CS601: Software Development for Scientific Computing

Autumn 2022

Week2: Program Development Environment, Minimal C++, Version Control Systems, Motifs

Recap: Toward Scientific Software

Physical process

$\downarrow$

Mathematical model

$\downarrow$

Algorithm

$\downarrow$

Software program

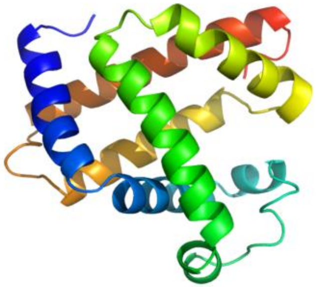
$\downarrow$

Simulation results

Scientific Software - Examples

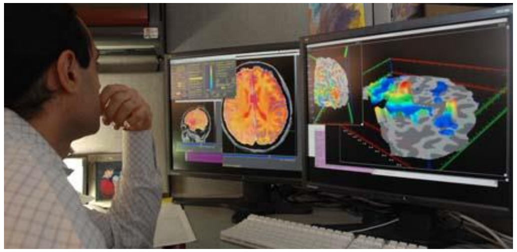
Biology

- Shotgun algorithm expedites sequencing of human genome



Credit: Wikipedia

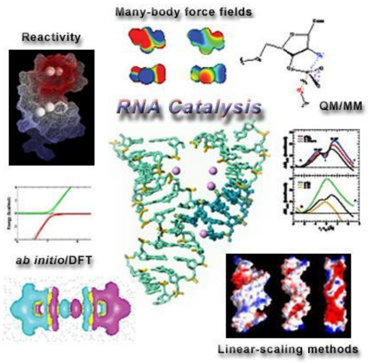
- Analyzing fMRI data with machine learning



Credit: Wikipedia

Chemistry

- optimization and search algorithms to identify best chemicals for improving reaction conditions to improve yields

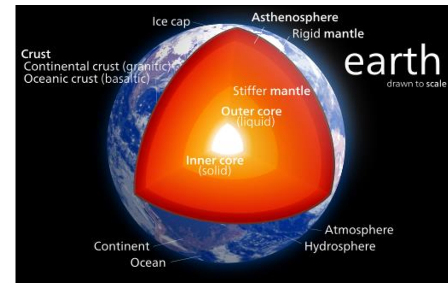


Credit: University of Minnesota

Scientific Software - Examples

Geology

- Modeling the Earth's surface to the core



Credit: Wikipedia

Astronomy

- kd-trees help analyze very large multi-dimensional data sets

Credit: Kaggle.com

Engineering

- Boeing 777 tested via computer simulation (not via wind tunnel)

Scientific Software - Examples

Economics

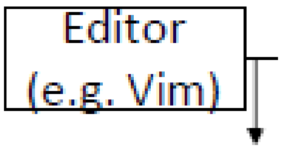
- ad-placement

Entertainment

- Toy Story, Shrek rendered using data-center nodes

Creating a Program

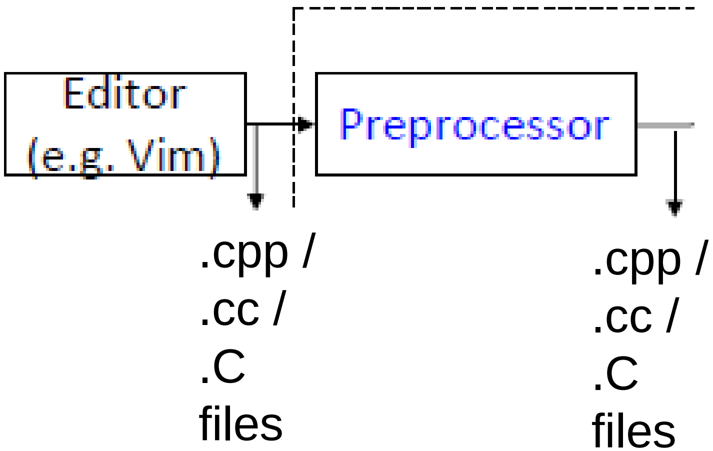
- Create your c++ program file



.cpp /
.cc /
.C
files

Creating a Program

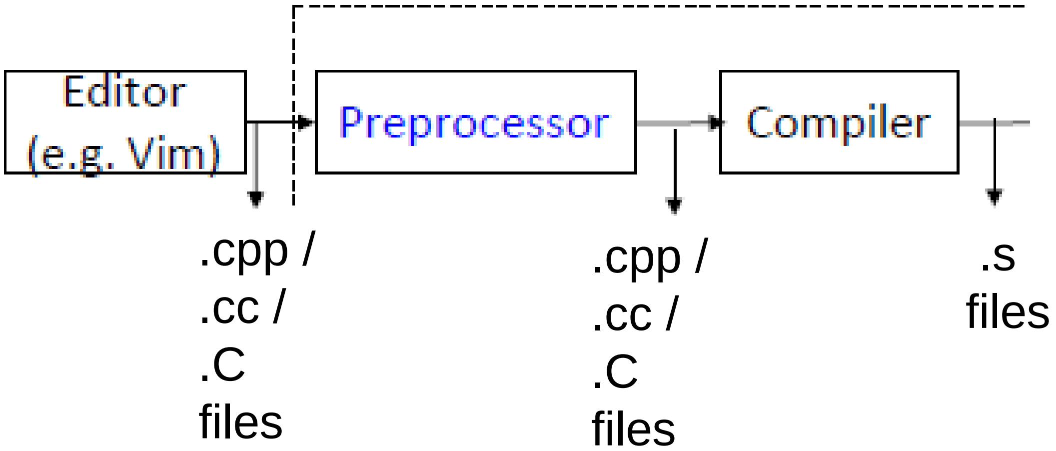
- Preprocess your c++ program file



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

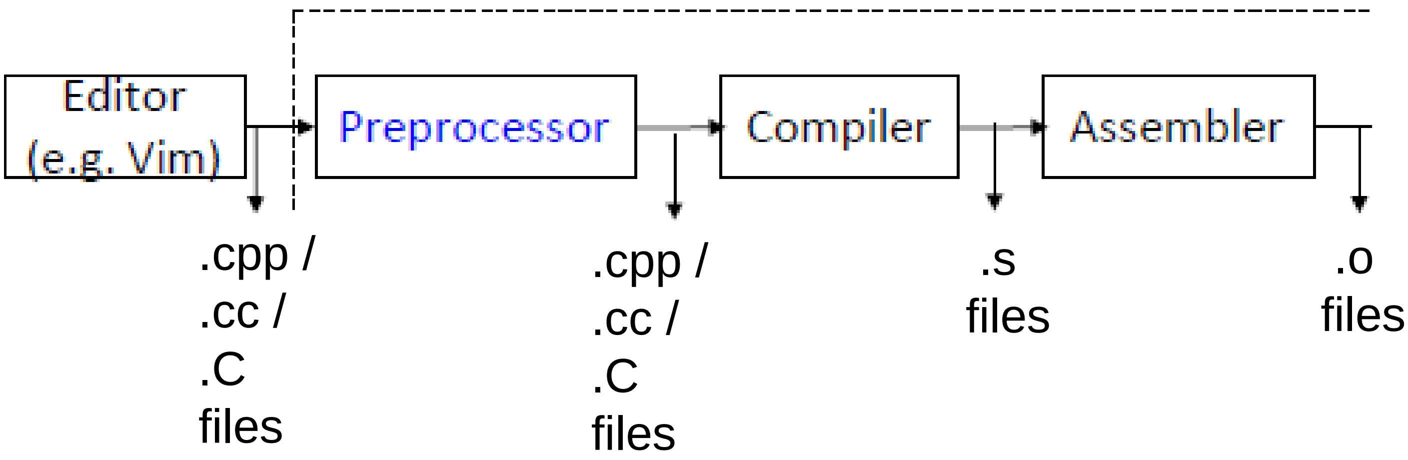
Creating a Program

- Translate your source code to assembly language



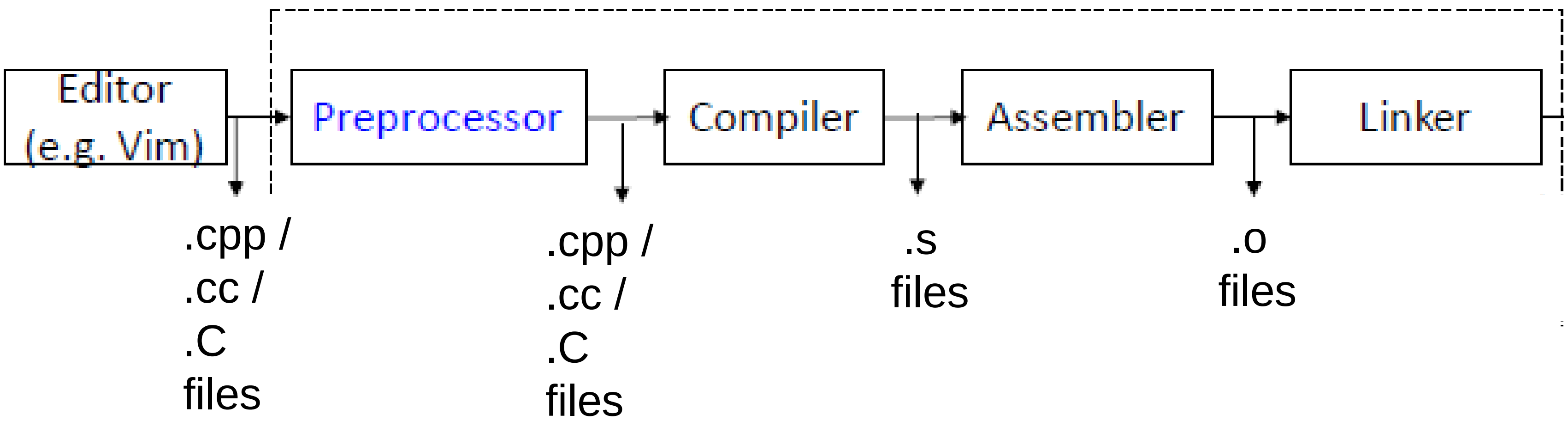
Creating a Program

- Translate your assembly code to machine code



Creating a Program

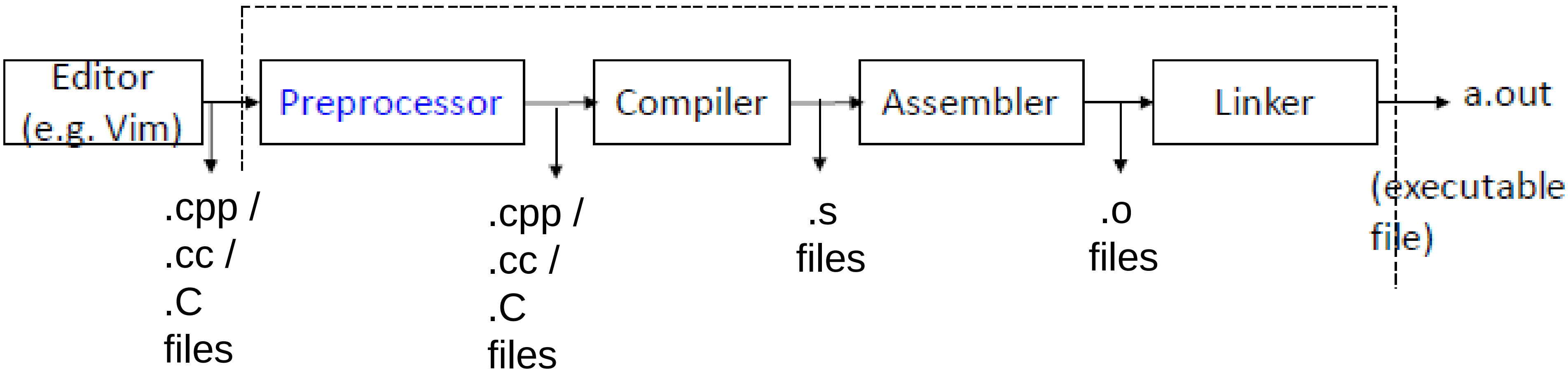
- Get machine code that is part of libraries*



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

Creating a Program

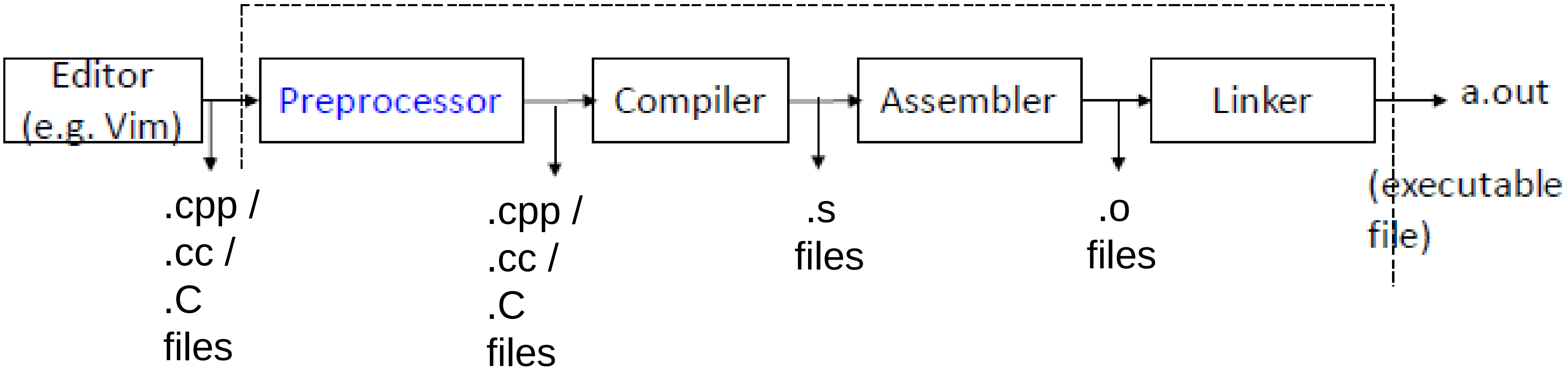
- Create executable



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

- `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>

Command Line Arguments

```
bash-4.1$./a.out
```

//this is how we ran 4_8_1.cpp (refer: week1_codesample)

- Suppose the initial guess was provided to the program as a *command-line argument* (instead of accepting user-input from the keyboard):

```
bash-4.1$./a.out 999
```

Command Line Arguments

- `bash-4.1$./a.out 999`
- Who is the receiver of those arguments and how?

```
int main(int argc, char* argv[]) {
    //some code here.
}
```

Identifier	Comments	Value
`argc`	Number of command-line arguments (including the executable)	2
`argv`	each command-line argument stored as a string	`argv[0]=“./a.out”` `argv[1]=“999”`

The `main` Function

- Has the following common appearance (signatures)

```
int main()
int main(int argc, char* argv[])
```

- Every program must have exactly one `main` function. Program execution begins with this function.
- Return 0 usually means success and failure otherwise
  - EXIT_SUCCESS and EXIT_FAILURE are useful definitions provided in the library `cstdlib`

Functions

- Definition

```
return_type function_name(parameters) {
    //statements
    return <optional_value>
}
```

- Function name and parameters form the *signature* of the function
- In a program, you can have multiple functions with same name but with differing signatures - *function overloading*
- Example:

```
double product(double a, double b) {
    double result = a*b;
    return result;
}
```

Functions – Declaration and Definition

- Declaration: `return_type` **`function_name`**`(parameters);`
- Function definition provided the complete details of the internals of the function. Declaration just indicates the signature.
  - – Declaration exposes the interface to the function

```
double product(double a, double b); //OK
double product(double, double); //OK
```

Functions - usage

- Calling: `function_name(parameters);`
- Example:

```
double product(double a, double b) {
    double result = a*b;
    return result;
}

int main() {
    double retVal, pi=3.14, ran=1.2;
    retVal = product(pi,ran);
    cout<<retVal;
}
```

Functions - usage

- Calling: **function_name**(parameters);
- Example:

```
double product(double a, double b) {
    double result = a*b;
    return result;
}

int main() {
    double retVal, pi=3.14, ran=1.2;
    retVal = product(pi,ran);
    cout<<retVal;
}
```

At least the signature of function must be visible at this line

Functions - usage

- Calling: **function_name**(parameters);
- Example:

```
double product(double a, double b) {
    double result = a*b;
    return result;
}

int main() {
    double retVal, pi=3.14, ran=1.2;
    retVal = product(pi,ran);
    cout<<retVal;
}
```

pi and ran are copied to a and b

Functions - usage

- Calling:
- Example:

```
function_name(parameters);
```

```
double product(double a, double b) {
    double result = a*b;
    return result;
}

int main() {
    double retVal, pi=3.14, ran=1.2;
    retVal = product(pi,ran);
    cout<<retVal;
}
```

pi and ran are copied to a and b

Pass-by-value

Functions - usage

- Calling:
- Example:

```
function_name(parameters);

double product(double& a, double& b) {
    double result = a*b;
    return result;
}

int main() {
    double retVal, pi=3.14, ran=1.2;
    retVal = product(pi,ran);
    cout<<retVal;
}
```

pi and ran are NOT copied to a and b
Pass-by-reference

Reference Variables

- Example:

```
int n=10;
int &re=n;
```

- Like *pointer* variables. `re` is constant pointer to `n` (`re` cannot change its value). Another name for `n`.
  – Can change the value of `n` through `re` though

Exercise: give an example of a variable that is declared but not defined