by invoking a collection of tools)
    - Above command produces `a.out` from `.cpp` file
  - `-l` option tells the linker to 'link' the math library

Creating a Program

- `g++:` other options
  - `-Wall` - Show all warnings
  - `-o myexe` - create the output machine code in a file called `myexe`
  - `-g` - Add debug symbols to enable debugging
  - `-c` - Just compile the file (don't link) i.e. produce a `.o` file
  - `-I/home/mydir` -Include directory called `/home/mydir`
  - -O1, -O2, -O3 – request to optimize code according to various levels

Always check for program correctness when using optimizations

Creating a Program

- The steps just discussed are ‘compiled’ way of creating a program. E.g. C++
- Interpreted way: alternative scheme where source code is ‘interpreted’ / translated to machine code piece by piece e.g. MATLAB
- Pros and Cons.
  - Compiled code runs faster, takes longer to develop
  - Interpreted code runs normally slower, often faster to develop

Creating a Program

- For different parts of the program different strategies may be applicable.
  - Mix of compilation and interpreted – interoperability
- In the context of scientific software, the following are of concern:
  - Computational efficiency
  - Cost of development cycle and maintainability
  - Availability of high-performant tools / utilities
  - Support for user-defined data types

Creating a Program - Executable

- `a.out` is a pattern of 0s and 1s laid out in memory – sequence of machine instructions
- How do we execute the program?
  - `./a.out` <optional command line arguments>

`Makefile` or `makefile`

- Is a file, contains instructions for the `make` program to generate a *target* (executable).
- Generating a target involves:
  1. Preprocessing (e.g. strips comments, conditional compilation etc.)
  2. Compiling (.c -> .s files, .s -> .o files)
  3. Linking (e.g. making printf available)
- A `Makefile` typically contains directives/instructions on how to do steps 1, 2, and 3.

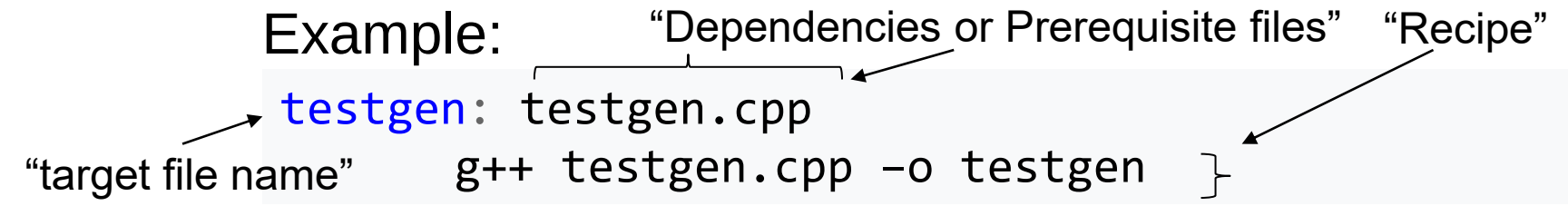
Makefile - Format

1. Contains series of ‘rules’-

```
target: dependencies
[TAB]  system command(s)
```

Note that it is important that there be a TAB character before the system command (not spaces).

Example:



2. And Macro/Variable definitions -

```
CFLAGS = -std=c++11 -g -Wall -Wshadow --pedantic -Wvla –Werror

GCC = g++
```

Makefile - Usage

– The ‘make’ command (Assumes that a file by name ‘makefile’ or ‘Makefile’. exists)

```
n2021/slides/week4_codesamples$ cat makefile
vectorprod: vectorprod.cpp scprod.cpp scprod.h
        g++ vectorprod.cpp scprod.cpp -o vectorprod
```

- Run the ‘make’ command

```
n2021/slides/week4_codesamples$ make
g++ vectorprod.cpp scprod.cpp -o vectorprod
```

`Makefile` - Benefits

- Systematic dependency tracking and building for projects
  - Minimal rebuilding of project
  - Rule adding is ‘declarative’ in nature (i.e. more intuitive to read *caveat: make also lets you write equivalent rules that are very concise and non-intuitive.*)
- To know more, please read: https://www.gnu.org/software/make/manual/html_node/index.html#Top