iClicker Attendance

Please click on A if you are here:

A. I am here today.

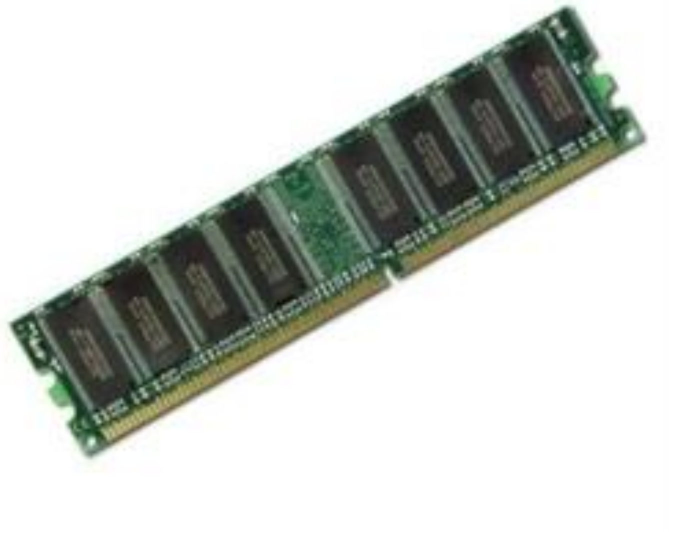
Pointers

Memory

- The act of keeping track of something over time
- “Remembering” is the concept of storing information
- A memory is no good unless you can retrieve that information
- In a computer, we remember information by writing bits (1/0) to memory
- We retrieve information by reading bits from memory

Memory is Different from Disk Storage

- All values in memory are forgotten when power is turned off
    - Memory is really "short-term" memory
- Reading and Writing memory is much faster than reading or writing disk
- Memory is organized differently than disk



Computer Memory Organization

- Computers read and write memory in 1 byte (8 bit) chunks
- Think of memory as a big C vector of chars:

```
char memory[2142240768];
```

- Like a vector, if we know the index of a byte of memory, we can either read or write to that byte:

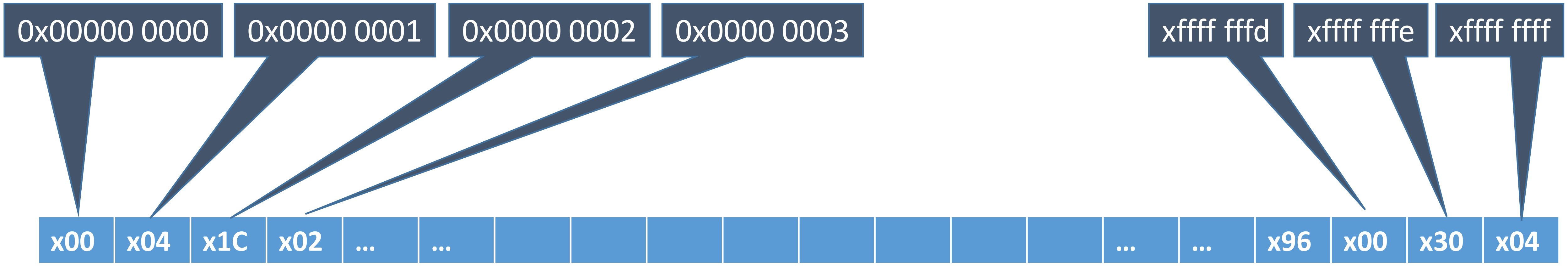
```
memory[1684501289] = ‘A’;
printf(“We stored %c\n”,memory[1684501289]);
```

What is a pointer?

- A 64 bit unsigned integer (a special kind of "unsigned long")
- Index into the “memory” vector
- Says “I’m not important… what’s important is over there…”
- Points AT or TO the real data in memory

Memory

- Vector of bytes
- Each byte has a value
- Each byte has an “index” or “address”
- Usually, the address is specified in hexadecimal



Cheap Memory

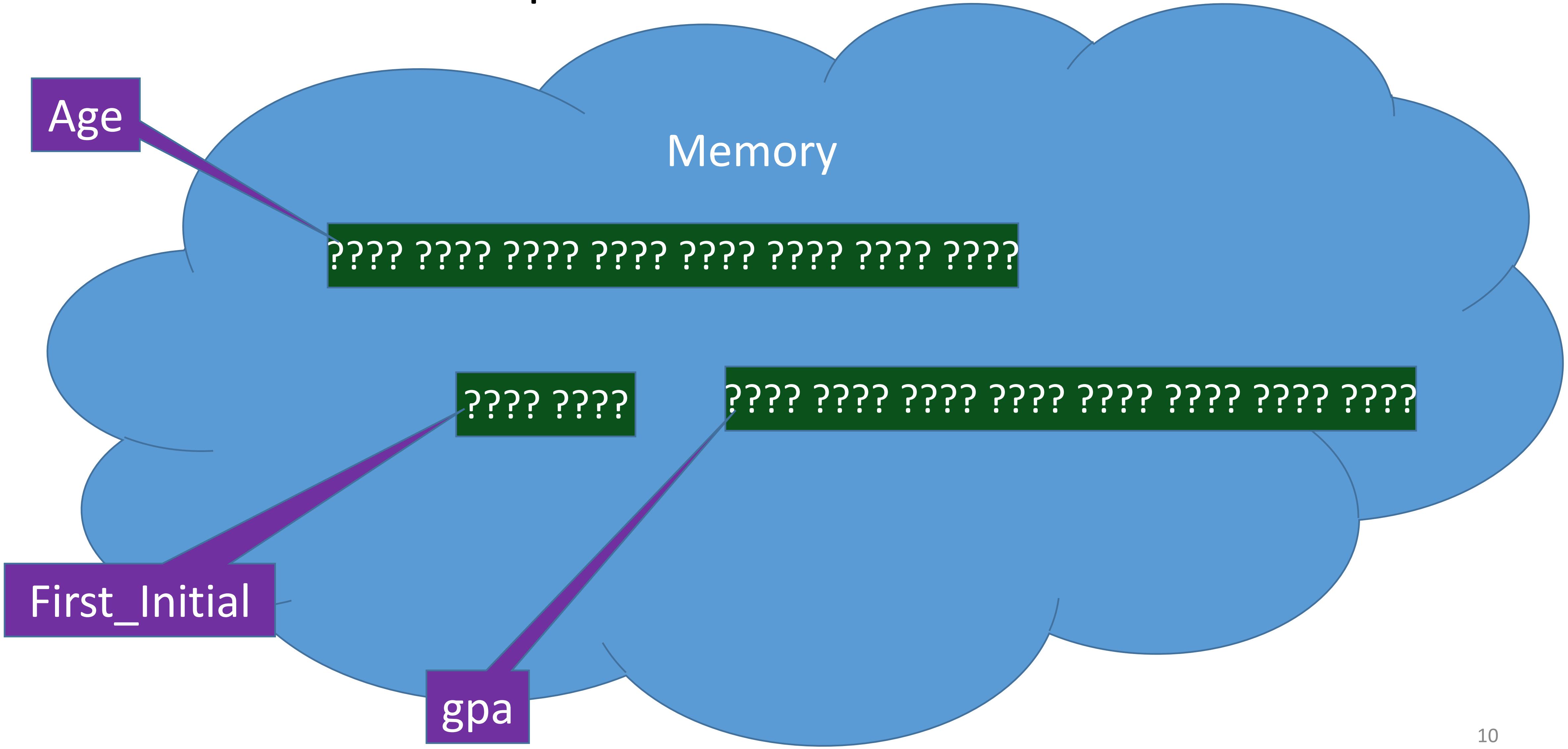
- Between Moore’s Law and brilliant OS parlor tricks, “Virtual Memory” is VERY cheap!
- Memory size depends on the size of the address

Address Size	Number of Bytes addressable
2 bytes (16 bits)	2^{16} = 64K = 65,376
4 bytes (32 bits)	2^{32} = 4G = 4,284,481,536
8 bytes (64 bits)	2^{64} = 16EiB >1.8 x 10^{19}

C Variables

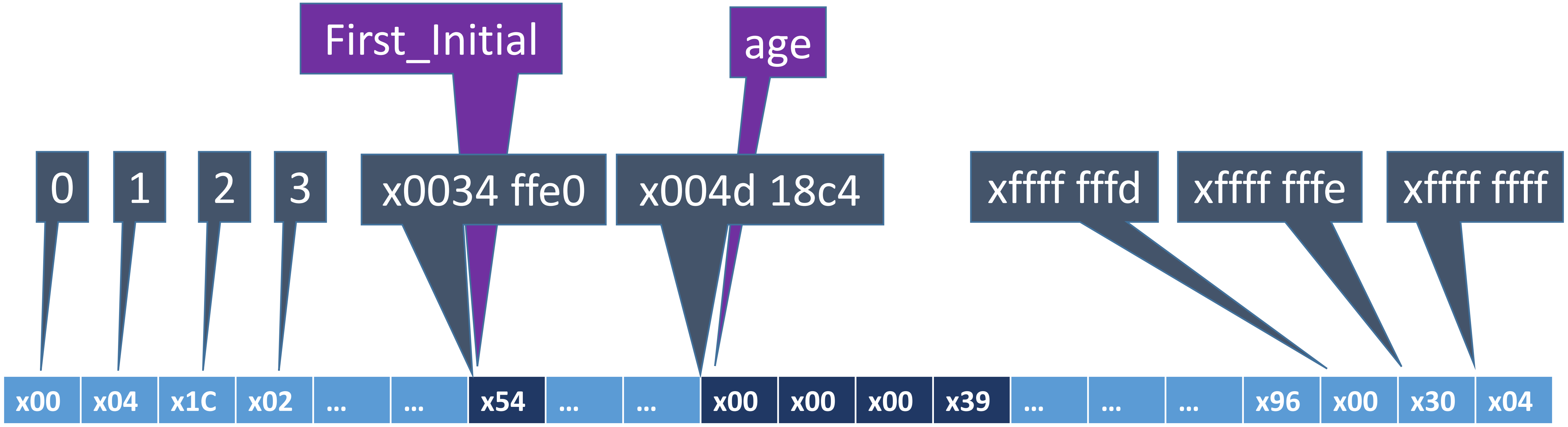
- A variable is a named piece of data
- Variables in C have…
  - A name (specified by the programmer)
  - A value (may be unassigned/unknown)
  - A location in memory (determined by the compiler)
  - A type (size and interpretation)
- Variables must be declared before they are used!

Variable Concept



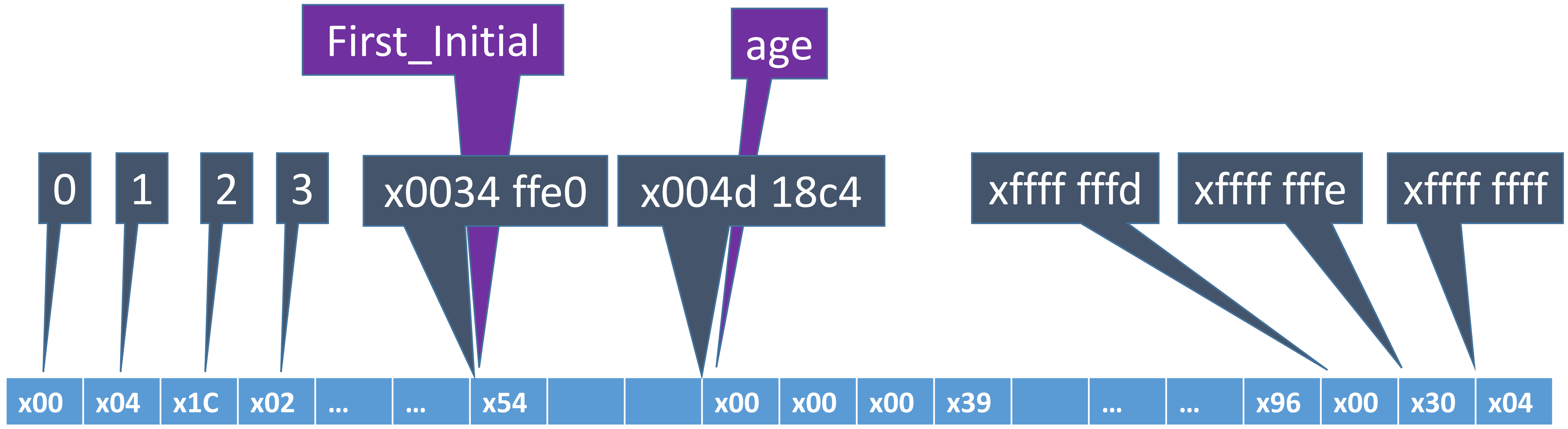
Variables In Memory

- Every variable starts at a specific location (address) in memory
- Type tells how many bytes in memory and how to interpret



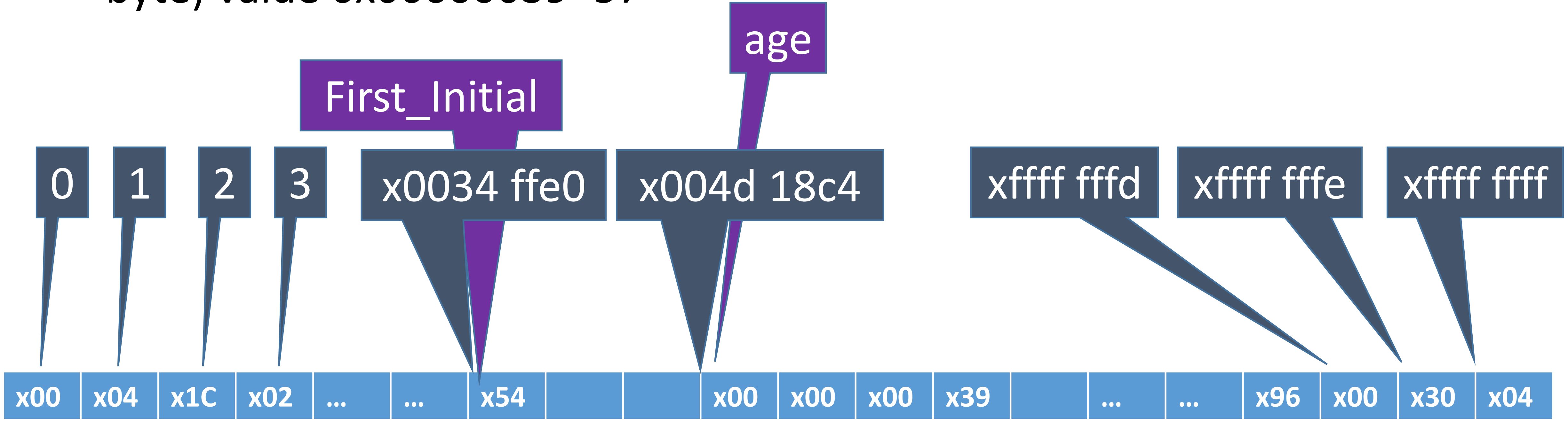
Variable Address/Location

- Where is the value for the variable in memory?
- The *address* of "First_Initial" is x0034ffe0, which *points to* 0x54 ='T'



Variable Address/Location

- Where is the value for the variable in memory?
- The *address* of “age” is x004d18c4, which *points to* the integer (4 byte) value 0x00000039=57

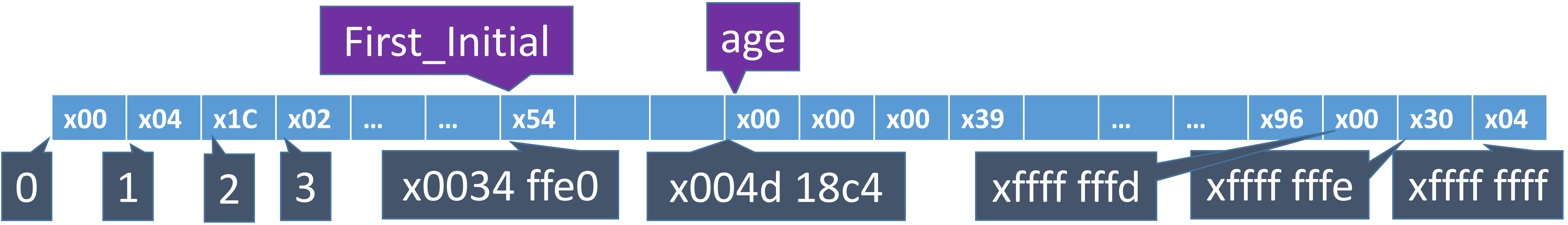


Address Of (&) operator

- An ampersand (&) in front of a variable indicates "address of"

```
char First_Initial='T';
int age=57;
printf("First_Initial is in memory at %p\n",&First_Initial);
```

```
First_Initial is in memory at 0x34ffe0
```



Pointers in C

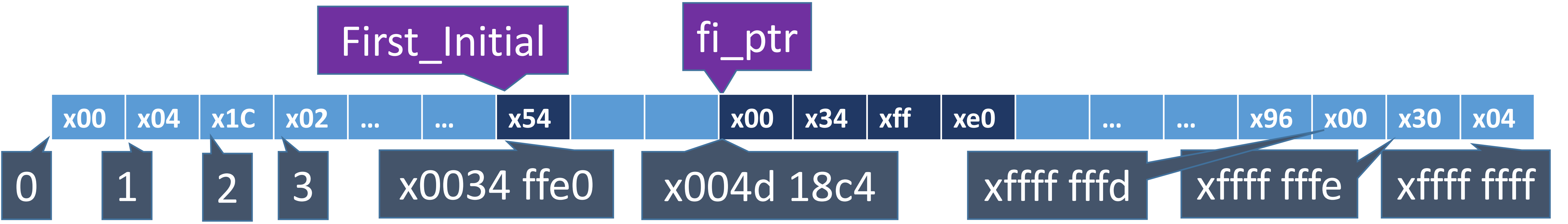
- Pointers are a special family of data types
  - A variable may be declared as a pointer
  - Like any other variable, space is reserved in memory for the value
- The *size* of a pointer is the size of an address (8 bytes, sometimes 4)
- The *value* of a pointer is an address – an index into memory
- The *type* of a pointer includes the type of value it is pointing to!
  - pointer to character
  - pointer to integer
  - pointer to float
  - pointer to an array of 14 characters
  - …

Declaring a Pointer

- Same as normal variable but need asterisk (*) : “pointer to”

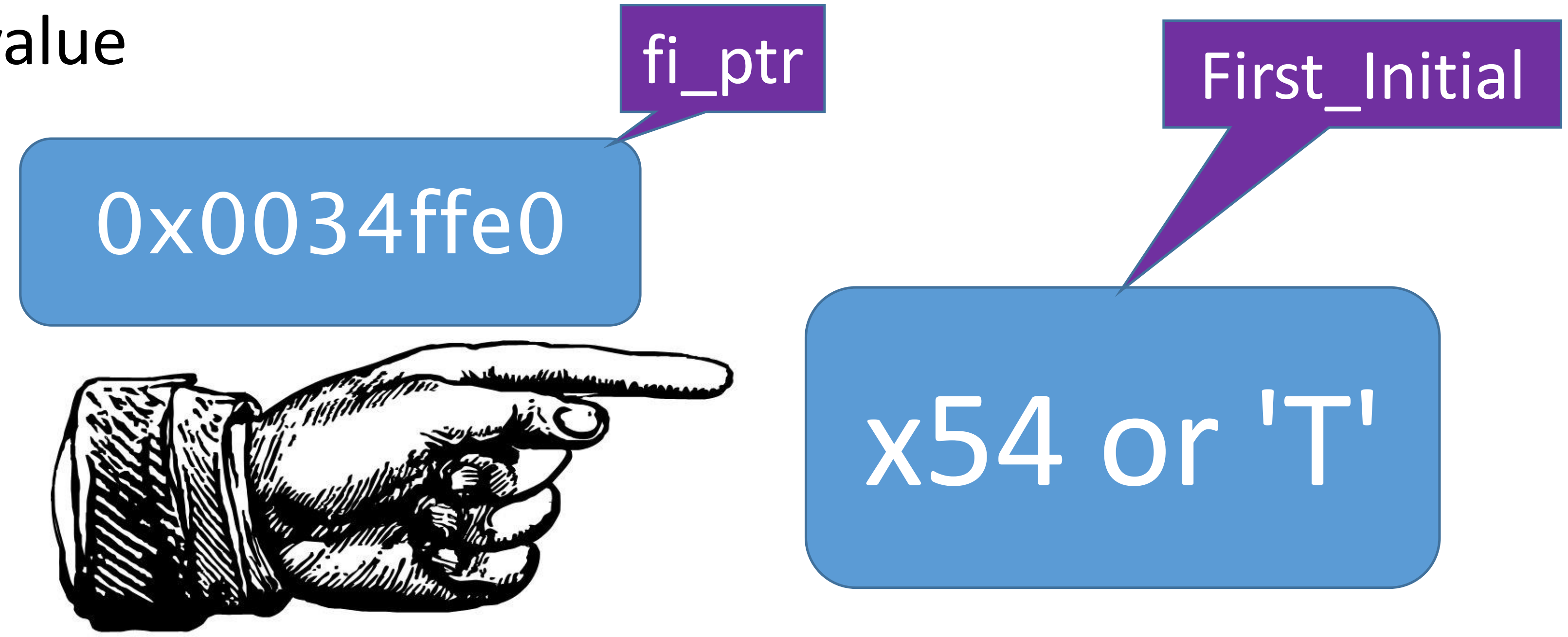
```
char First_Initial=‘T’; char * fi_ptr; // pointer to char
fi_ptr=&First_Initial;  // must be the address of char!
printf(“Value of fi_ptr at %p is %p\n”,&fi_ptr,fi_ptr);
```

```
Value of fi_ptr at 0x4d18c4 is 0x34ffe0
```



Pointers as References

- A pointer has a value… an address in memory
- A pointer *points to* another value… the data at that address
- Because we know what *type* the pointer is pointing to, we know how long the data at that address should be and how to interpret that value

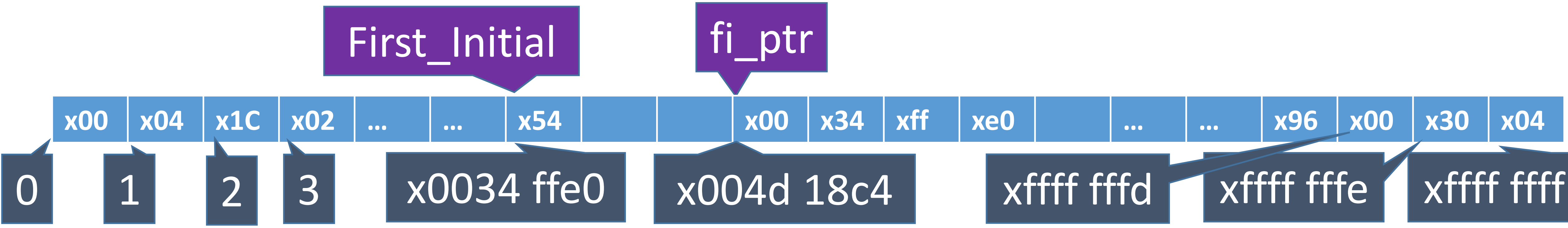


Getting the value at a Pointer

- Same as normal variable but need asterisk (*) : "value at"

```
char First_Initial='T';
char * fi_ptr=&First_Initial;   // pointer to char
printf("fi_ptr w/ value %p points at %c\n", fi_ptr,(*fi_ptr) );
fi_ptr w/ value 0x34ffe0 points at T
```

See also printMem.c

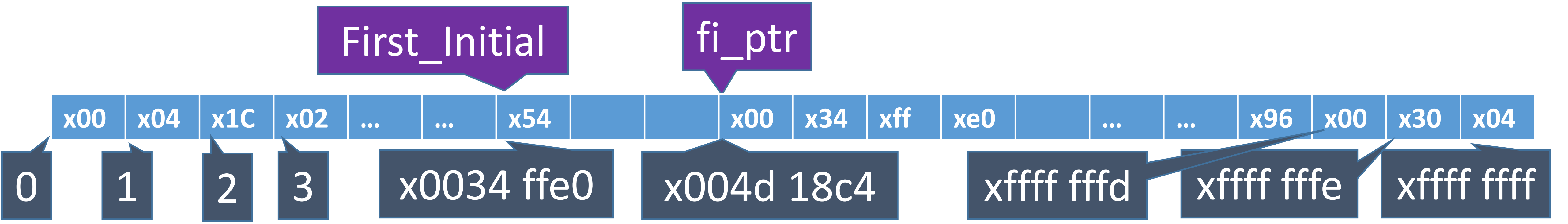


Terminology

fi_ptr *references* First_Initial

```
char First_Initial='T';
char * fi_ptr=&First_Initial;  // pointer to char
printf("fi_ptr points at %c\n", (*fi_ptr) );
fi_ptr points at T
```

*fi_ptr *dereferences* fi_ptr



Abuse of Symbols

Ampersand (&)

x & y // Bit-wise AND

x && y // Logical AND

&x // Address Of

Asterix (*)

x * y // multiplication

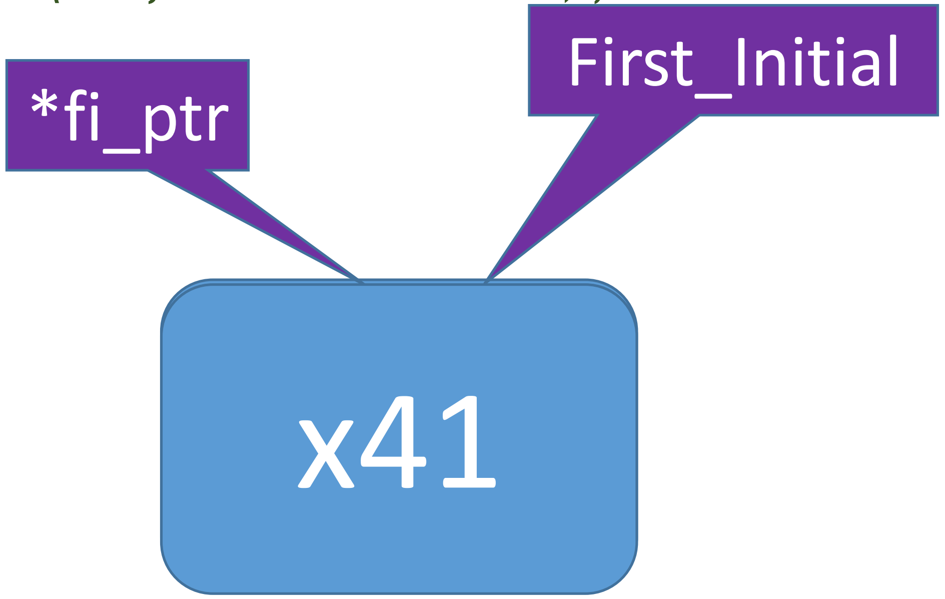
int * x // pointer to

(*x) // value at

Pointers as Aliases

```
char First_Initial=‘T’;
char * fi_ptr=&First_Initial;
(*fi_ptr)=‘A’; // Alias for First_Initial
printf(“First Initial: %c\n”,First_Initial);
```

```
First Initial: A
```



Dereferenced Assignments

- The left hand side of an assignment must be a location in memory that we can write to.
- Up to now, the left hand side of an assignment has always been a variable: `int x; x=17;`
  - The compiler knows where "x" is in memory, so this is legal
- When we added arrays, you could assign to an ELEMENT of an array: `float gpas[10]; gpa[7]=3.8;`
  - The compiler knows where "gpa[7]" is in memory, so this is legal
- A dereferenced pointer points to a location in memory
  - The compiler knows where `(* fi_ptr)` is in memory, so this is legal

Using NULL

- “NULL” is a special address whose value is 0x0000 0000 0000 0000.
- Beginning of Memory “belongs” to the operating system
  - General programs can read at 0, but cannot write at 0
- Therefore, we use NULL to indicate “pointer to nothing”
  - Or “pointer that we haven’t set yet”, or "invalid value for a pointer"

```
int *p=NULL; // p is a pointer to nothing (for now)
…
p=&age; // Now p is a pointer to an integer
```

C Gotcha: “Dereferencing a Null Pointer”

```
int *p=NULL; // p is a pointer to nothing (for now)
int x=foo();
if (x>0) { p=&x; }
(*p) = 5;
```

Segmentation Violation when x<=0

Pointers point to Types

- int *x; // x is a pointer to an integer
- &z – Type is: pointer to <type of z>
- (*myptr) – Type is: type which myptr is pointing to

  e.g. int *myptr=&area; (*myptr)='a';

assigning char to int
ASCII value of 'a' is 0x61
area is now 0x00000061
or 97

The Power of Pointers

- Pointers are a Reference to what they are pointing at
- Rather than passing an entire <type> element, we can pass a pointer to that type. (Pointers are 8 bytes long.)
  - Rather than passing an int, pass a pointer to an int
  - Rather than passing a struct, pass a pointer to that struct
  - Rather than passing an array, pass a pointer to that array
- If we pass a reference, then we can modify what we are pointing to, EVEN IF THE REFERENCE ITSELF IS A COPY!

iClicker Question

Have we seen & used to create a reference argument already? If so, in what context?

A. In a printf library call
B. In an if statement to connect two logic conditions
C. In a scanf library call
D. We've never used & in this class to create a reference

Example of Pass by Reference

```
int counter=0;
void incr(int x) {
        x = x + 1;
}
incr(counter);
printf(“counter=%d\n”,counter);
```

```
counter=0
```

```
int counter=0;
void incr(int *x) {
        (*x) = (*x) + 1;
}
incr(&counter);
printf(“counter=%d\n”,counter);
```

```
counter=1
```

Another example of pass by reference

```
int v0;
printf("Enter the initial velocity :> ");
scanf("%d",&v0);
```

- Argument expression &v0 is evaluated to the address of v0
- If scanf parameter is "int * intPointer", the VALUE of intPointer is the address of v0
- scanf reads a number from the terminal, and writes it to (*intPointer), which is an alias for v0

Resources

- Programming in C, Chapter 10
- Wikepedia Pointers : https://en.wikipedia.org/wiki/Pointer_(computer_programming)
- C Pointer Tutorial : http://www.tutorialspoint.com/cprogramming/c_pointers.htm