CS601: Software Development for Scientific Computing

Autumn 2022

Week3: Minimal C++ (contd..), Build tool (Make)

Suggested Reading

- Pointers and Pointer Arithmetic (slide 3 to slide 52)

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`

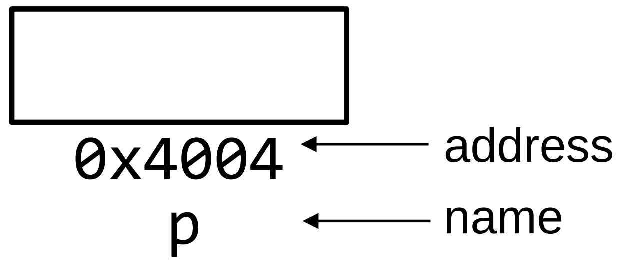
- A pointer always stores an address
- `<type>` of the pointer tells us what kind of data is stored at that address
- Example:
  - `int* p;`

    declares a pointer variable p holding an address, which identifies a memory location capable of storing an integer.

Initializing Pointers

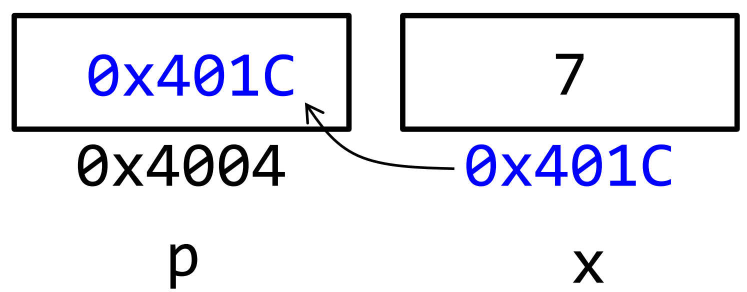
- `int* p;`

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`

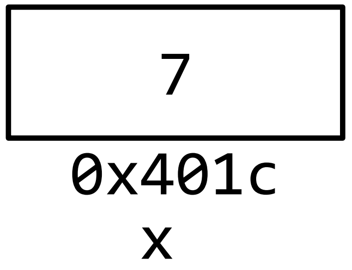
```
int* p=NULL; //p points to nowhere
```

- 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

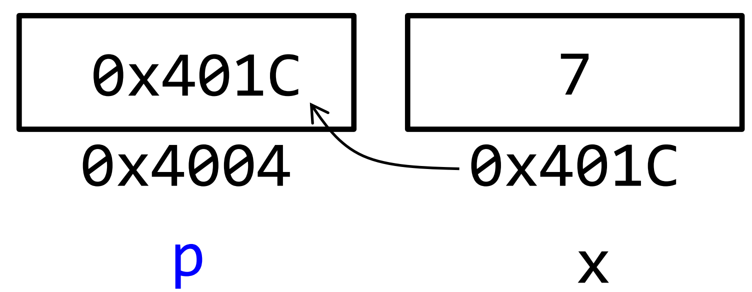
```
int x=7;
```



Using Pointers

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

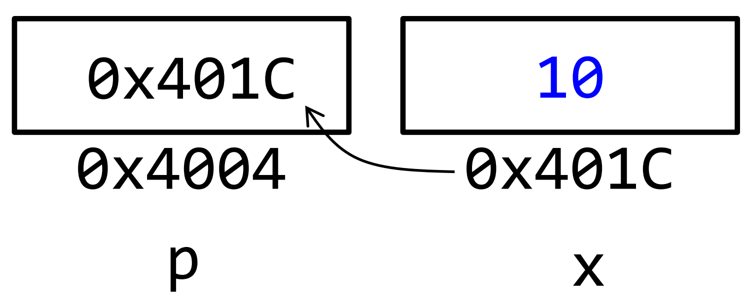
```
int x=7;
int* p = &x; //p now points to x
```



Using Pointers

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

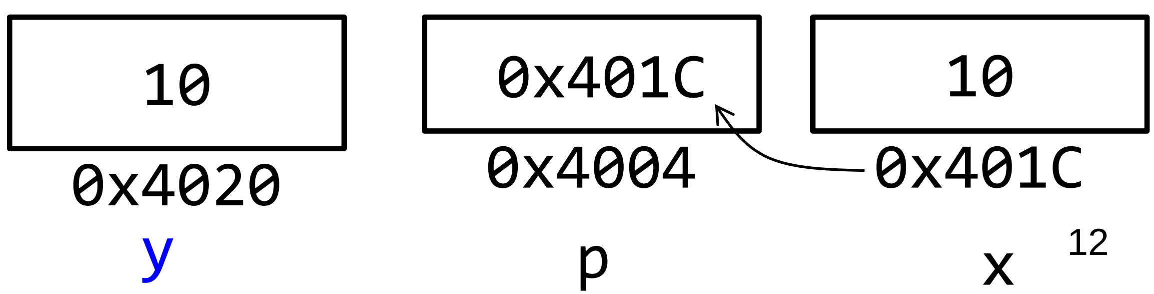
```
int x=7;
int* p = &x; //p now points to x
*p = 10; //this is the same as x=10
```



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
```

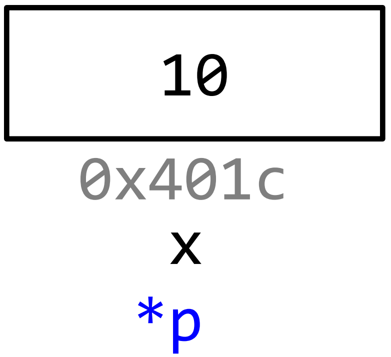


- Pointers as alternate names to memory locations

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

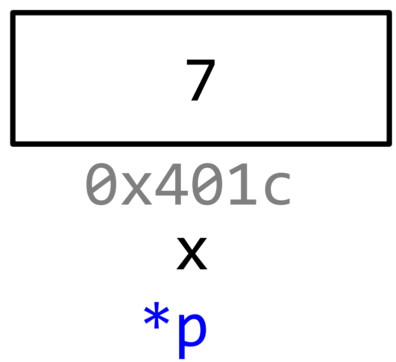
`x` is the name for an address

`*p` is the name for an address



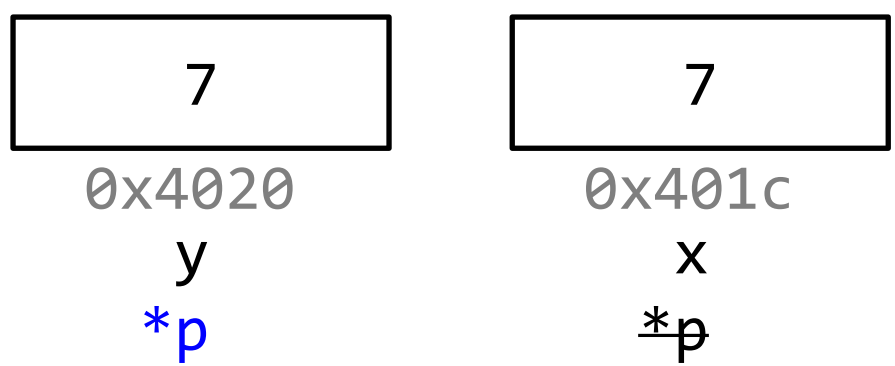
- Pointers as “dynamic” names to memory locations

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



- 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

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

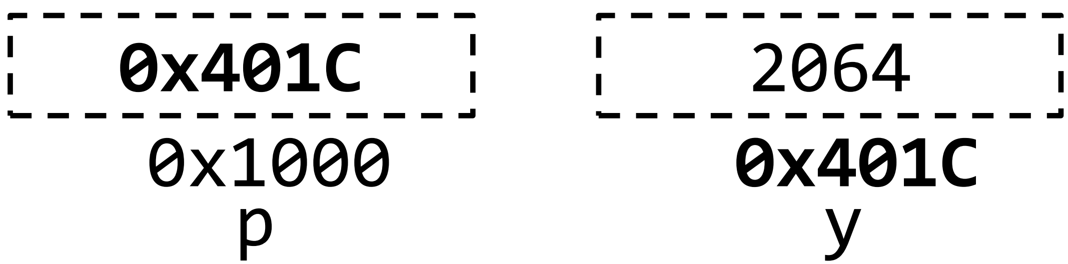
```
int y = 1040;
int* p= &y;
```

- 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

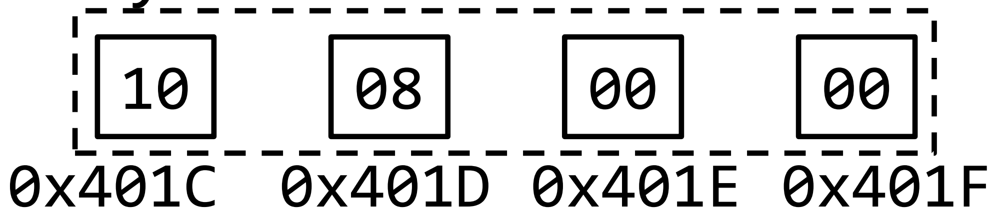
- Our representation of

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



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

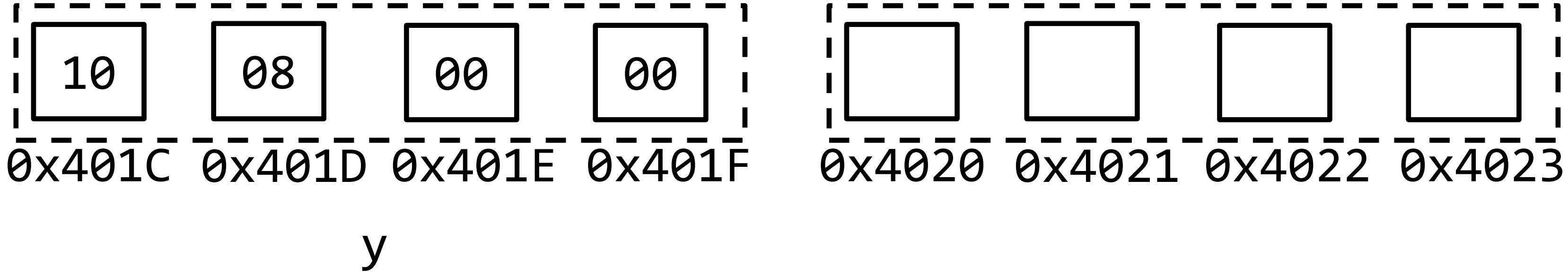
- (p+1) gets the "address of the next integer"

0x401C	2064
0x1000	**0x401C**
p	y

What is the address of the next integer?

Pointer Arithmetic

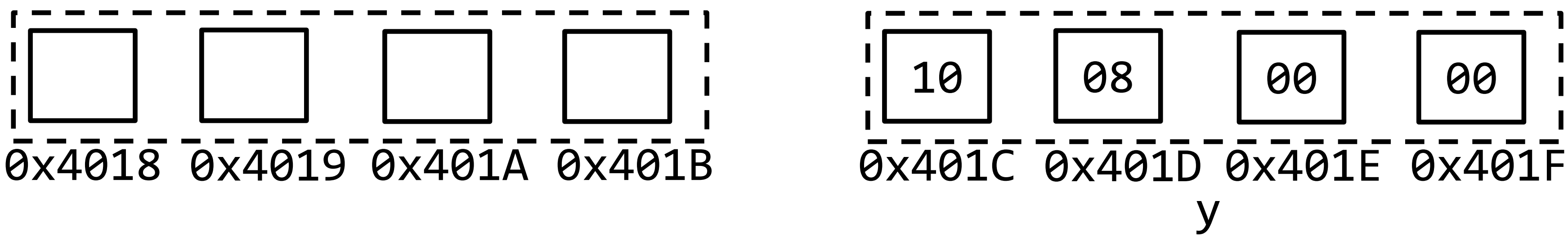
- What is the address of the next integer?
  - Add 4 to current value of p (0x401C) = `0x4020`



Pointer Arithmetic

- (p-1) computes the address before y

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



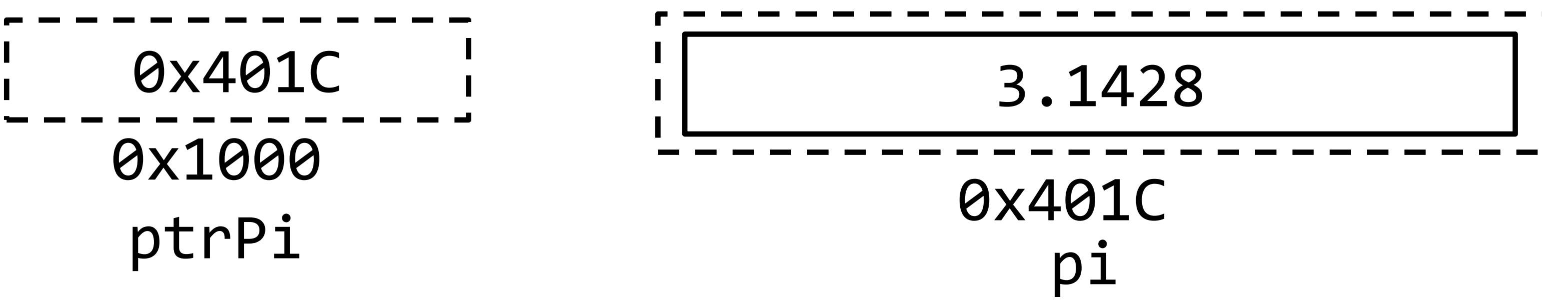
subtract 4 from the current value of `p (0x401C)` = `0x4018`

- 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;
```



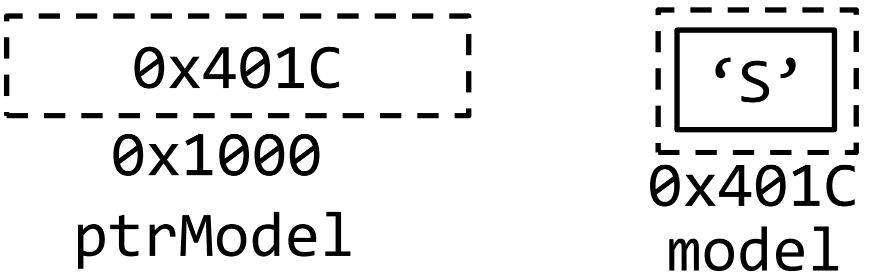
What is the address computed for `(ptrPi+1)`*?* `0x4024`

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

Pointer Arithmetic

- Pointer to `char`

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

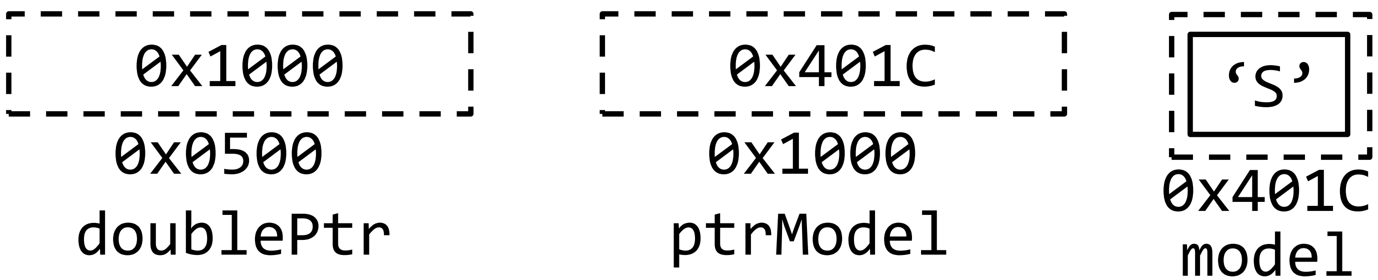


What is the address computed when we do `(ptrModel+1)`*?*

Pointer Arithmetic

- Pointer to pointer

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



Bonus: what is the address computed when we do `(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]`

  `...`

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`

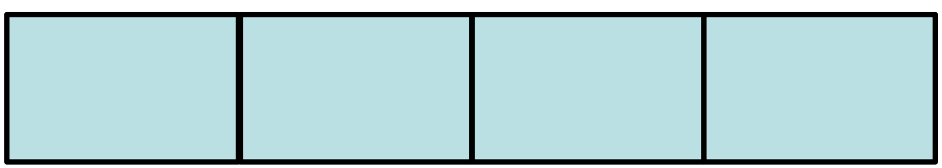
    “take back this memory so that it can be used for something else”

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:

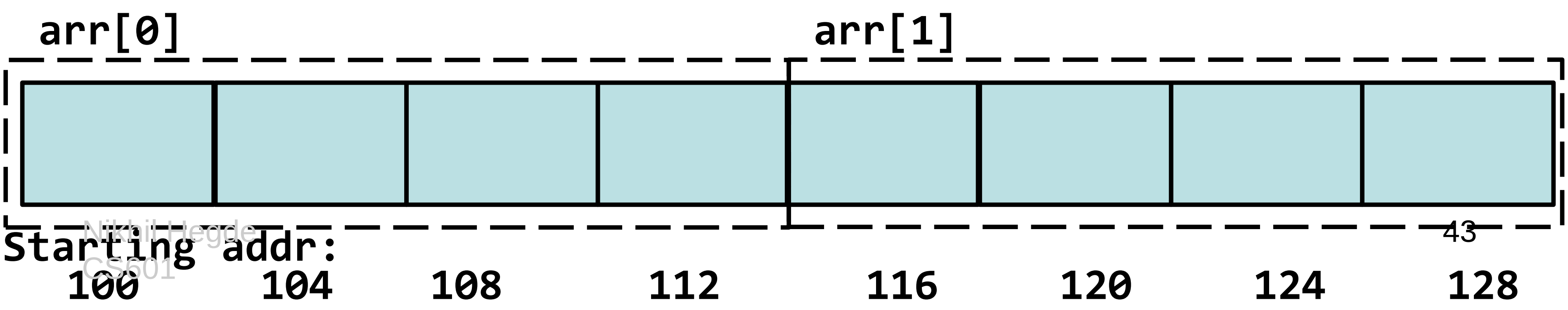
100 104 108 112

We have seen this

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 **`

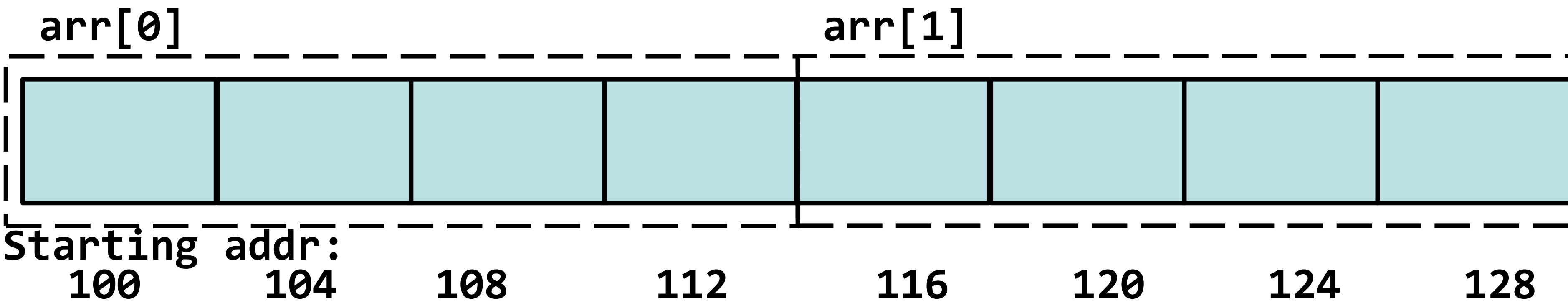
2D Arrays (on heap)

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 **)`

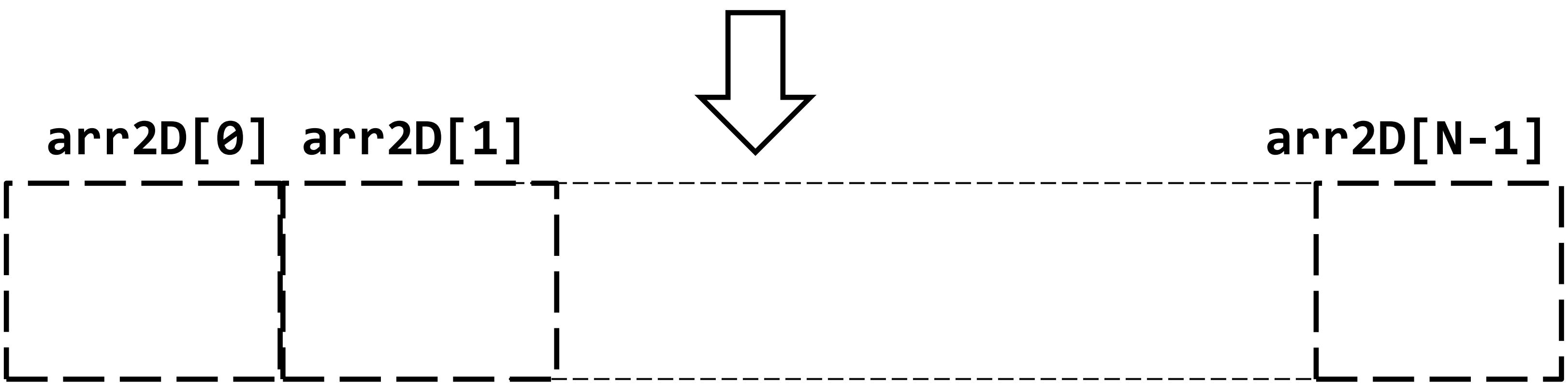
```
int N = GetNumberOfLinesFromFile(fileName);

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

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

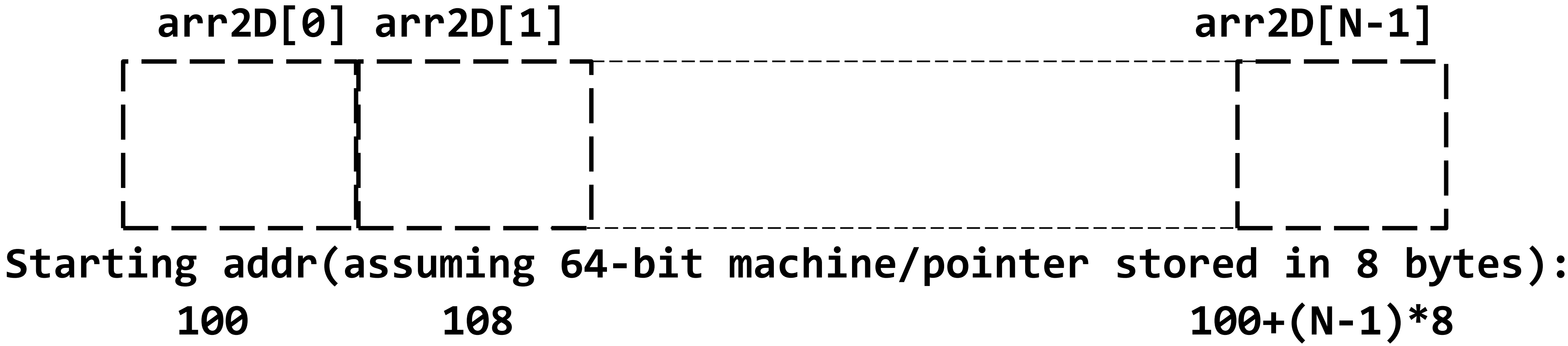



```
int N = GetNumberOfLinesFromFile(filename);
int** arr2D = new int*[N];
```



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]
}
```

arr2D[0] arr2D[1] arr2D[N-1]

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:

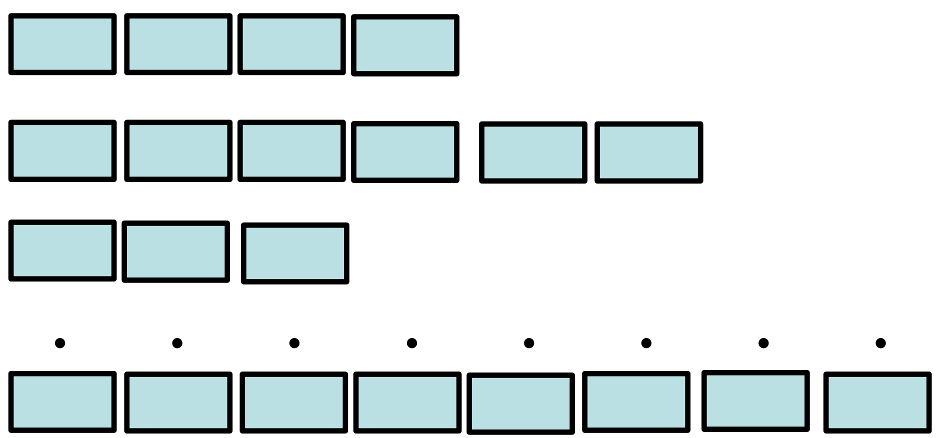
1000

5004

9000

. . . .

50



2D Arrays (on heap)

Summary:

Creation: 2-steps

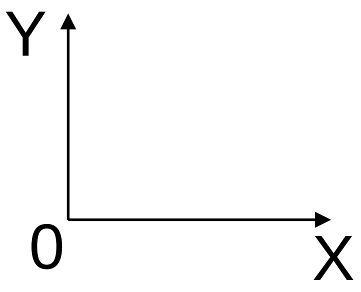
Initializing: 2-steps

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:

arr2D[M][N] = move to (M+1)th row (along Y axis), to (N+1)th column (along X axis)!

arr2D[0][0] accesses 1st row, 1st element
arr2D[0][1] accesses 1st row, 2nd element
arr2D[1][1] accesses 2nd row, 2nd element
arr2D[N][M] accesses N+1th row, M+1th 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?

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

Discussion

Refer to:

- `vectorprod_v1.cpp`
  - What if `atoi` doesn't provide accurate status about the value returned?
- `vectorprod_v2.cpp`
  - C++ stringstreams are an option. Is this code modular?
- `vectorprod_v3.cpp scprod.cpp`
  - What if there is already built-in function by the same name?
- `vectorprod_v4.cpp scprod_v4.cpp`
  - Namespaces

`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 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: "Dependencies or Prerequisite files" "Recipe"

```
testgen: testgen.cpp
        g++ testgen.cpp –o testgen
```

"target file name"

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