

CS601: Software Development for Scientific Computing

Autumn 2022

Week3: Minimal C++ (contd..), Version
Control Systems (Git), Build tool (Make)

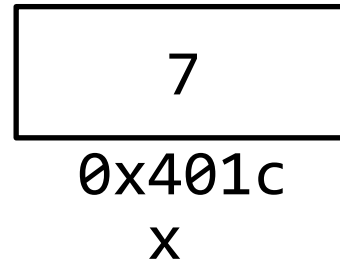
Suggested Reading

- Pointer and Pointer Arithmetic

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`

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

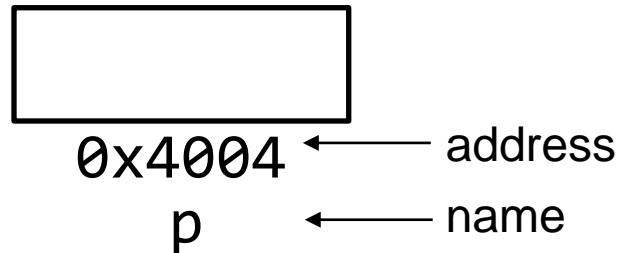
- A pointer always stores an address
- `<type>` of the pointer tells us what kind of data is stored at that address
- Example:
 - `int* p;`

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

Initializing Pointers

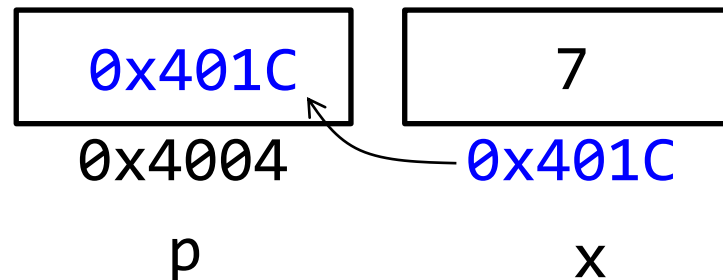
- `int* p;`

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

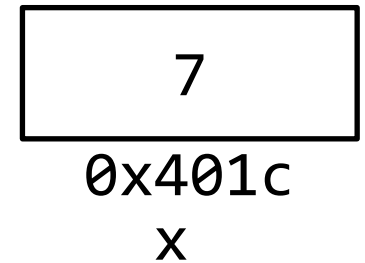
```
int* p=NULL; //p points to nowhere
```

- 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;
```

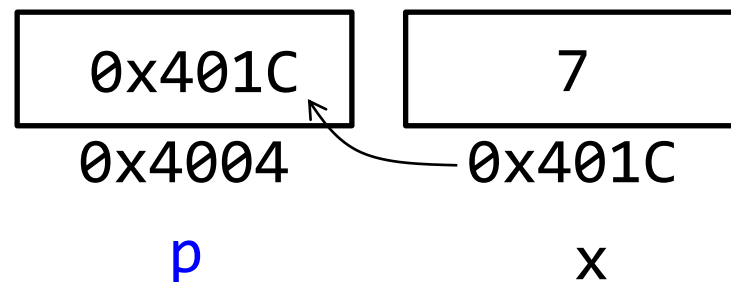


Using Pointers

- 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
```



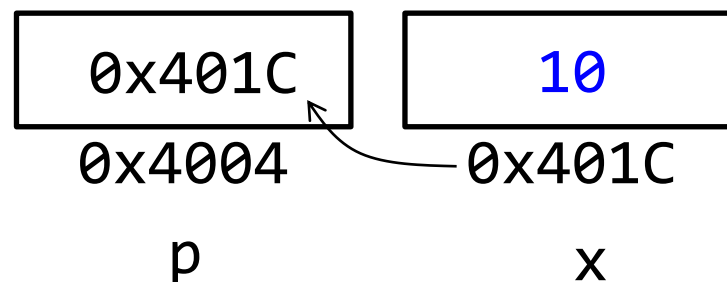
Using Pointers

- 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

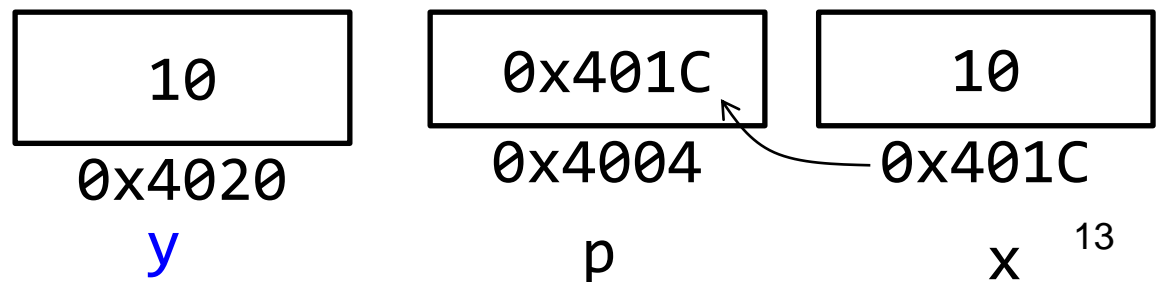
- The *dereference* operator (*)
 - Lets us access the memory location at the address stored in the pointer

```
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
```

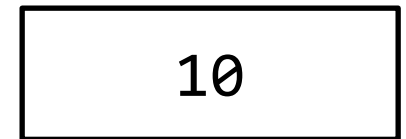


- Pointers as alternate names to memory locations

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

x is the name for an address

*p is the name for an address



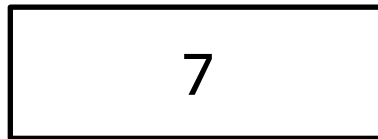
0x401c

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
```



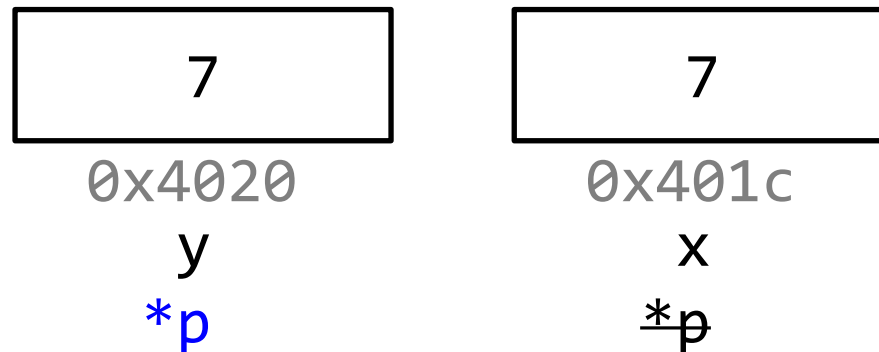
0x401c

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

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

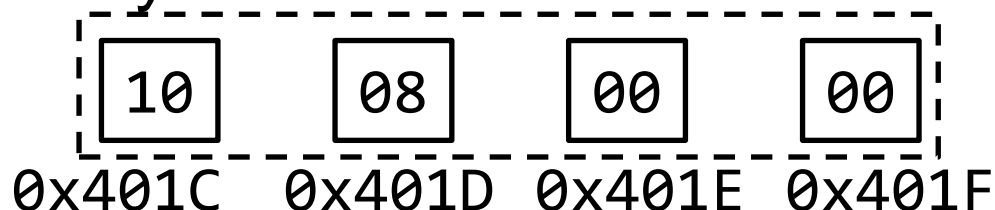
```
int y=2064;  
int* p = &y;
```

0x401C
0x1000
p

2064
0x401C
y

Pointer Arithmetic

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

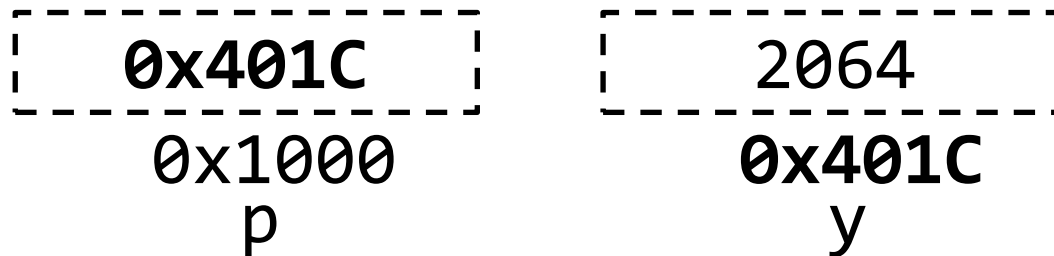


*2064 = 0x810 (=0x00,00,08,10 when written using 8 digits and x86 is little-endian)

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

Pointer Arithmetic

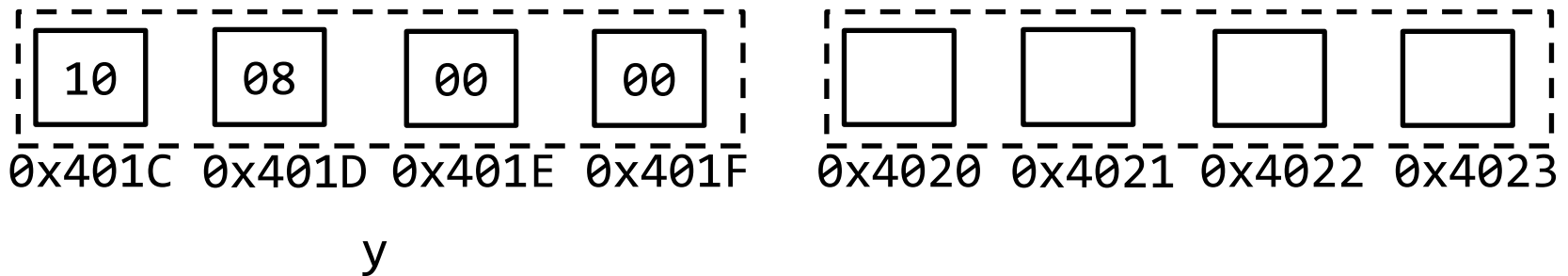
- $(p+1)$ gets the “address of the next integer”



What is the address of the next integer?

Pointer Arithmetic

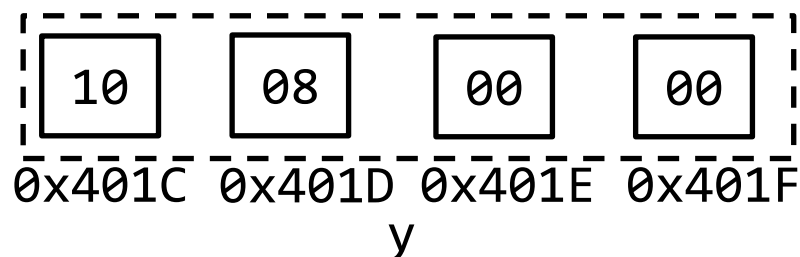
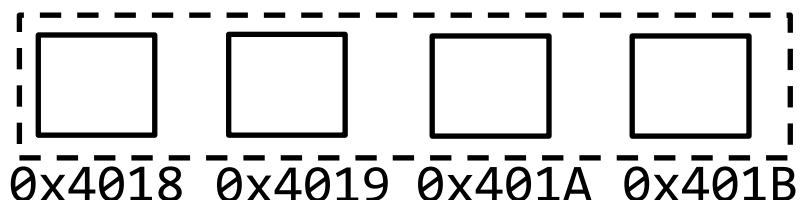
- What is the address of the next integer?
 - Add 4 to current value of p (0x401C) = 0x4020



Pointer Arithmetic

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

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



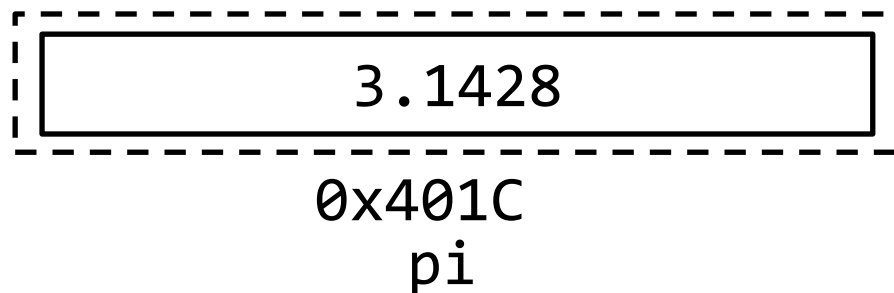
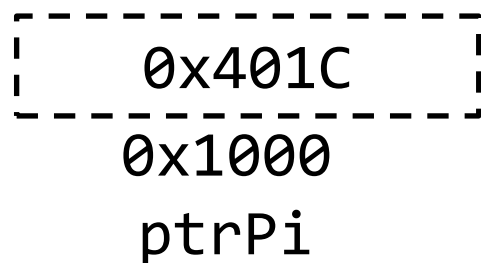
subtract 4 from the current value of p ($0x401C$) = $0x4018$

- 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;
```



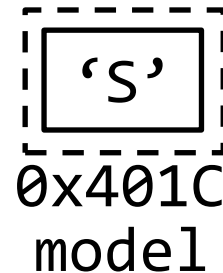
What is the address computed for $(ptrPi+1)$? `0x4024`

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

Pointer Arithmetic

- Pointer to char

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

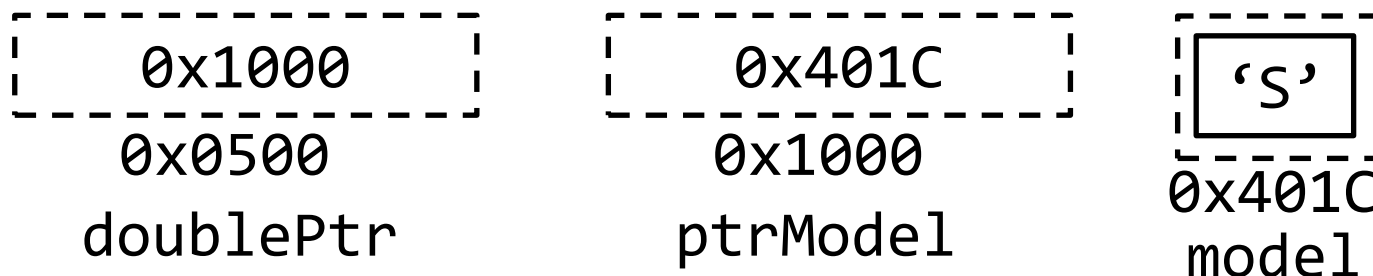


What is the address computed when we do `(ptrModel+1)`?

Pointer Arithmetic

- Pointer to pointer

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



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

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