

CS601: Software Development for Scientific Computing

Autumn 2021

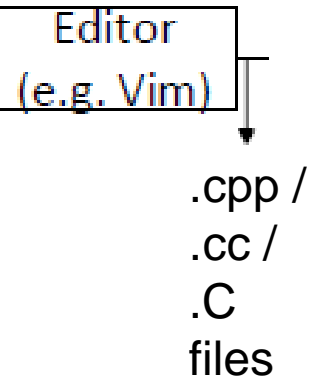
Week2: Program Development Environment,
Minimal C++, Version Control Systems,
Structured Grid

Program Development Environment

- Demo

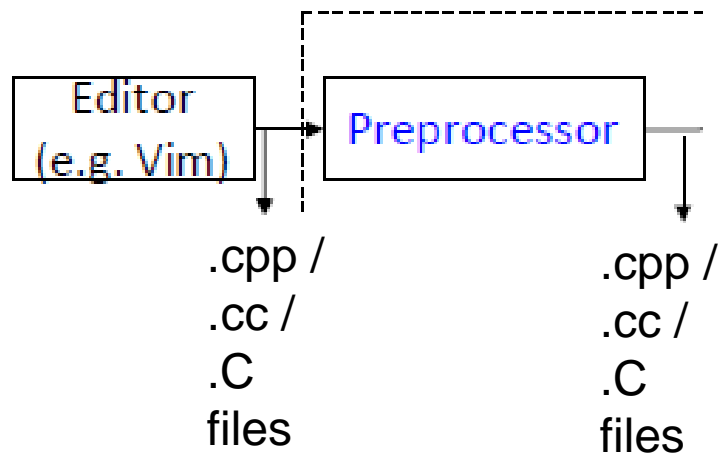
Creating a Program

- Create your c++ program file



Creating a Program

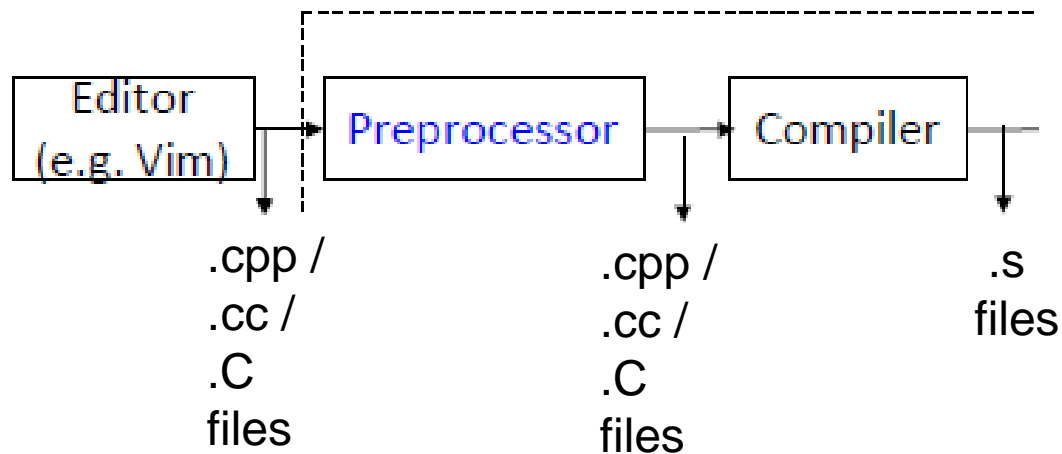
- Preprocess your c++ program file



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

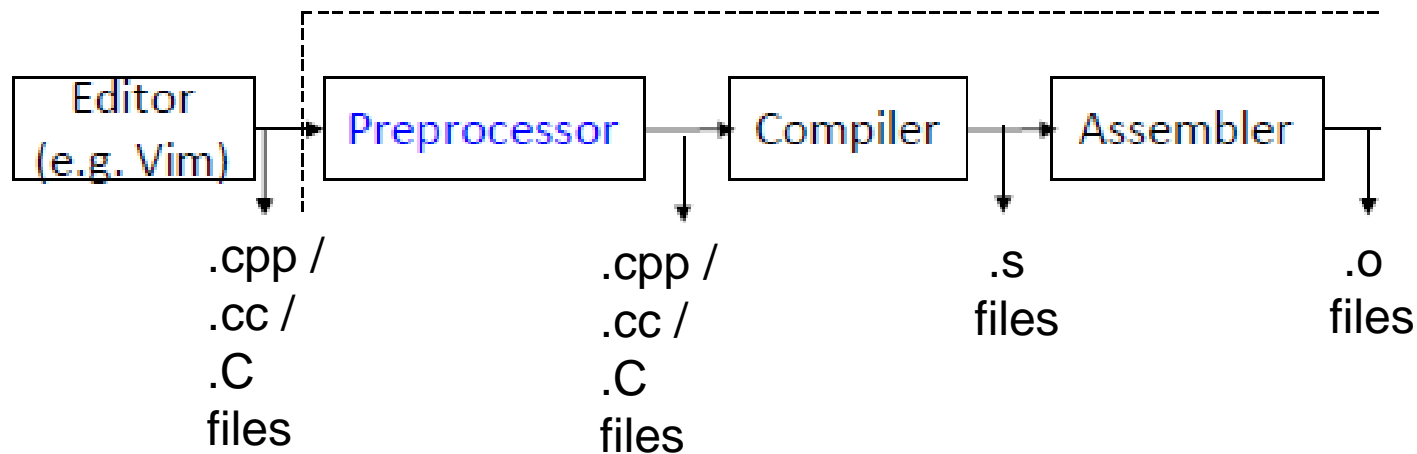
Creating a Program

- Translate your source code to assembly language



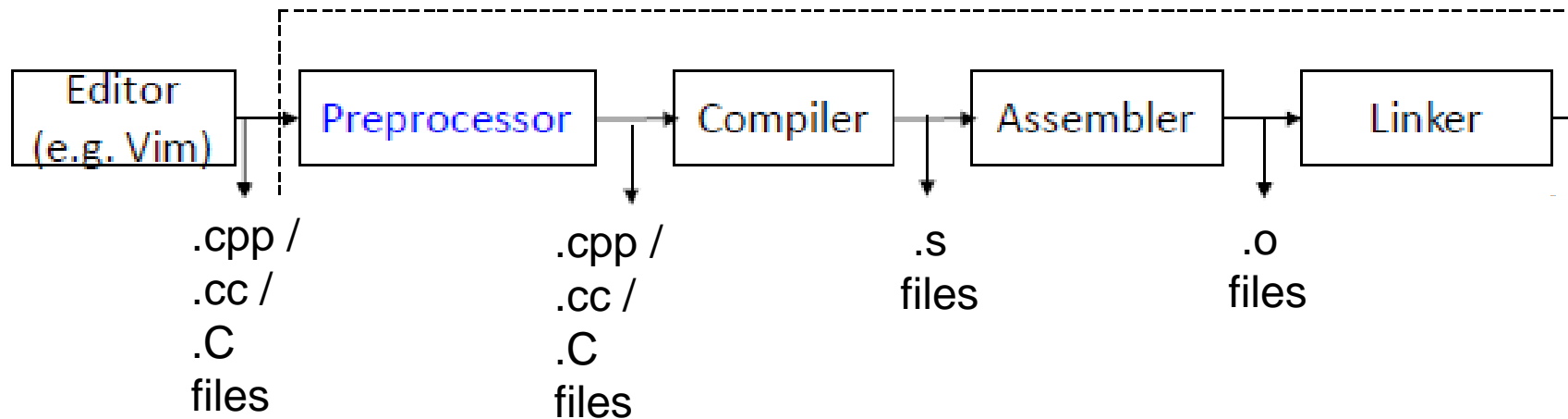
Creating a Program

- Translate your assembly code to machine code



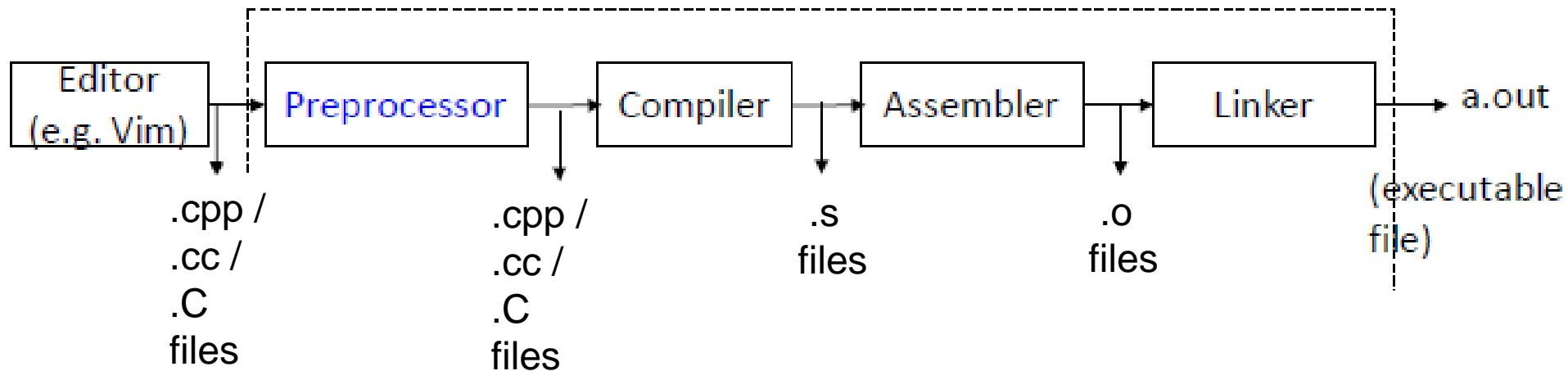
Creating a Program

- Get machine code that is part of libraries



Creating a Program

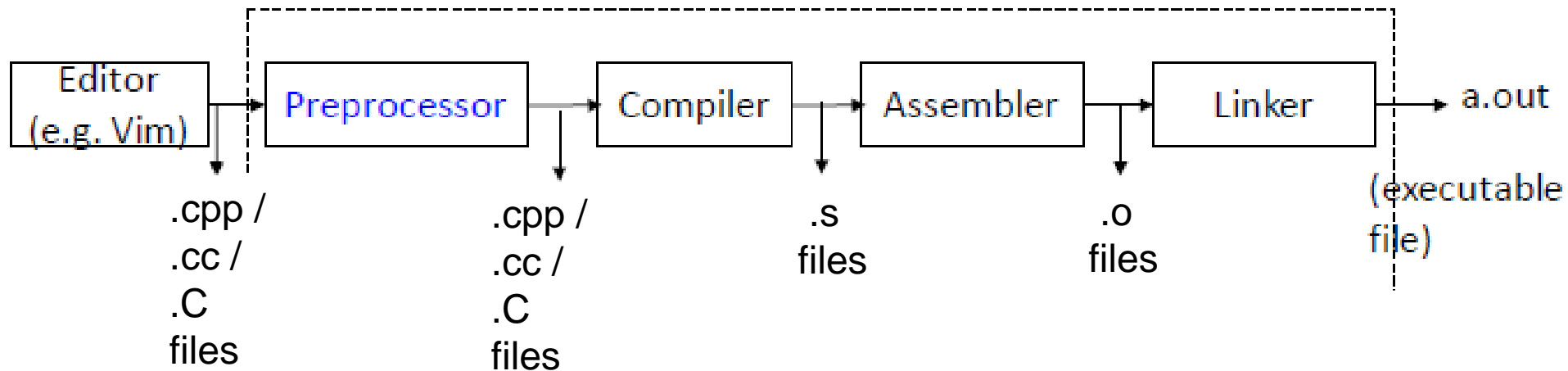
- Create executable



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
 - omyexe - 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

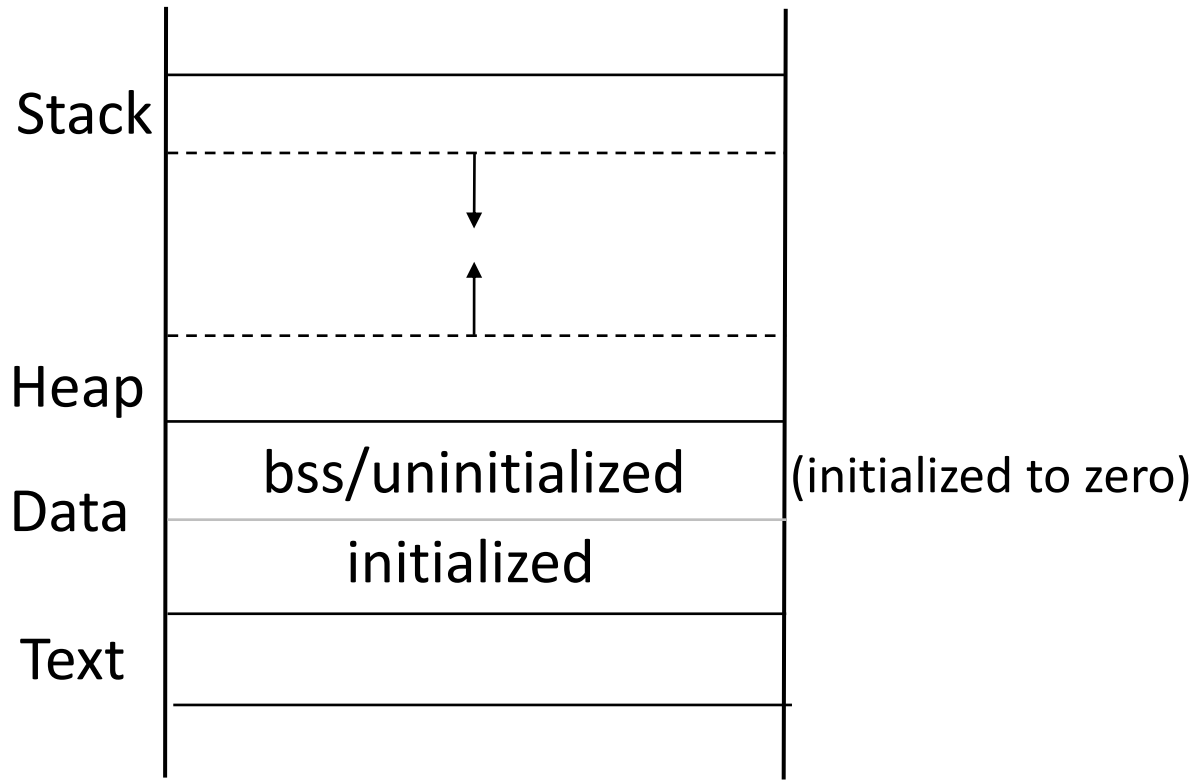
Creating a Program

- `a.out` is a pattern of 0s and 1s laid out in memory
 - sequence of machine instructions
- How is a program laid out in memory?
 - Helpful to debug
 - Helpful to create robust software
 - Helpful to customize program for embedded systems

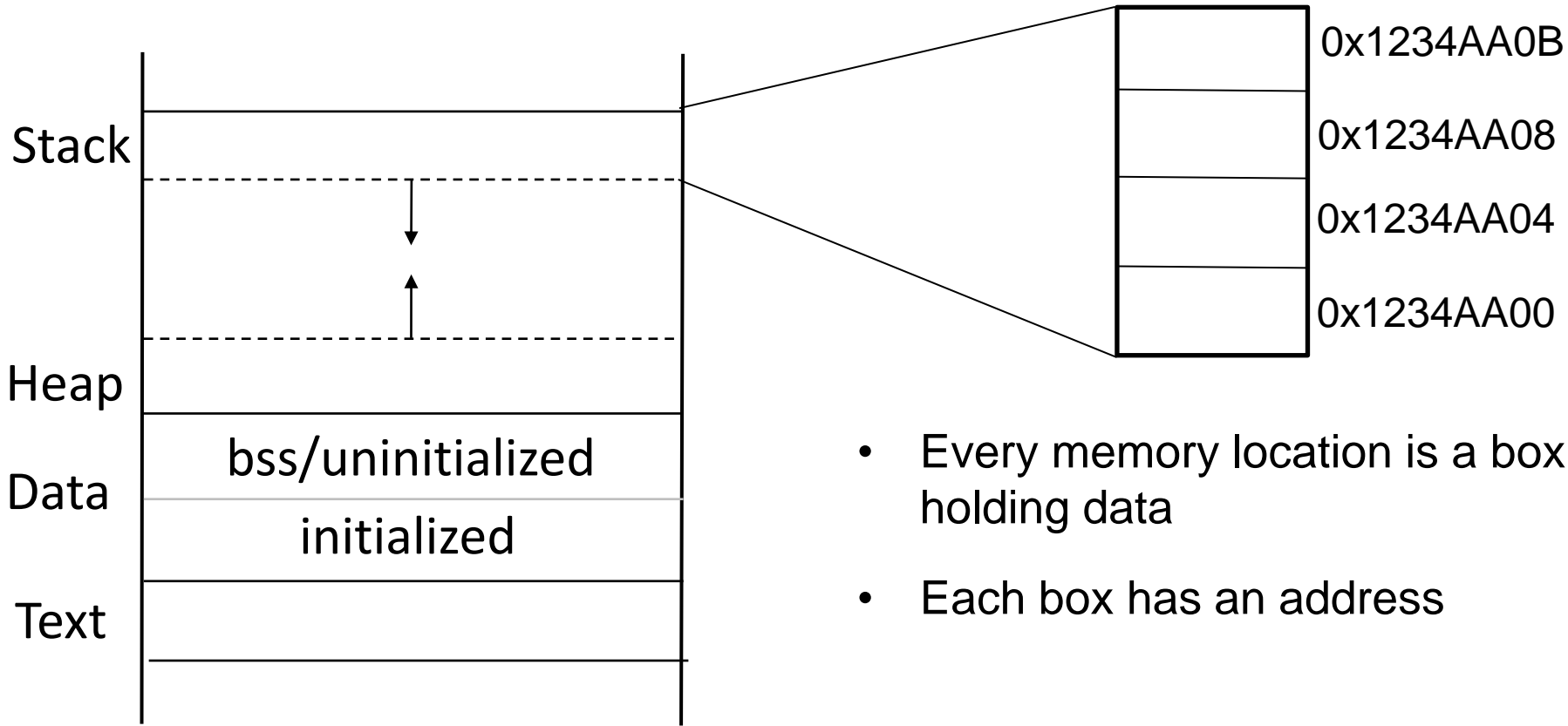
Program Layout in Memory

- 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



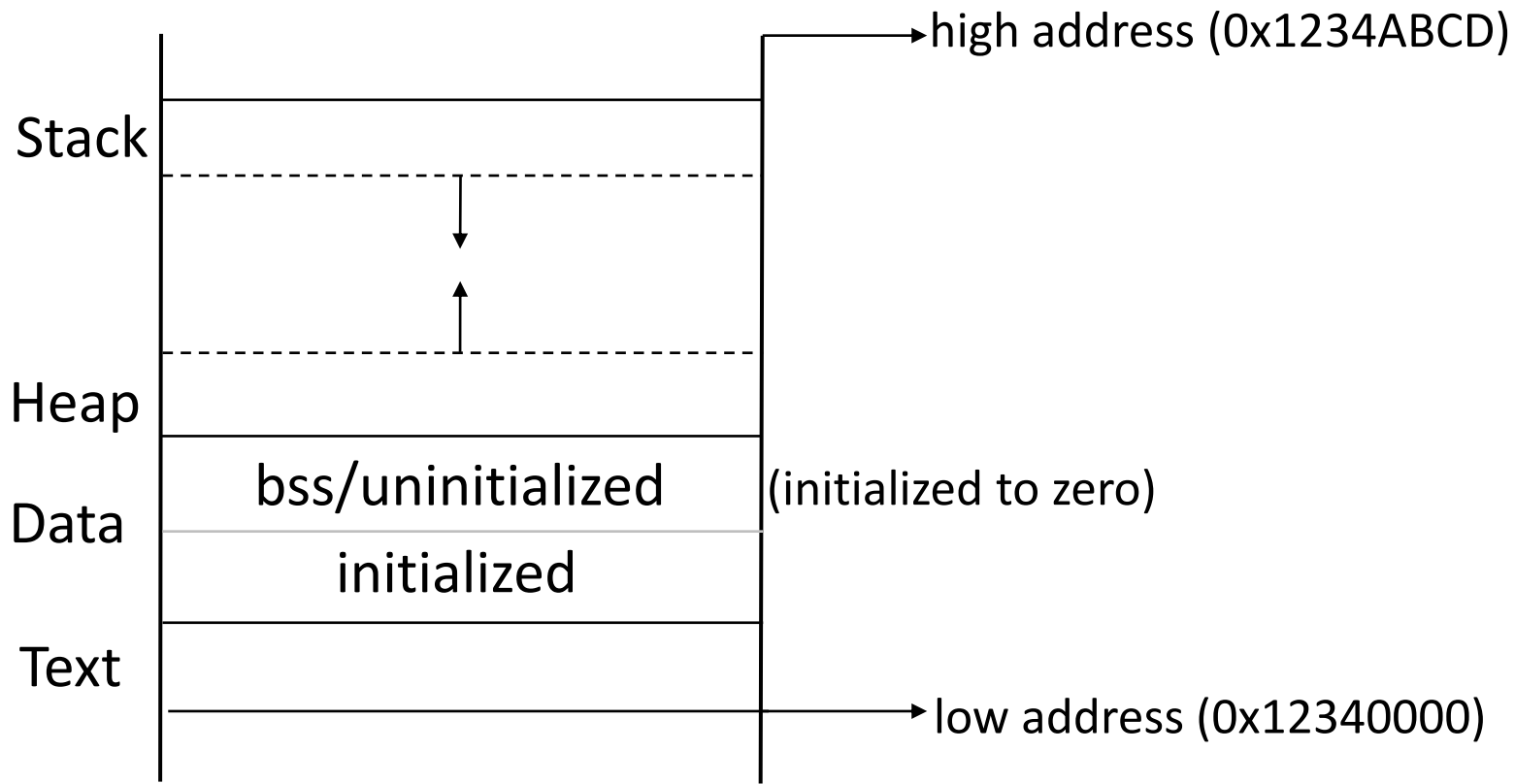
Program Layout in Memory



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

Program Layout in Memory



Addresses

- Humans are not good at remembering numerical addresses.

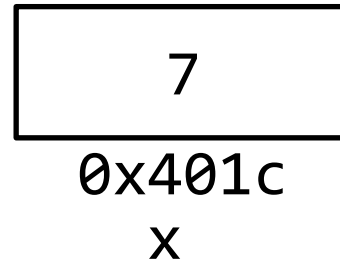
what are the *GPS* coordinates (latitude and longitude) of your residence?

- 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`

```
int x;
```

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

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

2. *How much space does this variable take up?*

32 bits

3. *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
```

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

- Modifiers

- short, long, signed, unsigned.

- Compound types

- pointers, structs, enums, arrays, etc.

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)

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`

Visualizing Addresses

- The *address of* (&) operator fetches a variable's address in C
- &x would return the address of x

```
#include<iostream>
int main(int argc, char* argv[]) {
    int x = 7;
    std::cout<<"Address of x is:"<<&x<<std::endl;
    return 0;
}
```

- prints the Hexadecimal address of x

```
Address of x is:0x7ffd1d5e2844
```

Pointers

- Pointer is a data type that *holds an address*.

`<type>* <pointer_name>;`

- Example:

- `int* p;` //is a variable named p whose type is
//pointer to int OR p is an integer
//pointer

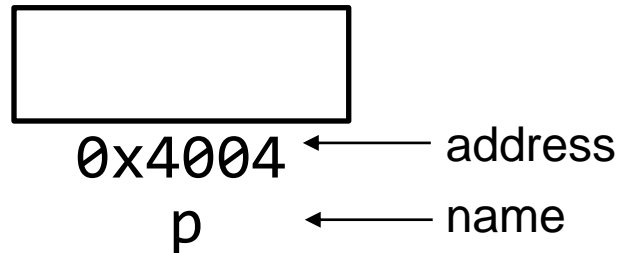
Note that the variable declared is p, *not* *p

- A pointer always stores an address
- `<type>` of the pointer tells us what kind of data is stored at that address
- Example:
 - `int* p;`
declares a pointer variable `p` holding an address, which identifies a memory location capable of storing an integer.

Initializing Pointers

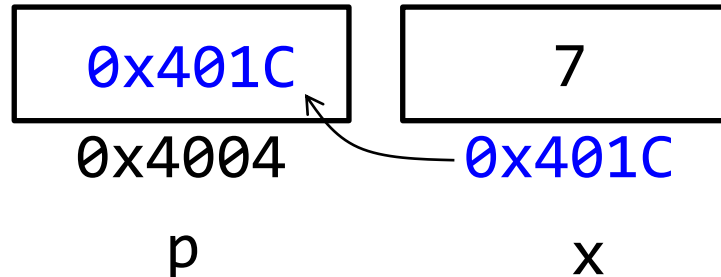
- `int* p;`

Remember `p` is a variable and all variables are just names identifying addresses.



In addition, `p` holds the address of a memory location that stores an integer

- `p=&x;`



- Cannot assign arbitrary addresses to pointers.
- Example:
`int* p=5;`
- Operating system determines addresses available to each program.

The NULL address

- NULL is a special address

- Example

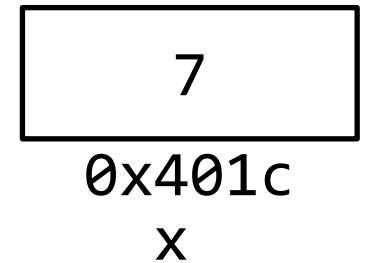
```
int* p=NULL; //p points to nowhere
```

- Useful when it is not yet known where p points to.
- Uninitialized pointers store garbage addresses

Using Pointers

- The *dereference* operator (*)
 - Lets us access the memory location at the address stored in the pointer

```
int x=7;
```

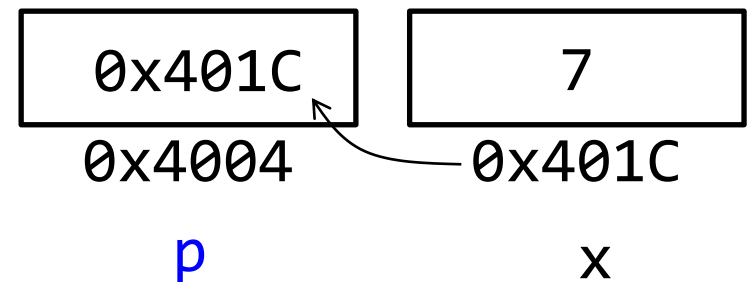


Using Pointers

- The *dereference* operator (*)
 - Lets us access the memory location at the address stored in the pointer

```
int x=7;
```

```
int* p = &x; //p now points to x
```



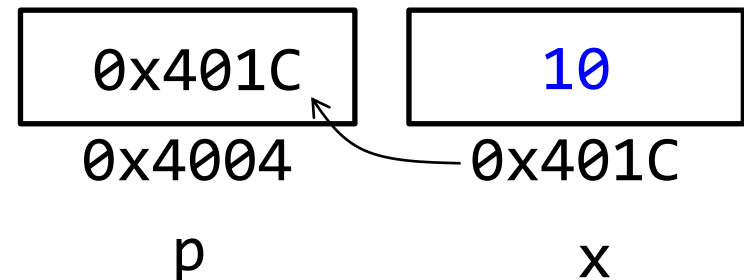
Using Pointers

- The *dereference* operator (*)
 - Lets us access the memory location at the address stored in the pointer

```
int x=7;
```

```
int* p = &x; //p now points to x
```

```
*p = 10; //this is the same as x=10
```



Using Pointers

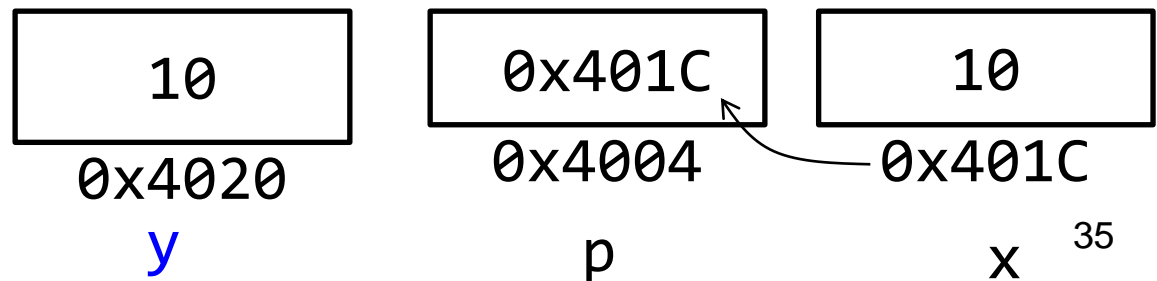
- The *dereference* operator (*)
 - Lets us access the memory location at the address stored in the pointer

```
int x=7;
```

```
int* p = &x; //p now points to x
```

```
*p = 10; //this is the same as x=10
```

```
int y=*p; //this is the same as y=x
```

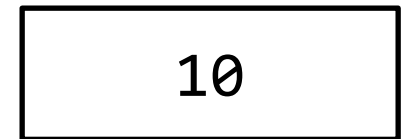


- Pointers as alternate names to memory locations

```
int x=7;  
int *p = &x;
```

x is the name for an address

*p is the name for an address



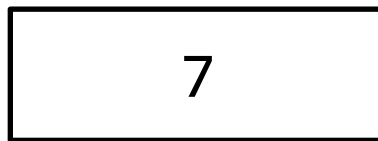
0x401c

x

*p

- Pointers as “dynamic” names to memory locations

```
int x=7; //x always names the location 0x401C  
int *p = &x; // *p is now another name for x
```



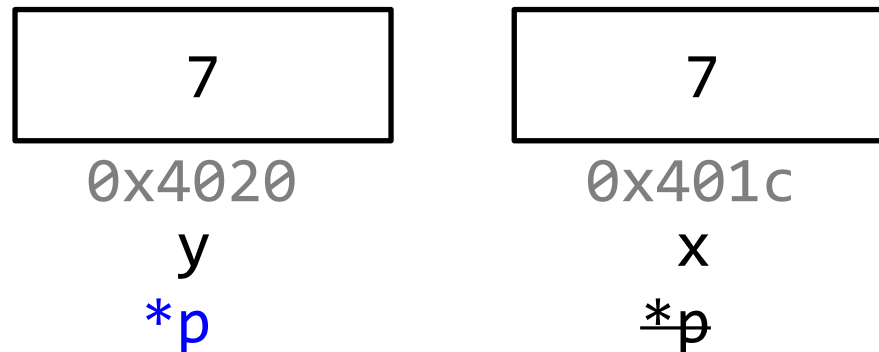
0x401c

x

*p

- Pointers as “dynamic” names to memory locations

```
int x=7; //x always names the location 0x401C  
int *p = &x; //*p is now another name for x  
int y = *p //like saying y=x  
p = &y; //*p is now another name for y
```



Pointers to Different Types

- What can pointers point to? any data type!
 - Basic data types – we have seen these.
 - Structures – Next set of slides.
 - Pointers! and
 - Functions

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

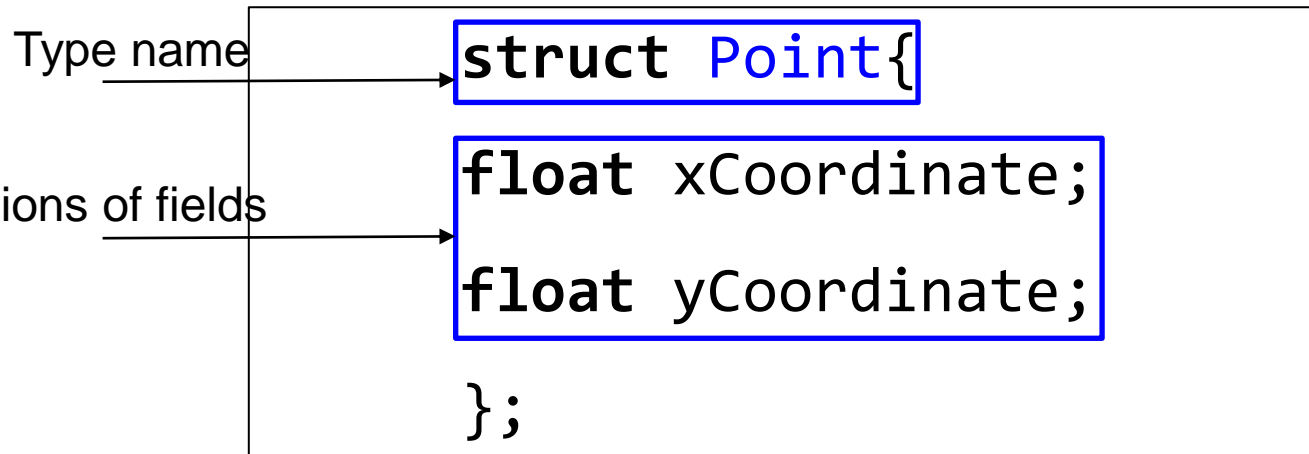
```
typedef [existing_type] [new_type];
```

- Resembles a declaration without initializer;
E.g. `int x;`
- Mostly used with user-defined types

User-defined Types

- *Structures* in 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

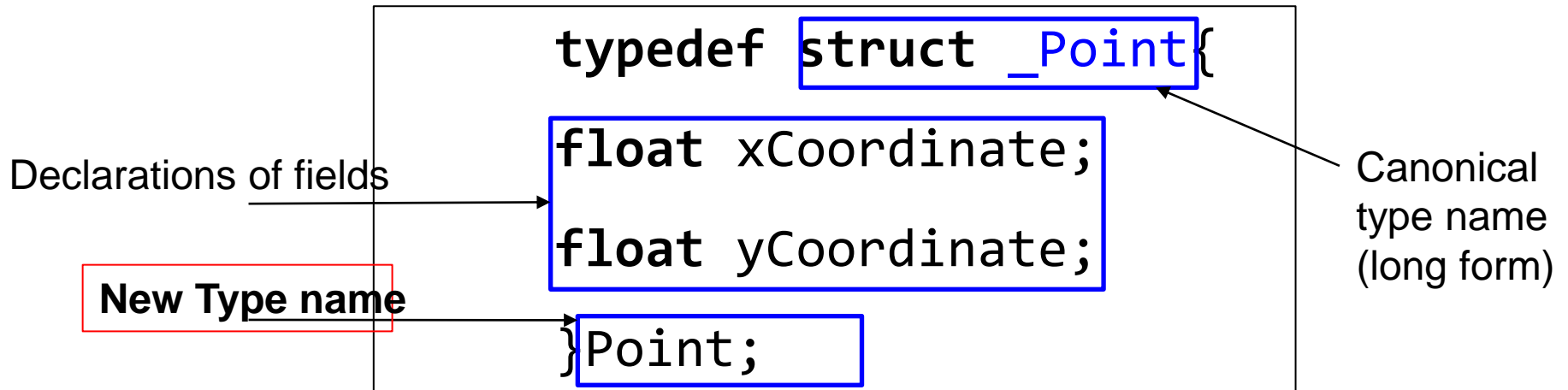


– Variable definition:

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

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

Structures - Definition



- Variable definition:
 - `Point p1;`

Structures - Usage

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

```
Point p;
```

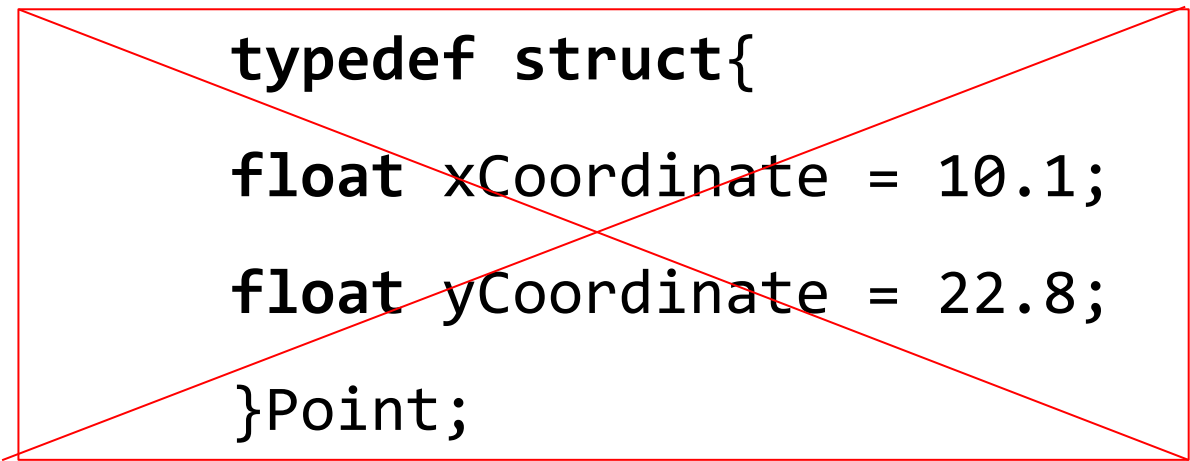
```
p.xCoordinate = 10.1;
```

```
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;
```

Structures - Initialization

- `Point p1={10.1,22.8};`
- `Point p2={.x=10.1,.y=22.8};`
//Introduced in C99.
//Designated initializers
//Best-way

Pointers to Structures

```
typedef struct {  
    int year;  
    char model;  
    float acceleration; //0-60mph in seconds  
}Car;
```

```
Car t1 = {.year = 2017, .model = 'S',  
    .acceleration = 2.8 };
```

```
Car * pt1 = &t1; //now you can use *pt1  
anywhere you use t1
```



```
(*pt1).acceleration = 2.3;  
(*pt1).year = 2019;  
(*pt1).model = 'X';  
float avg_acceleration = ((*pt1).acceleration  
+ (*pt2).acceleration) / 2.0;
```

We can also use the -> operator to access structure members.

```
pt1->acceleration = 2.3;  
pt1->year = 2019;  
pt1->model = 'X'  
float avg_acceleration = (pt1->acceleration +  
pt2->acceleration) / 2.0;
```

Pointer Chains

```
int x = 7;  
int *p = &x; //p points to x; *p is same as x.  
  
int ** q=&p; //q is a pointer to pointer to int  
  
*q is same as p.  
*(*q) is the same as *p, which is same as x
```

Address of (&) operator and Type

- Adding & to a variable adds * to its type
- Example:
 - if a is an int, then &a is an int*
 - if b is an int*, then &b is an int**
 - if c is an int**, then &c is an int***
 - ...

Dereference (*) operator and Type

- Adding * to a variable subtracts * from its type
- Example:
 - if a is an int*, then *a is an int
 - if b is an int**, then *b is an int*
 - if c is an int***, then *c is an int**
 - ...

Pointer Arithmetic

```
int y = 1040;  
int* p = &y;
```

- What does `*(p+1)` mean?
 - Data at “one element past” `p`
- What does “one element past” mean?
 - `p` is a pointer, so holds the address of a memory location
 - `p` is an `int` pointer, so that memory location holds an integer
 - `p+1` is interpreted as **address of the next integer**

Pointer Arithmetic

- Our representation of

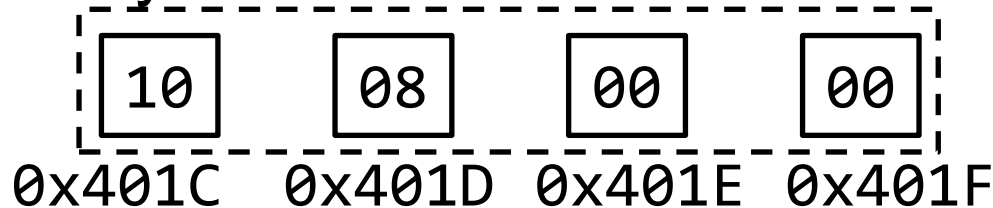
```
int y=2064;  
int* p = &y;
```

0x401C
0x1000
p

2064
0x401C
y

Pointer Arithmetic

- ints occupy 4 bytes. 0x401C is the address of the first byte*:

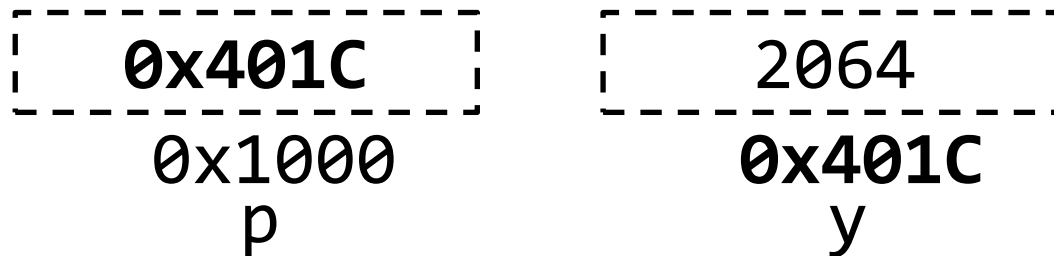


*2064 = 0x810 (=0x00,00,08,10 when written using 8 digits and x86 is little-endian)

- (**p*) = data at 0x401C
 - returns the correct value of 2064 and not 0x10. Why?*

Pointer Arithmetic

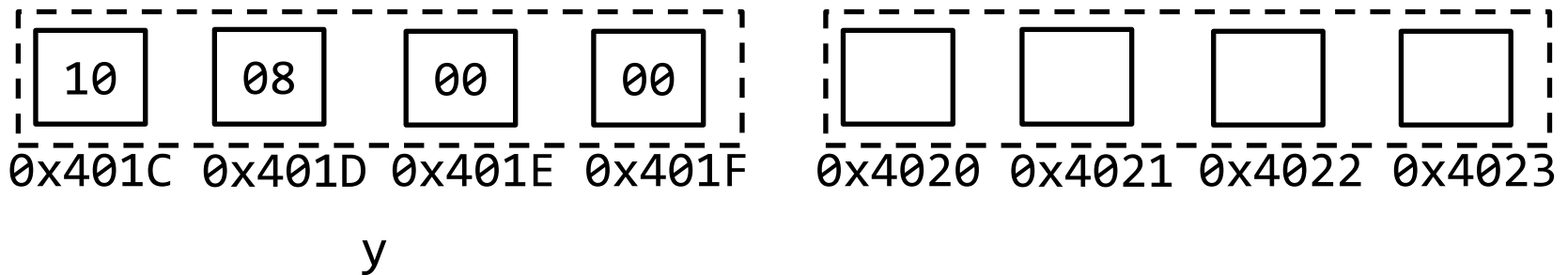
- $(p+1)$ gets the “address of the next integer”



What is the address of the next integer?

Pointer Arithmetic

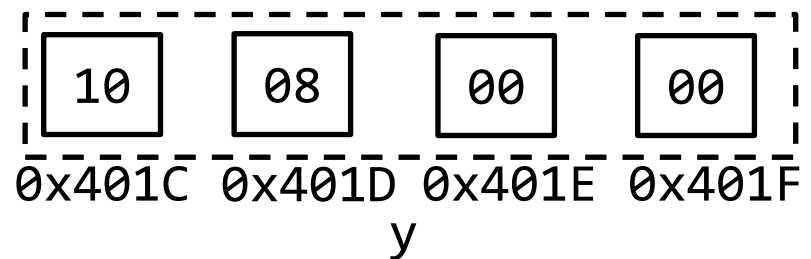
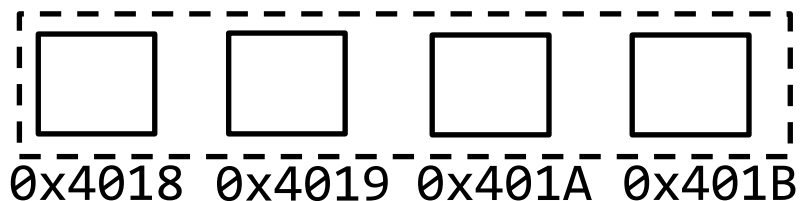
- What is the address of the next integer?
 - Add 4 to current value of p (0x401C) = 0x4020



Pointer Arithmetic

- $(p-1)$ computes the address before y

```
int y=2064;  
int* p = &y;
```



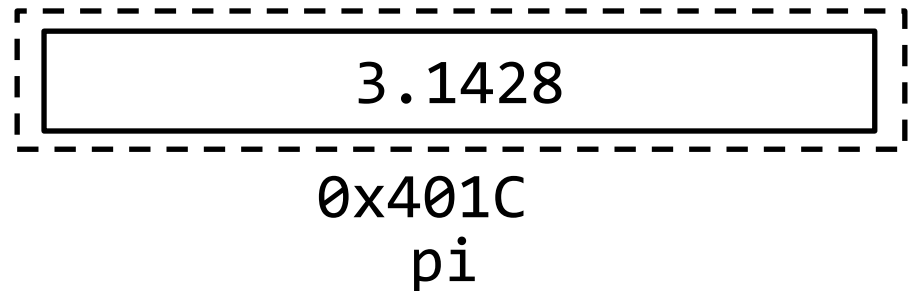
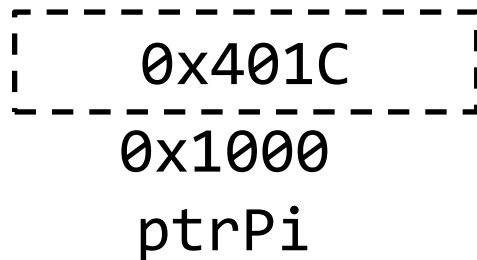
subtract 4 from the current value of p ($0x401C$) = $0x4018$

- Similarly we can add/subtract any number to/from a pointer variable.
- Compare to a specific address (E.g. `if(p == NULL)`)

Pointer Arithmetic

- Pointer to double (double occupies 8 bytes)

```
double pi=3.1428;  
double* ptrPi = &pi;
```



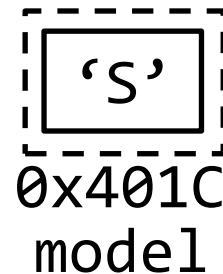
What is the address computed for $(ptrPi+1)$? `0x4024`

What is the address computed for $(ptrPi-1)$? `0x4014`

Pointer Arithmetic

- Pointer to char

```
char model='S';  
char* ptrModel = &model;
```

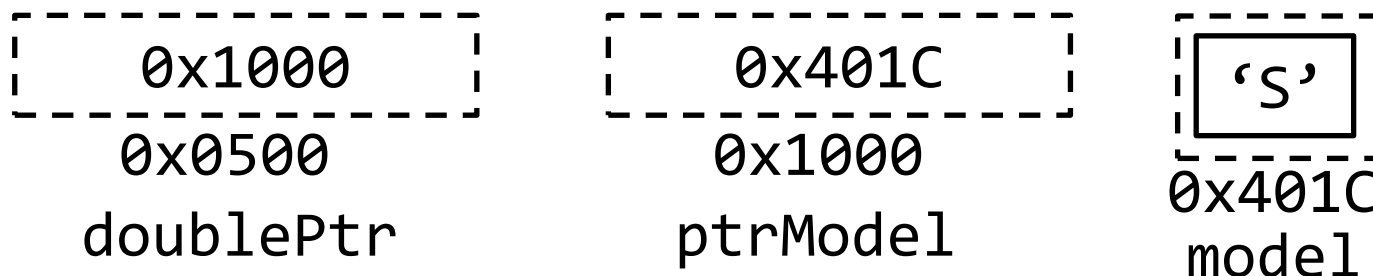


What is the address computed when we do `(ptrModel+1)`?

Pointer Arithmetic

- Pointer to pointer

```
char model='S';  
char* ptrModel = &model;  
char** doublePtr = &ptrModel;
```



Bonus: what is the address computed when we do $(\text{doublePtr}+1)$? (assuming we are using 32-bit machines)

C-style Arrays

Declaring arrays:

```
type <array_name>[<array_size>];  
int num[5];
```

Initializing arrays:

```
int num[3]={2,6,4};  
int num[]={2,6,4}; //array_size is not  
required.
```

Accessing arrays:

num[0] accesses the first integer

num[1] accesses the second integer and so on..

Arrays

- Another data type!
 - Array of ints, structs etc.
 - Array of chars (strings in C)
- Work a little bit like pointers

```
int a[10]={11,21,31,41,51,61,71,81,91,101};  
//array of 10 integers
```

11	21	31	41	51	61	71	81	91	101
----	----	----	----	----	----	----	----	----	-----

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

10 elements **guaranteed** to be next to each other in
memory

Arrays

```
int a[10]={11,21,31,41,51,61,71,81,91,101};
```

11	21	31	41	51	61	71	81	91	101
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]



a

0x4001

- 0x4001 is starting address of the array = address of a[0] = **&a[0]**

- Fetch the address of a = &a = 0x4001

Arrays

- Array name in C is the address of the first element of the array

```
int a[10]={1,2,3,4,5,6,7,8,9,10};
```

Therefore, `a == &a[0]`

a, &a, &a[0] are the same and have values 0x4001.

Arrays

- Array name in C is the address of the first element of the array

Array names are converted to pointers (in most cases) but a's type is not a pointer.

```
int* ptr=a; //ptr holds the address of the  
first element of the array (also &a[0]).
```

```
ptr[1] gets a[1]
```

```
ptr[2] gets a[2]
```

```
...
```

How is this possible?

Arrays

- Array dereferencing operator [] is implemented in terms of pointers.
 - $a[3]$ means: start at the address a , go forward 3 elements, fetch the *data at* that address.
 - In pointer arithmetic syntax, this is equivalent to:

$*(a+3)$

So,

$a[0]$ really means: $*(a+0)$

$a[1]$ really means: $*(a+1)$

Arrays

- So, when

```
int* ptr = a;
```

- `ptr[0]` really means `*(ptr+0)`, which is the same as `*(a+0)`, which is `a[0]`
- `ptr[1]` really means `*(ptr+1)`, which is the same as `*(a+1)`, which is `a[1]`
- ...

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
<code>*&n</code>	<code>int</code>	value at n
<code>*t</code>	<code>char</code>	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;
```

//At this line, 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';
```

//At this line, what would str contain?

Dynamic Memory Allocation

- Statically allocated arrays:

```
int arr[3]={1, 2, 3};
```



Must be known
at compile time

- Can't expand arr once defined

Dynamic Memory Allocation

- What if we don't know the array length?
 - Option 1: Variable length arrays.
Not an option with -Wvla, -Wall, and -Werror flags
 - Option 2: use heap.
Preferred option

Dynamic Memory Allocation

- We interact with heap using
 - new

“Give us X bytes of storage space (memory) from the heap so that we can use it to store data”
 - delete

“take back this memory so that it can be used for something else”

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: `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
```

The main Function

- 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

- 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;  
}
```

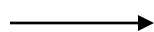
Functions

- 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

- 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

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

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

pi and ran are copied to
a and b

Pass-by-value

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

Functions

- Calling: `function_name(parameters);`

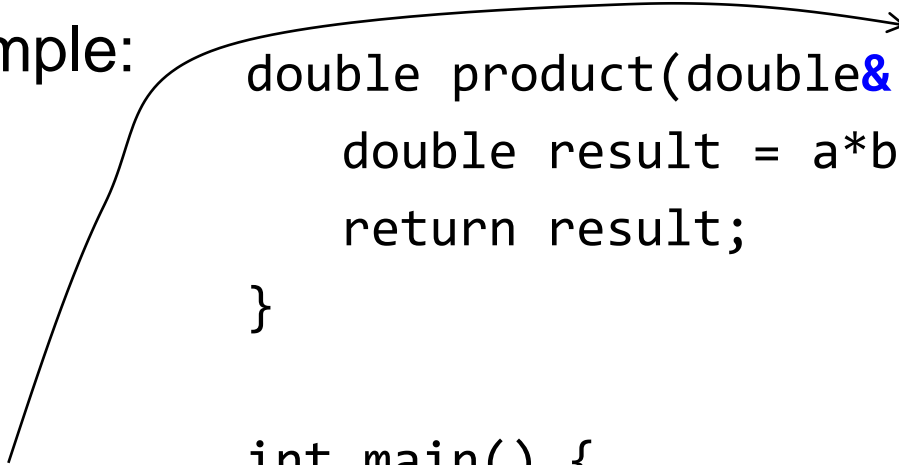
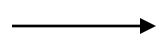
- Example:

```
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 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

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
```

```
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"

Exercise

- Write a C++ program with the following requirements:
 - User should be able to provide the dimension of two vectors (do not use C++ vectors from STL)
 - The program should allocate two vectors of the required size and initialize them with meaningful data
 - The program should compute the scalar product of the two vectors and print the result

Discretization

- Cannot store/represent infinitely many continuous values
 - To model turbulent features of **flow through a pipe**, say, I am interested in velocity and pressure at all points in a region of interest
 1. Represent region of interest as a mesh of small discrete **cells** - **discretization spacing**
 2. Solve equations for each cell

Example: diameter of the pipe = 5cm
length=2.5cm
discretization spacing = 0.1mm
(volume of cylinder = $\pi r^2 h$)

Exercise: *how many variables do you need to declare?*

Discretization

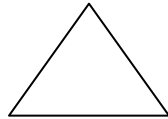
- All problems with ‘continuous’ quantities don’t require discretization
 - Most often they do.
- When discretization is done:
 - How refined is your discretization depends on certain parameters: step-size, cell shape and size. E.g.
 - Size of the largest cell (PDEs in FEM),
 - Step size in ODEs
 - Accuracy of the solution is of prime concern
 - Discretization always gives an approximate solution. Why?
 - Errors may creep in. Must provide an estimate of error.

Accuracy

- Discretization error
 - Is because of the way discretization is done
 - E.g. use more number of rays to minimize discretization error in ray tracing
- Solution error
 - The equation to be solved influences solution error
 - E.g. use more number of iterations in PDEs to minimize solution error
- Accuracy of the solution depends on both solution and discretization errors
- Accuracy also depends on cell shape

Cell Shape

- 2D:

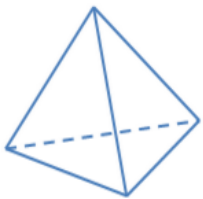


triangle

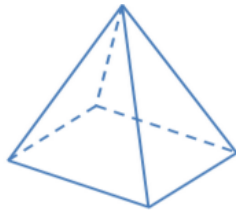


quadrilateral

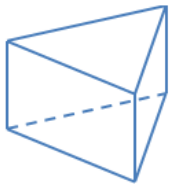
- 3D: triangular or quadrilateral faced. E.g.



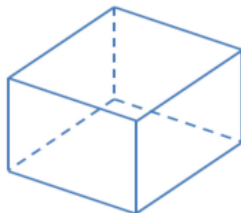
Tetrahedron



Pyramid





Triangular Prism



Hexahedron

Tetrahedron: 4 vertices, 4 edges, 4  faces

Pyramid: 5 vertices, 8 edges, 4  and 1  face

Triangular prism: 6 vertices, 9 edges, 2  and 3  faces

Hexahedron: 8 vertices, 12 edges, 6  faces

source: wikipedia

Error Estimate

- You will have to deal with errors in the presence of discretization
 - Providing error estimate is necessary
- *Apriori* error estimate
 - Gives insight on whether a discretization strategy is suitable or not
 - Depends on discretization parameter
 - Properties of the (unknown) exact solution
 - Error is bound by: Ch^p where, C depends on exact solution, h is discretization parameter, and p is a fixed exponent. *Assumption: exact solution is differentiable, typically, $p+1$ times.*

Error Estimate

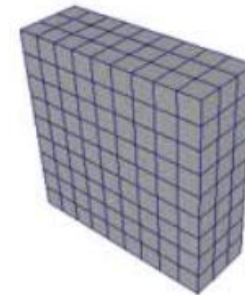
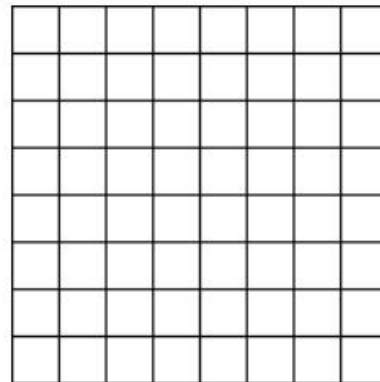
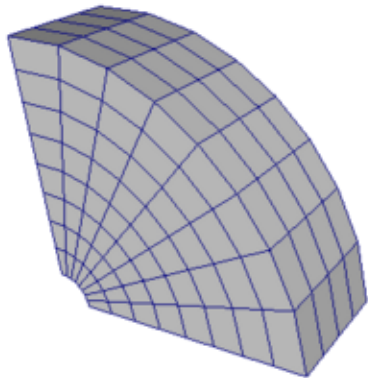
- *Aposteriori* error estimate
 - Is estimation of the error in computed (Approximate) solution and does not depend on information about exact solution
 - E.g. Sleipner-A oil rig disaster

Exercise

- *does increasing mesh size always yield better accuracy?*
- *does decreasing cell size always yield better accuracy?*
- *How does changing mesh size affect computational cost?*
- *How does changing cell size affect computational cost?*

Structured Grids

- Have regular connectivity between cells
 - i.e. every cell is connected to a predictable number of neighbor cells
- Quadrilateral (in 2D) and Hexahedra (in 3D) are most common type of cells
- Simplest grid is a rectangular region with uniformly divided rectangular cells (in 2D).



Structured Grids – Problem Statement

- Given:
 - A geometry
 - A partial differential equation
 - Initial and boundary conditions
- Goal:
 - Discretize into a grid of cells
 - Approximate the PDE on the grid
 - Solve the PDE on the grid

Structured Grids - Representation

- Because of regular connectivity between cells
 - Cells can be identified with indices (x,y) or (x,y,z) and neighboring cell info can be obtained.
 - How about identifying a cell here?

Given:

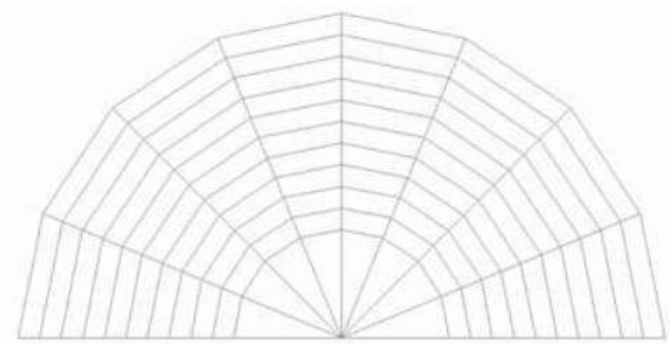
ξ = (“Xi”) radius

η = (“Eta”) angle

Compute:

$$x = \left(\frac{1}{2} + \xi \right) \cos(\pi\eta)$$

$$y = \left(\frac{1}{2} + \xi \right) \sin(\pi\eta)$$



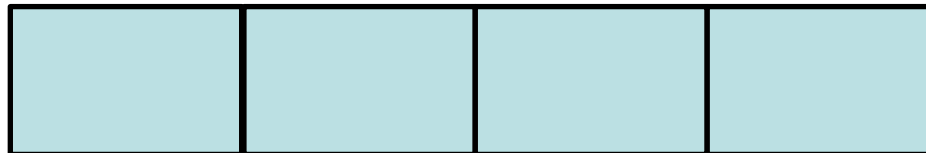
Structured Grids - Representation

- In next class....
 - Grid generation and grid types
 - Partial Differential Equations (PDEs)
 - Solving PDEs (turning PDEs into large set of algebraic equations)
- Now.

2D Arrays

- 1D array gives us access to *a row* of data
- 2D array gives us access to *multiple rows* of data
 - A 2D array is basically an *array of arrays*
- Consider a fixed-length 1D array:
`int arr1[4];`//defines array of 4 elements; every element is an integer. Reserves contiguous memory to store 4 integers.

`arr1[0] arr1[1] arr1[2] arr1[3]`



Starting addr:

100

104

108

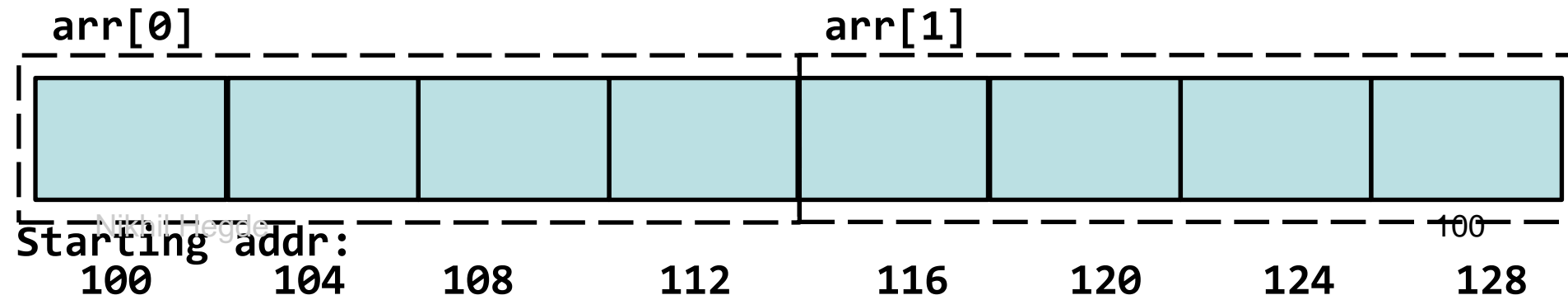
112

2D Arrays (fixed-length)

- Consider a fixed-length 2D array (*array of arrays*). Think:
array of integers => every element is an `int`
array of characters => every element is a `char`
array of array => every element is an *array*
- Example:

```
int arr[2][4];
```

//defines array of 2 elements; every element is an array of 4 integers. Therefore, reserves contiguous memory to store 8 integers



2D Arrays (on heap)

- What if we don't know the length of the array upfront?
E.g. A line in a file contains number of people riding a bus every trip. Multiple trips happen per day and the number can vary depending on the traffic.

Day1 numbers: 10 23 45 44

Day2 numbers: 5 33 38 34 10 4

Day3 numbers: 9 17 10

.....

DayN numbers: 13 15 28 22 26 23 22 21

//we need array arr of N elements; every element is an array of M integers. Both N and M vary with every file input.

2D Arrays (on heap)

1. First, we need to create an array `arr2D` of N elements.
So, get the number of lines in the input file.
 - But what is the *type* of every element? - array of M elements, where every element is an integer (i.e. every element is an integer array). `int *`
 - What is the type of `arr2D`? (array of array of integers)
Think:
type of an integer => `int`
type of array of integers => `int *`
(append a `*` to the type for every occurrence of the term array)
type of array of array of integers => `int **`

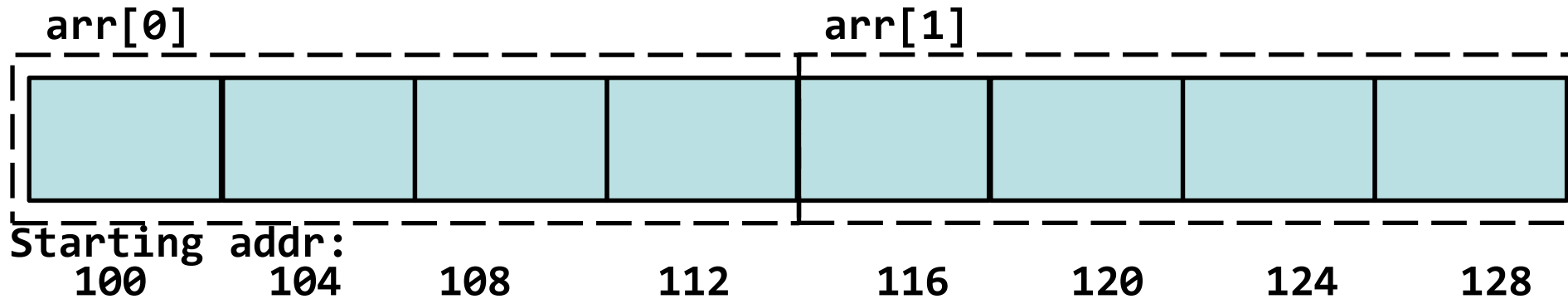
2D Arrays (on heap)

1. First, we need to create an array `arr2D` of N elements. So, get the number of lines in the input file.
 - What is the type of `arr2D`? (`int **`)

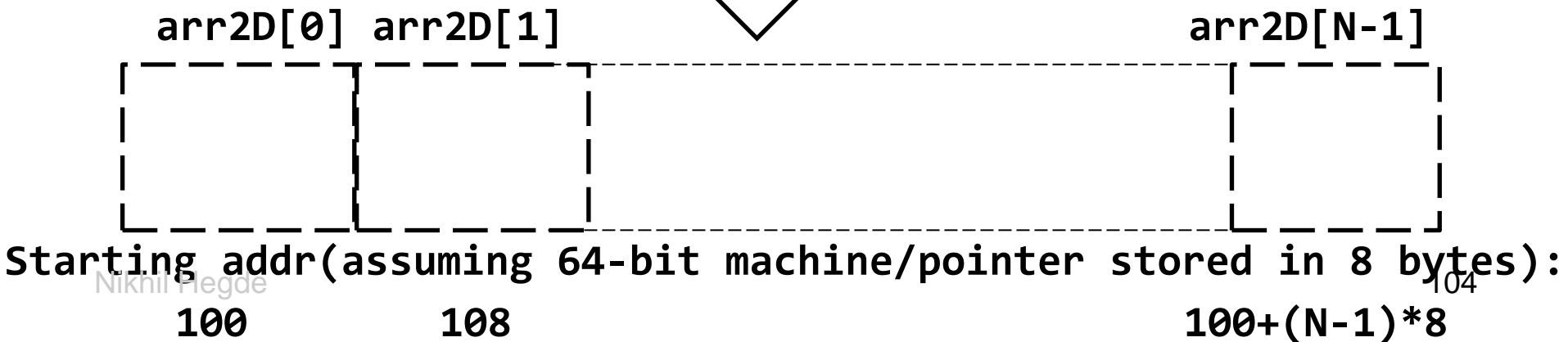
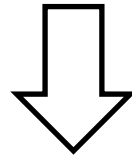
```
int N = GetNumberOfLinesFromFile(fileName);
```

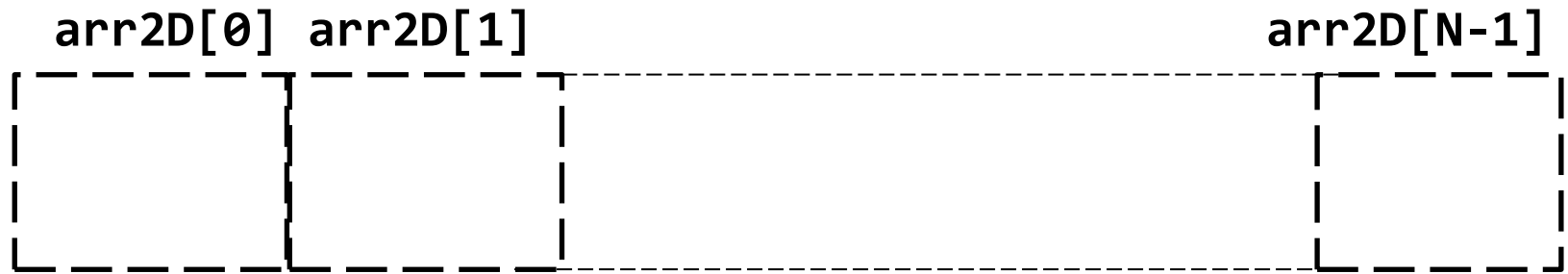
```
int** arr2D = new int*[N];
```

Recall boxes with dashed lines in `int arr[2][4];`



```
int N = GetNumberOfLinesFromFile(filename);  
int** arr2D = new int*[N];
```

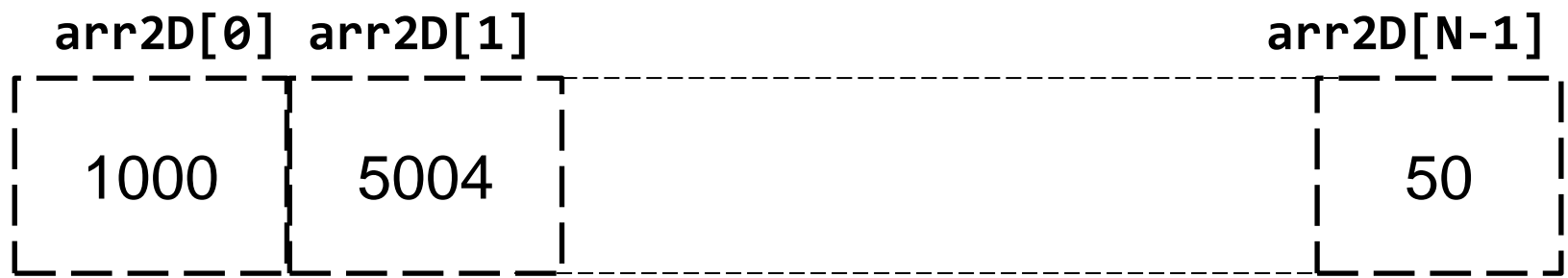




Starting addr(assuming 64-bit machine/pointer stored in 8 bytes):
 100 108 $100 + (N-1) * 8$

2. `arr2D[0]`, `arr2D[1]`, etc. are not initialized. They hold garbage values. How do we initialize them?

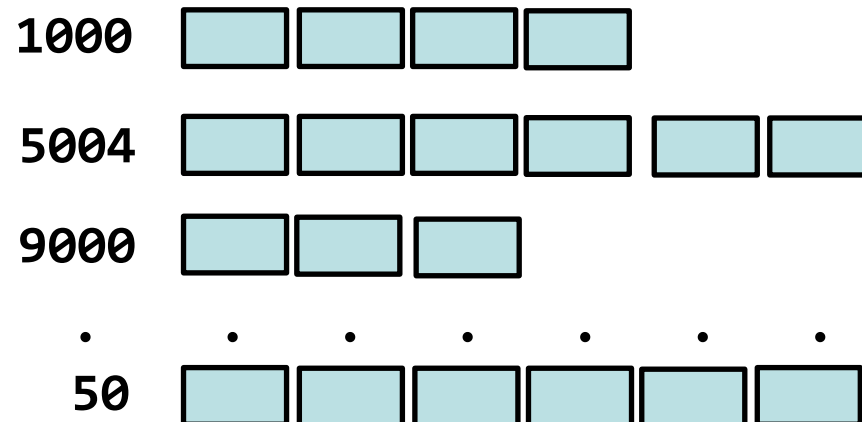
```
for(int i=0;i<N;i++) {
    char* line = ReadLineFromFile(filename);
    int M = GetNumberOfIntegersPerLine(line);
    arr2D[i] = new int[M]
}
```



Starting addr(assuming 64-bit machine/pointer stored in 8 bytes):
 100 108 100+(N-1)*8

```
for(int i=0;i<N;i++) {
    char* line = ReadLineFromFile(filename);
    int M = GetNumberOfIntegersPerLine(line);
    arr2D[i] = new int[M]
}
```

Starting addr:



2D Arrays (on heap)

Summary:

Creation: 2-steps

Initializing: 2-steps

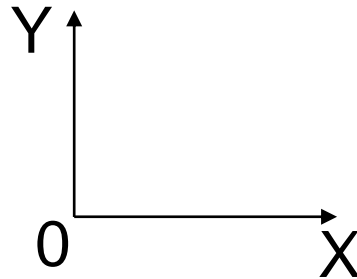
Releasing: 2-steps

```
for(int i=0;i<N;i++)  
    delete [] arr2D[i]; //frees memory at 1000, 5004,  
etc.  
delete [] arr2D; //frees memory at 100
```

2D Arrays (trivia)

- Notation used to refer to elements **different** from cartesian coordinates

- Cartesian:



(M,N) = move M along X axis,
N along Y axis

- 2D Arrays:

$\text{arr2D}[M][N]$ = move to $(M+1)^{\text{th}}$
row (along Y axis), to $(N+1)^{\text{th}}$
column (along X axis)!

$\text{arr2D}[0][0]$	accesses 1 st row, 1 st element
$\text{arr2D}[0][1]$	accesses 1 st row, 2 nd element
$\text{arr2D}[1][1]$	accesses 2 nd row, 2 nd element
$\text{arr2D}[N][M]$	accesses $N+1^{\text{th}}$ row, $M+1^{\text{th}}$ element

- From the previous bus trip data, what if we wanted to:

Day1 numbers: 10 23 45 44

Day2 numbers: 5 33 38 34 10 4

Day3 numbers: 9 17 10

.....

DayN numbers: 13 15 28 22 26 23 22 21

- Drop certain days as we analyzed arr2D?
- Add more days to (read from another file) to arr2D ?

i.e.

modify arr2D as program executes?