CS601: Software Development for Scientific Computing

Autumn 2022

Week2: Scientific software- examples, Program Development Environment, Minimal C++, Version Control Systems, Motifs

Recap: Toward Scientific Software

Physical process

↓

Mathematical model

↓

Algorithm

↓

Software program

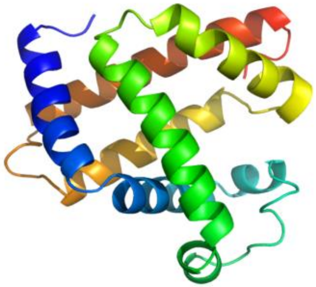
↓

Simulation results

Scientific Software - Examples

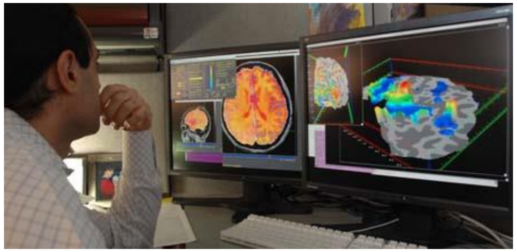
Biology

- Shotgun algorithm expedites sequencing of human genome



Credit: Wikipedia

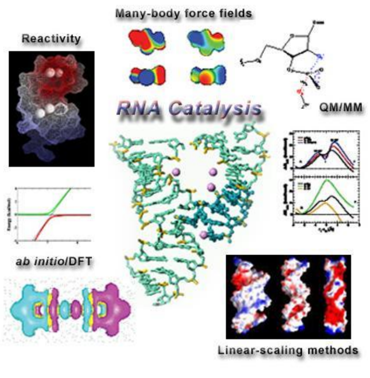
- Analyzing fMRI data with machine learning



Credit: Wikipedia

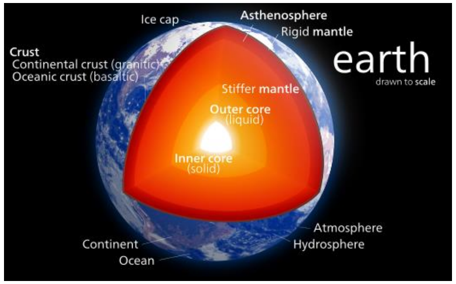
Chemistry

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



Credit: University of Minnesota

Scientific Software - Examples



Credit: Wikipedia

Geology

- Modeling the Earth's surface to the core

Credit: Kaggle.com

Astronomy

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

Engineering

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

Scientific Software - Examples

Economics

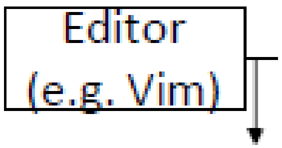
- ad-placement

Entertainment

- Toy Story, Shrek rendered using data-center nodes

Creating a Program

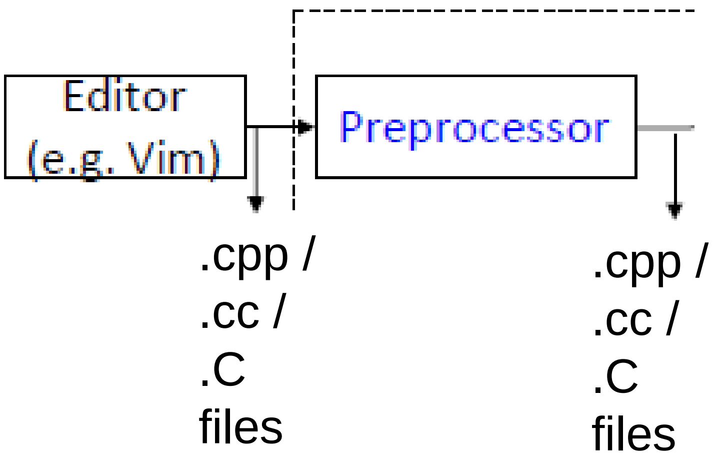
- Create your c++ program file



.cpp /
.cc /
.C
files

Creating a Program

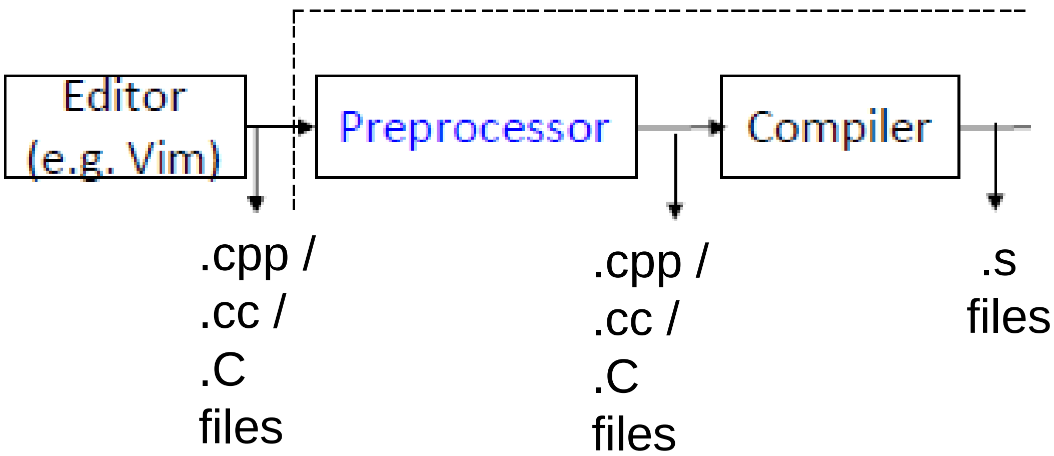
- Preprocess your c++ program file



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

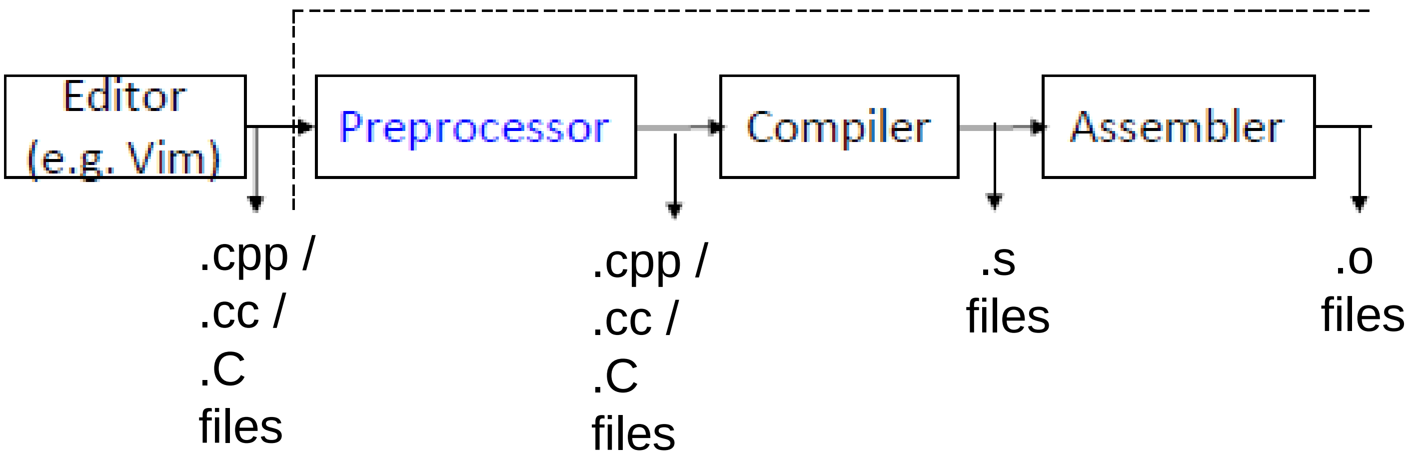
Creating a Program

- Translate your source code to assembly language



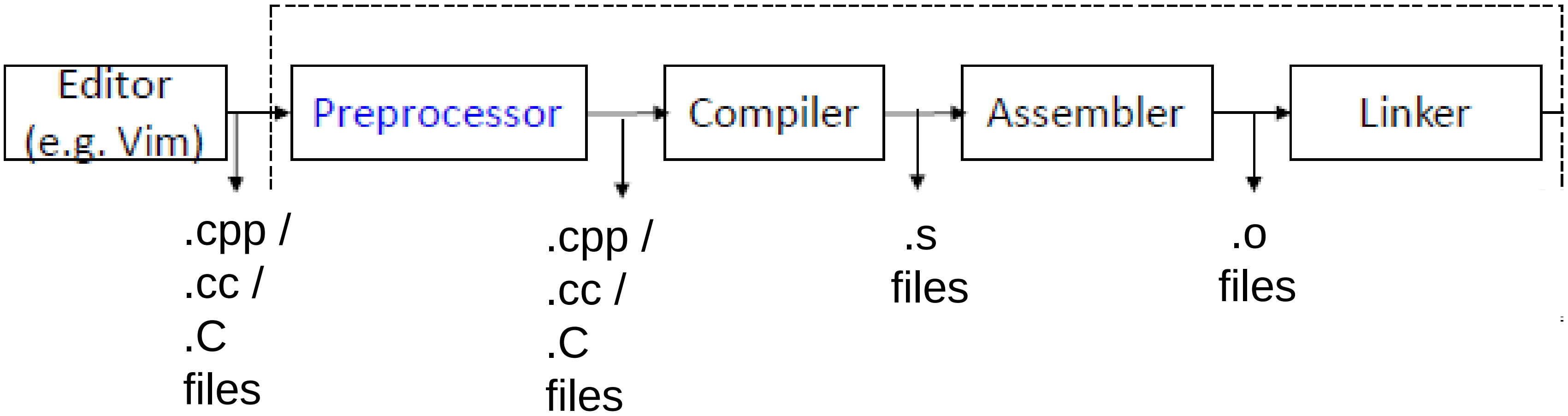
Creating a Program

- Translate your assembly code to machine code



Creating a Program

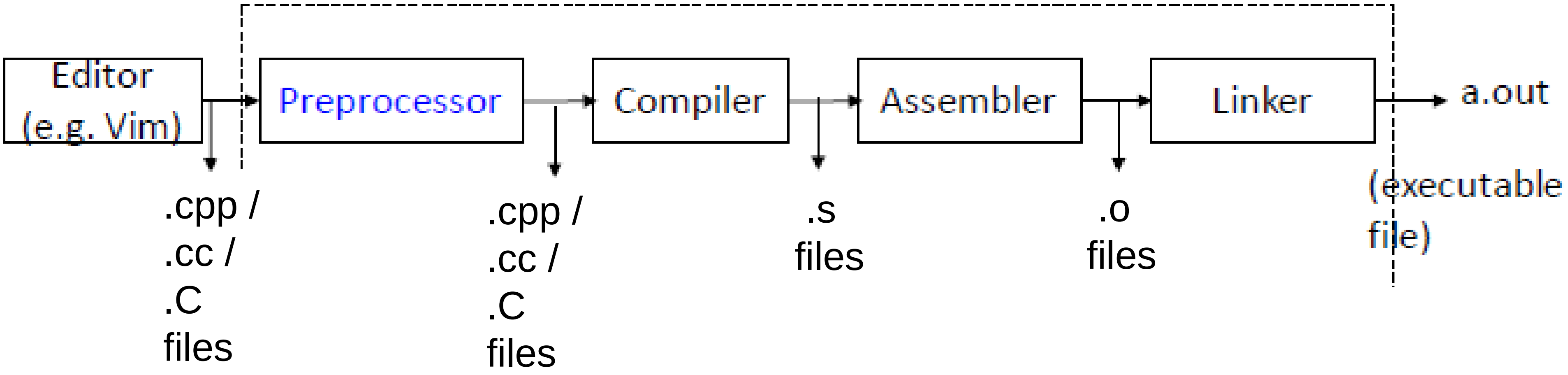
- Get machine code that is part of libraries*



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

Creating a Program

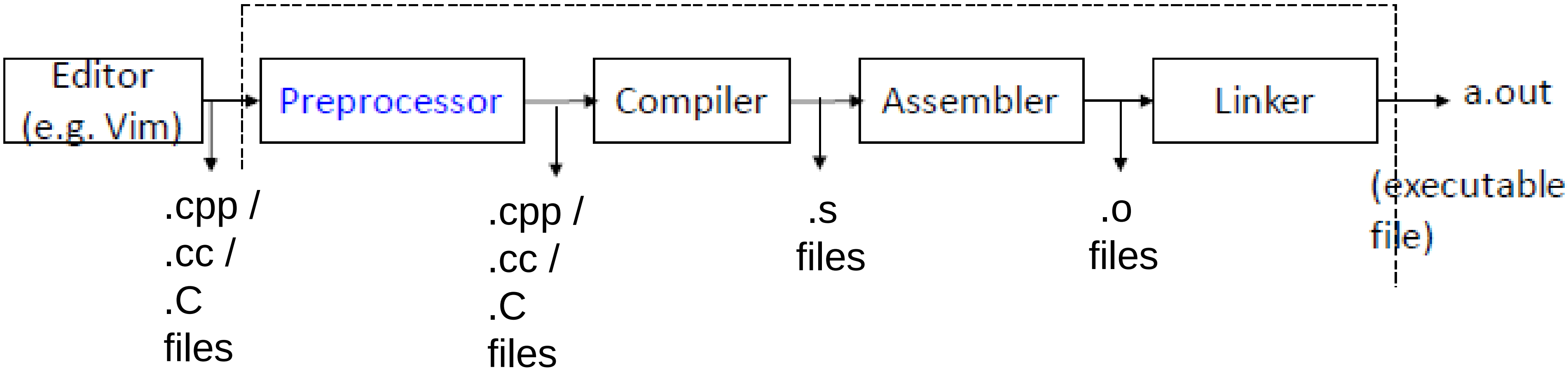
- Create executable



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:
- 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: **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: **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

C++ standard types

- Integer types: `char, short int, int, long int, long long int, bool`
- Float: `float, double, long double`
- `Pointers`: handle to addresses
- `References`: safer than pointers but less powerful
- `void`: nothing

C++ standard types

- Compound types
  - pointers, structs, enums, arrays, etc.
- Modifiers
  - `short, long, signed, unsigned.`

types / representation

`E.g. int x;`

1. What is the set of values this variable can take on in C?

   -2^{31} to $(2^{31} - 1)$

2. How should operations on this variable be handled?
   integer division is different from floating point divisions

   ```
   3 / 2 = 1  //integer division
   3.0 / 2.0 = 1.5  //floating-point division
   ```

3. How much space does this variable take up?

   32 bits

C++ standard types – storage space

Data type	Number of bytes
`char`	1
`short int`	2
`int / long int`	4
`long long int`	8
`float`	4
`double`	8
`long double`	12

- All built-in types are represented in memory as a contiguous set of bytes
- Use sizeof() operator to check the size of a type
  - e.g. `sizeof(int)`

Typedef

– Lets you give alternative names to C data types

– Example:

```
typedef unsigned char BYTE;
```

This gives the name `BYTE` to an `unsigned char` type. Now,

```
BYTE a;
BYTE b;
```

Are valid statements.

Typedef Syntax

`typedef <existing_type> <new_type>;`

– Resembles a definition/declaration without initializer;

  E.g. `int x;`

– Mostly used with user-defined types

User-defined Types

- *Structures* in C/C++ are one way of defining your own type.
- Arrays are compound types but have the *same* type within.
  - E.g. A string is an array of `char`
  - `int arr[]={1,2,3}; arr` is an array of integer types
- Structures let you compose types with *different* basic types within.

Structures - Declaration

Type name

Declarations of fields

```
struct Point{
float xCoordinate;
float yCoordinate;
};
```

– Variable definition:

- `struct Point p1;`
- ```
  struct Point{
   float xCoordinate;
   float yCoordinate;
   }p1;
  ```

p1 is a variable (an object) of type `struct Point`
  ```

Structures - Definition

```c
typedef struct _Point{
float xCoordinate;
float yCoordinate;
}Point;
```

Declarations of fields

New Type name

Canonical type name (long form)

- Variable definition:
  - `Point p1;`

Structures - Usage

- Structure fields are accessed using dot (.) operator
- Example:

```
Point p;
p.xCoordinate = 10.1;
p.yCoordinate = 22.8;
printf("(x,y)=(%f,%f)\n",p.xCoordinate,
p.yCoordinate);
```

Structures - Initialization

– Error to initialize fields in declaration;

```
typedef struct{
float xCoordinate = 10.1;
float yCoordinate = 22.8;
}Point;
```

Data types - quirks

- if no type is given compiler automatically converts it to `int` data type.
  - `signed x;`
- `long` is the only modifier allowed with `double`
  - `long double y;`
- `signed` is the default modifier for `char` and `int`
- Can’t use any modifiers with `float`

Exercise

```
char s[3] = “Hi”;
char *t = “Si”;
int u[3] = {5, 6, 7};
int n=8;
```

Expression	Type	Comments
`s`	`char[3]`	`array of 3 chars`
`t`	`char*`	`address of a char`
`u`	`int[3]`	`array of 3 ints`
`&u[0]`	`int*`	`address of an int`

Exercise

```
char s[3] = “Hi”;
char *t = “Si”;
int u[3] = {5, 6, 7};
int n=8;
```

Expression	Type	Comments
`*&n`	`int`	`value at n`
`*t`	`char`	`data at address Held by t`

Exercise

- Array initializers:

`1. int u[3] = {5, 6};`
Is this valid?
If yes, what is the value held in the third element u[2]?

`2. int u[3] = {5, 6, 7, 8};`
Is this valid?

`3. char s1[]=“Hi”;`
What is the size of s1? (how many bytes are reserved for s1)

`4. char s2[3]=“Si”;`
Is this valid?

Exercise

```
int u[3] = {5, 6, 7};
int* p=u;
p[0]=7;
p[1]=6;
p[2]=5;
```

//Now, u would contain the numbers in reverse order.
`u[0] = 7, u[1]=6, u[2]=5.`

```
char *str = “Hello”;
char* p=str;
p[0]=‘Y’;
```

//Now, what would `str` contain?

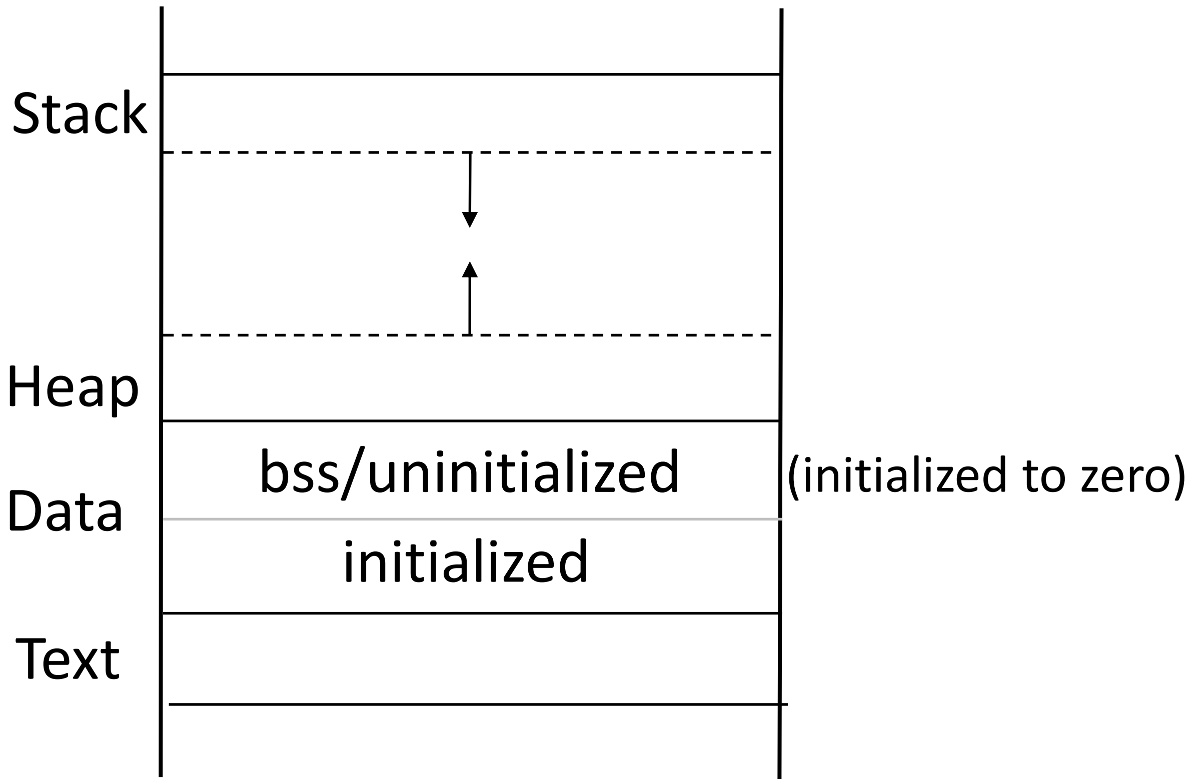
Program layout in memory

- How is a program laid out in memory?
  - Helpful to debug
  - Helpful to create robust software
  - Helpful to customize program for embedded systems

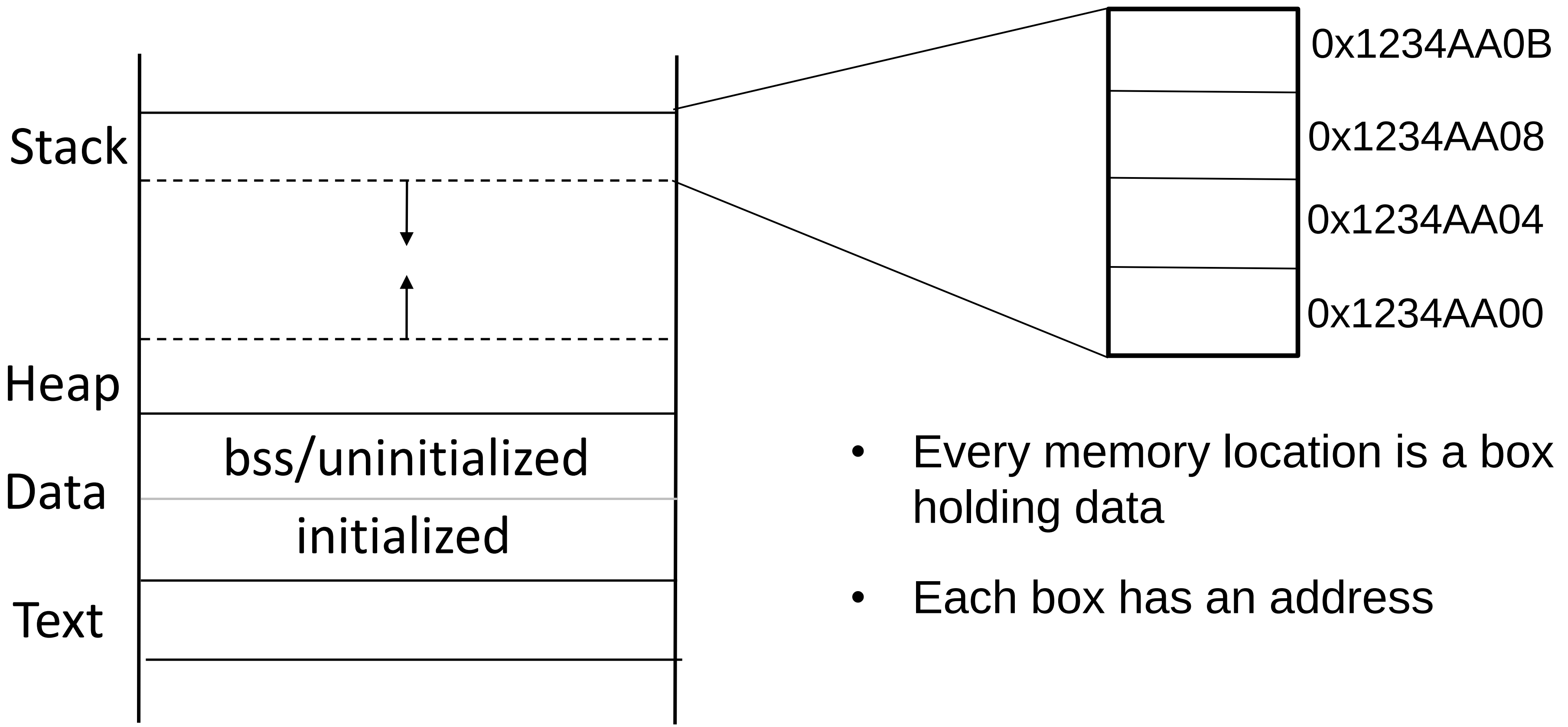
Program Layout in Memory

- A program's memory space is divided into four segments:
  1. Text
     - source code of the program
  2. Data
     - Broken into uninitialized and initialized segments; contains space for global and static variables. E.g. `int x = 7; int y;`
  3. Heap
     - Memory allocated using malloc/calloc/realloc/new
  4. Stack
     - Function arguments, return values, local variables, special registers.

Program Layout in Memory

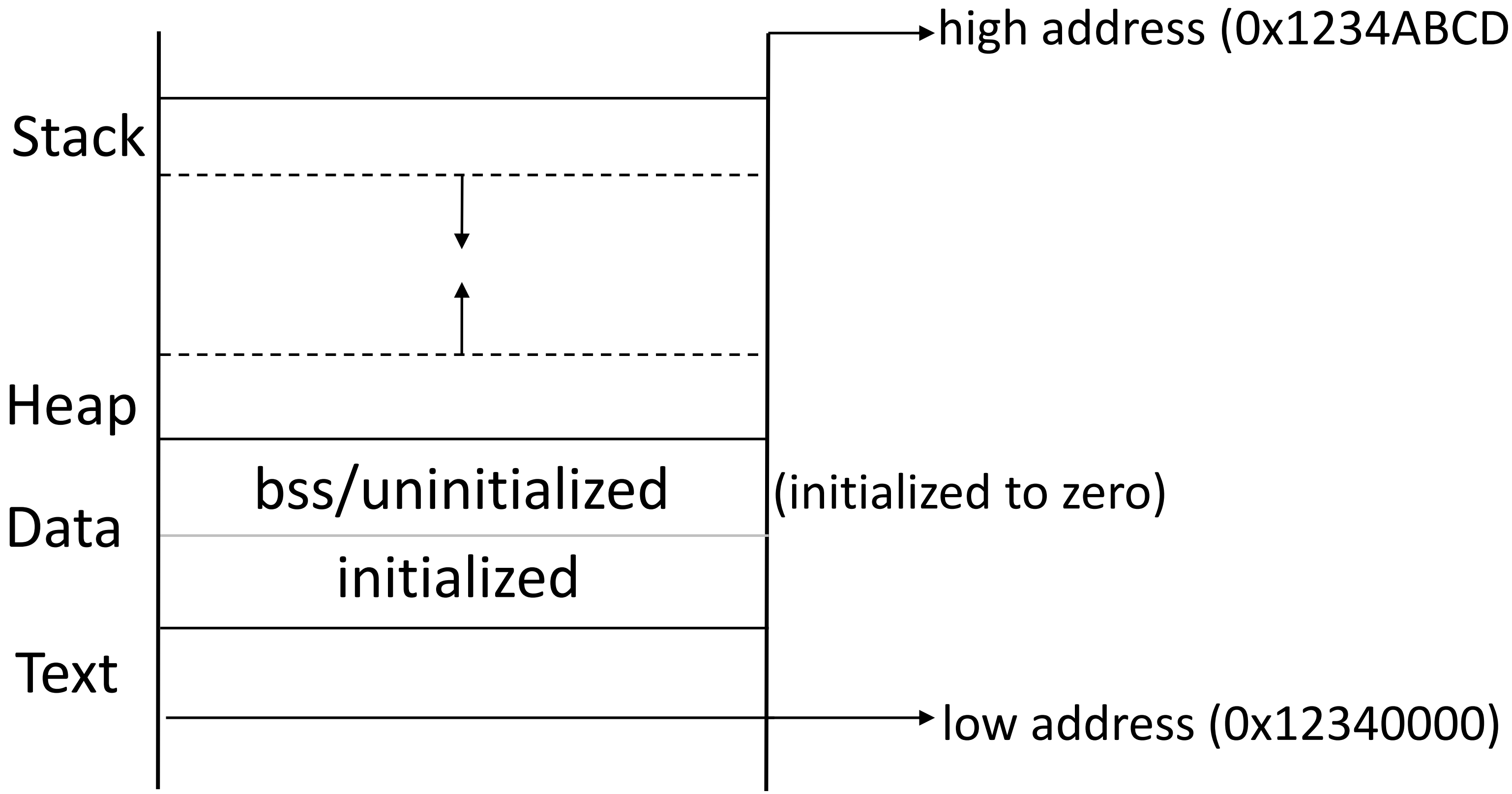


Program Layout in Memory



- Every memory location is a box holding data
- Each box has an address

Program Layout in Memory



Addresses

- Computer programs think and live in terms of memory locations
- Addresses in computer programs are just numbers identifying memory locations
- A program navigates by visiting one address after another

Addresses

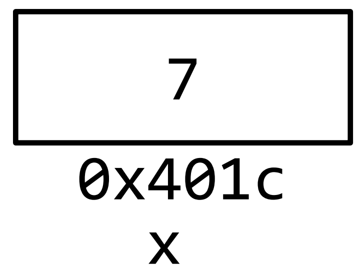
- Humans are not good at remembering numerical addresses.

what are the GPS coordinates (latitude and longitude) of your residence?

- We (humans) choose convenient ways to identify addresses so that we can give directions to a program. E.g. Variables

Handles to Addresses

- Variables
  - Its just a handle to an address / program memory location
- `int x = 7;`



- Read `x` => Read the content at address `0x401C`
- Write `x`=> Write at address `0x401C`