CS601: Software Development for Scientific Computing

Autumn 2023

Week2: Real Numbers, Programming Environment, ..

Recap: Toward Scientific Software

Physical process

↓

Mathematical model

↓

Algorithm

↓

Software program

↓

Simulation results

Real Numbers ℝ

- Most scientific software deal with Real numbers. Our toy code dealt with Reals
  - Numerical software is scientific software dealing with Real numbers
- Real numbers include rational numbers (integers and fractions), irrational numbers (pi etc.)
- Used to represent values of continuous quantity such as time, mass, velocity, height, density etc.
  - Infinitely many values possible
  - But computers have limited memory. So, have to use approximations.

Representing Real Numbers

- Real numbers are stored as *floating point numbers*
  (floating point system is a scheme to represent real numbers)
- E.g. floating point numbers:
  - $\pi = 3.14159$,
  - $6.03*10^{23}$
  - $1.60217733*10^{-19}$

General format: $\pm x \times b^{e}$

exponent

mantissa
(number ranges from:
1 to b OR 1/b to 1)

base
(e.g. base 10, 8, 2, 16)

3-digit Calculator

- Suppose base, b=10 and
- $x = \pm d_0 . d_1 d_2 \times 10^e$ where $\begin{cases} 1 \leq d_0 \leq 9, \\ 0 \leq d_1 \leq 9, \\ 0 \leq d_2 \leq 9 \\ -9 \leq e \leq 9 \end{cases}$
- precision = length of mantissa
  – What is the precision here?
- Exercise: What is the smallest positive number?
- Exercise: What is the largest positive number?
- Exercise: How many numbers can be represented in this format?
- Exercise: When is this representation not enough?

Floating Point System - Fundamentals

- **Precision (p)** - Length of mantissa
  - E.g. p=3 in 1.00×10^{-1}
- **Unit roundoff (u)** – smallest positive number where the *computed* value of 1+u is different from 1
  - E.g. suppose p=4 and we wish to compute 1.0000+ 0.0001=1.0001
  - But we can't store the exact result (since p=4). We end up storing 1.000.
  - So, computed result of 1+u is same as 1
  - Suppose we tried adding 0.0005 instead. 1.0000+0.0005=1.0005 Now, round this: 1.001

  $\Rightarrow$ **u =0.0005**
- **Machine epsilon (ϵ_{mach})** – smallest a-1, where a is the smallest representable number greater than 1
  - E.g. consider 1.001 – 1.000 = 0.001.

  $\Rightarrow$ **usually $\epsilon_{mach} = 2 * u$**

Floating Point System - Fundamentals

- **Forward error and backward error**

$\text{Comp}(f(x)) = (1+\epsilon_1)f((1+\epsilon_2)x),$

where $\epsilon_i \text{ <= } u$ (u is unit roundoff)

Comp(f(x)) is the computed value i.e. machine representable value of f(x).

Suppose ϵ_2 is zero. Then $\dfrac{\text{Comp}(f(x)) - f(x)}{f(x)} = \epsilon_1$

Floating Point System - Fundamentals

- **Forward error example**

Let $y = \sqrt{2}$, $z = y^2$ and

$y = \sqrt{2}$ `implemented as: y = sqrt(2);`

$z = y^2$ `implemented as: z = y * y;`

with double precision floating point system

Then forward error, $\left\{\frac{Comp(f(x)) - f(x)}{f(x)}\right\}$, can be calculated

(note: f(x) = z = 2, and Comp(f(x)) = y*y)

```
y:1.41421356237
z:2
res1=z-2:4.44089209985e-16
res2=res1/z:2.22044604925e-16
```

Absolute error / relative error

Forward error
(also happens to be `u, unit rouondoff,` for `double`)

Floating Point System - Fundamentals

- **Backward error example**

  Let $z = \sin(2\pi)$. Then forward error is infinity!

  Subtract x with a multiple of 2π to make $0 \leq x < 2\pi$

  And then compute sin(x) to get the absolute error for $x \geq 2\pi$ at most u$|x|$ (u is unit roundoff)

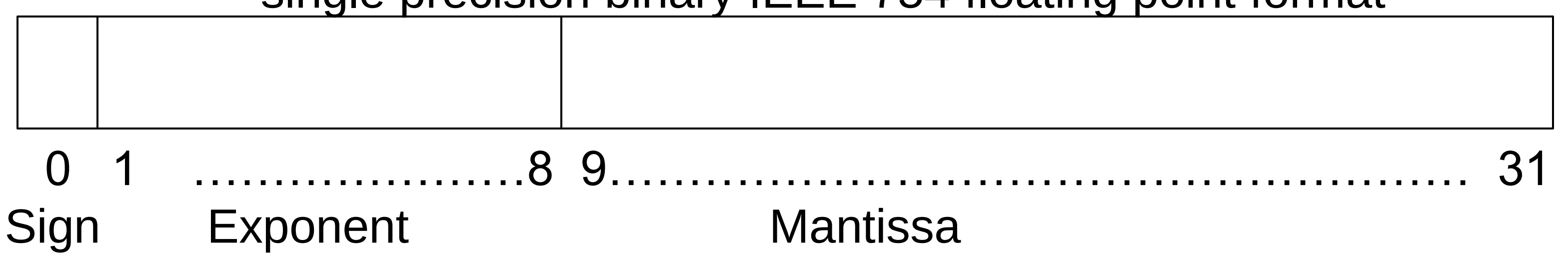
  This is *perturbing* the argument x (*argument reduction*). Instead of computing sin(x) we are computing sin($(1+\epsilon_2)x$). This is example of backward error.

IEEE 754 Floating Point System

- Prescribes single, double, and extended precision formats

Precision	**u**	**Total bits used (sign, exponent, mantissa)**
Single	6×10^{-8}	32 (1, 8, 23)
Double	2×10^{-16}	64 (1, 11, 52)
Extended	5×10^{-20}	80 (1, 15, 64)

single precision binary IEEE 754 floating point format



IEEE 754 Floating Point System

double precision binary IEEE 754 floating point format

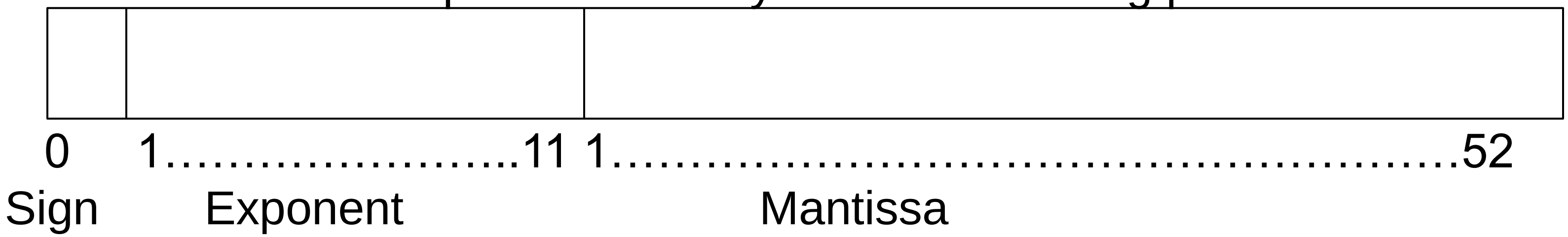
- if exponent bits e_1-e_{11} are not all 1s or 0s, then the *normalized* number

$$n = \pm(1.m_1 m_2 .. m_{52})_2 \times 2^{(e_1 e_2 .. e_{11})_2 - 1023}$$

- **Machine epsilon** is the gap between 1 and the next largest floating point number. $2^{-52} \approx 10^{-16}$ for double.
- Exercise: What is minimum positive *normalized* double number?
- Exercise: What is maximum positive *normalized* double number?

IEEE 754 Floating Point System

double precision binary IEEE 754 floating point format



- if exponent bits e_1-e_{11} are all 0s, then:

  the *subnormal* number

$$n = \pm(\mathbf{0}.m_1 m_2..m_{52})_2 \times 2^{(e_1 e_2..e_{11})_2 - 102\mathbf{2}}$$

- if exponent bits e_1-e_{11} are all 1s, then:

  we can get –inf, NaN, or +inf based on value of $m_1 m_2..m_{52}$
  - If any m is non-zero, the number is NaN (not a number)

IEEE 754 Floating Point – Misc..

- **+0, -0, Inf, and NaN –**
  - Stop your program when you see a NaN (indicative of a bug)
  - How to check if a number is NaN?

    `if (x == x)` is `false`

    Exercise: Give an example when you get a NaN?

- **Rounding modes –** Round up, Round down, Round to nearest, Round towards zero
  - Default is round to nearest. Can be set using compiler options and library methods. Avoid changing rounding modes.
  - Can use this to flush out bugs! (change round modes and results shouldn't change drastically).

IEEE 754 Floating Point Arithmetic

- Be wary of comparison
  - The special case of `x=y; if(y == x)`
- Order is important
  - Floating point arithmetic is *not associative*
    - (x+y)+z not the same as x+(y+z)
- Explicit coding of textbook formula may not be the best option to solve
  - $x^2 - 2px - q = 0$ p and q are positive: p=12345678, q=1
  - **Exercise:** find the minimum of the roots.

- Subtracting approximations of two nearby numbers results in a bad approximation of the actual difference – **catastrophic cancellation**

Creating a Program (Program Development Environment)

Implementation ⇨ Toolchain ⇨ Executable

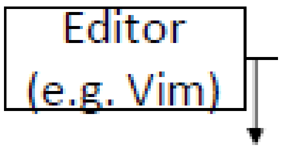
- Tools involved: editors, IDEs, documentation tools
- Tools that are involved: preprocessor, compiler, assembler, loader, linker

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

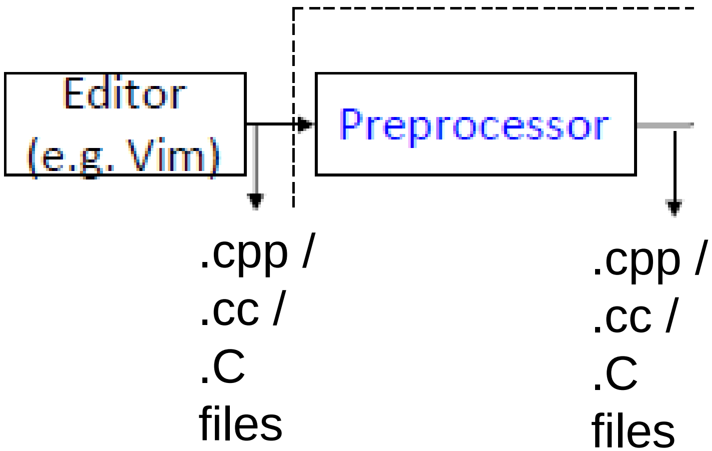
- Create your c++ program file



.cpp /
.cc /
.C
files

Creating a Program

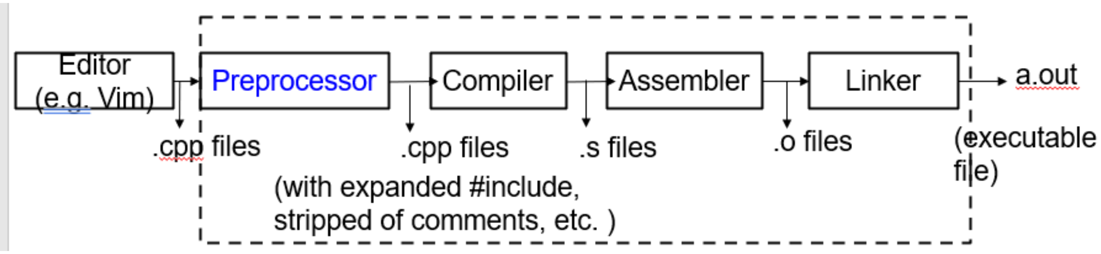
- Preprocess your c++ program file



- 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
```

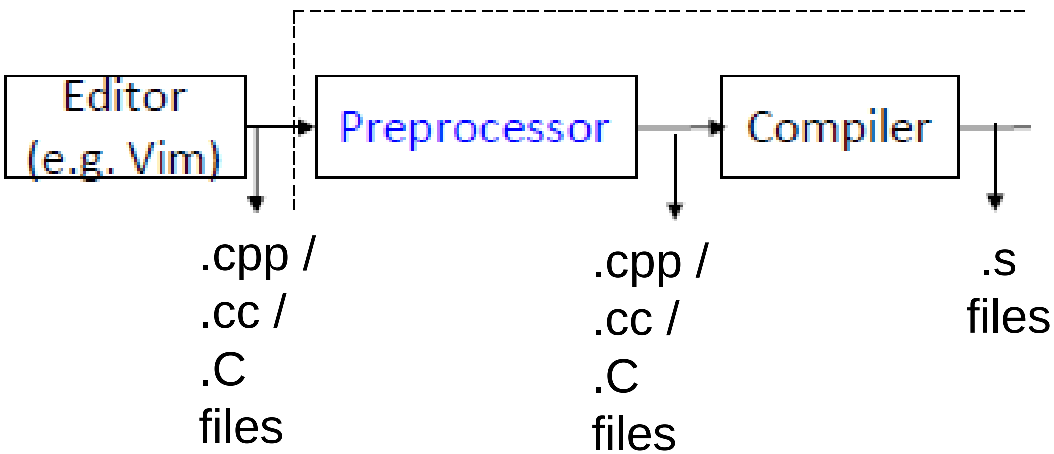
//same as if #ifdef COMP

```
#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.

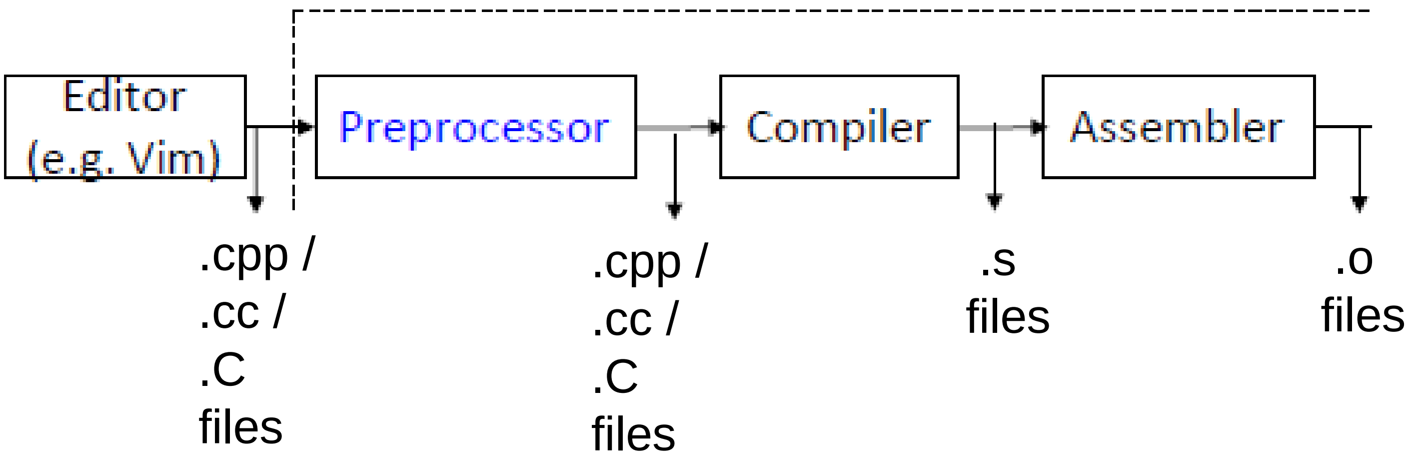
Creating a Program

- Translate your source code to assembly language



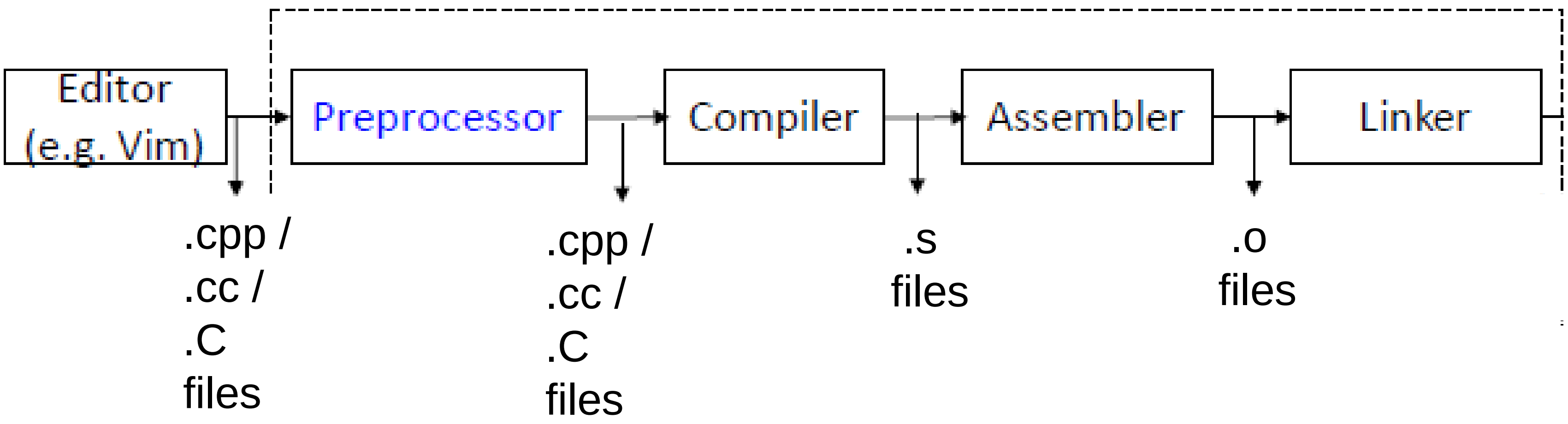
Creating a Program

- Translate your assembly code to machine code



Creating a Program

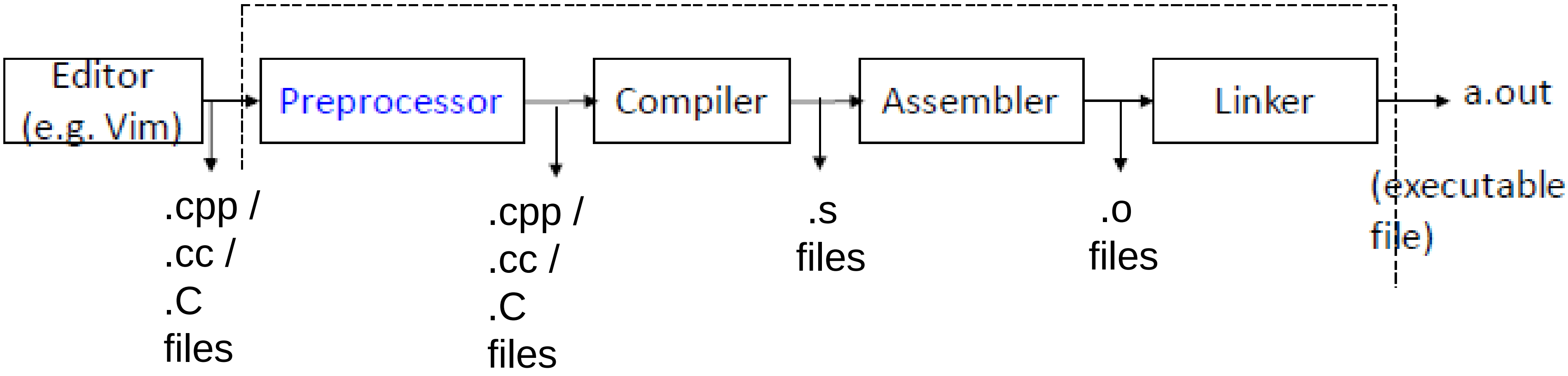
- Get machine code that is part of libraries*



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

Creating a Program

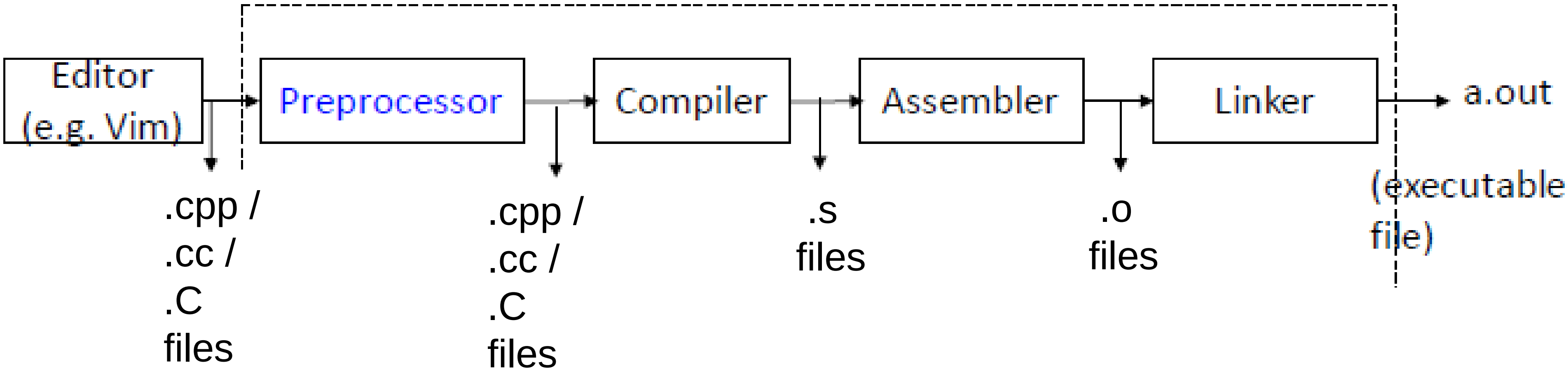
- Create executable



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>

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:
- 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;
}
```

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:
- 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

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

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