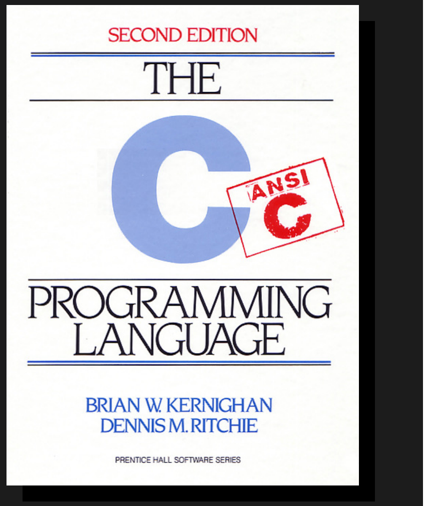
More Number Representation, C Programming Language Overview



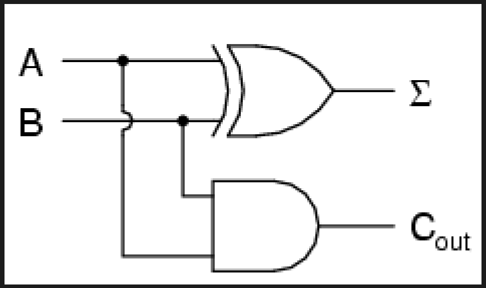
Math 230

Assembly Language Programming (Computer Organization)

Lecture 3

Operations with Numbers

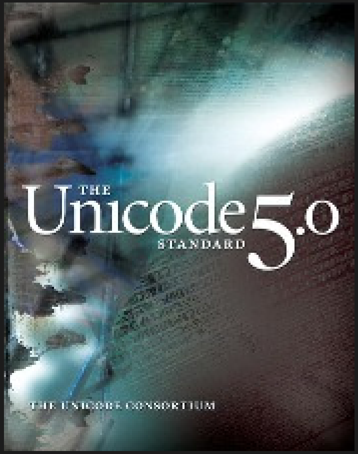
- What things can we do with numbers?
  - Add
  - Subtract
  - Multiply
  - Divide
  - Compare
- Addition
  - can easily build a circuit to do it!



- Subtraction
  - can we use the circuitry for addition?
- Comparison
  - How can you tell if one number is larger than another?

Bits Can Represent Anything

- Colors, music, text,video, texture, smell
  - We can quantize anything
- Characters
  - 26 letters => 5 bits
  - upper + lowercase+ punctuations => 7 bits (ASCII)
  - to cover all world's various alphabets
    - => 8, 16, 32 bits ("Unicode")
- Logical Values
  - 0 = false
  - 1 = true
- Colors
  - using red, blue, and yellow we can represent a wide range of colors using paint
- Locations/Addresses/Commands
  - Memorize: N bits ← → 2^N things



Negative Numbers

- So far, only positive numbers?
- Consider:
  - Borrow most significant bit and call it a sign bit

How to Represent Negative Numbers?

- So far, unsigned numbers
- Obvious solution: define leftmost bit to be sign!
  - 0 ⇒ +, 1 ⇒ –
  - Rest of bits can be numerical value of number
- Representation called sign and magnitude
- MIPS uses 32-bit integers. $+1_{ten}$ would be:

  0000 0000 0000 0000 0000 0000 0000 0001

- And $–1_{ten}$ in sign and magnitude would be:

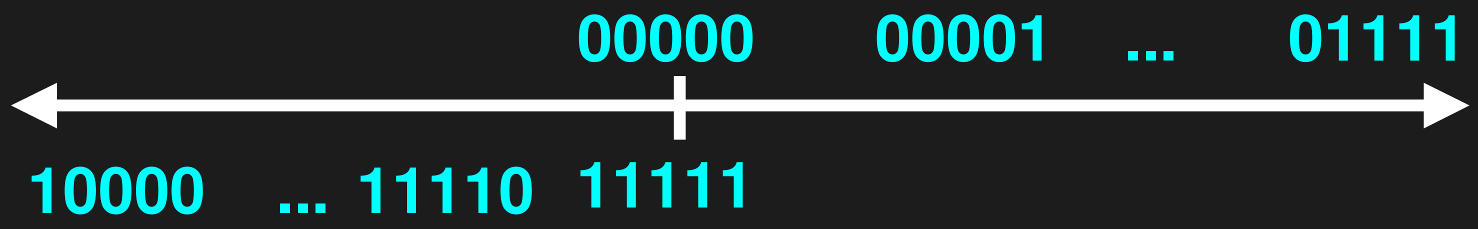
  1000 0000 0000 0000 0000 0000 0000 0001

Shortcomings of sign and magnitude?

- Arithmetic circuit complicated
  - Special steps depending whether signs are the same or not
- Also, two zeros
  - 0x00000000 = $+0_{ten}$
  - 0x80000000 = $–0_{ten}$
  - What would two 0s mean for programming?
- Therefore sign and magnitude abandoned

Another try: complement the bits

- Example: $7_{10} = 00111_2$ $-7_{10} = 11000_2$
- Called One's Complement
- Note: positive numbers have leading 0s, negative numbers have leadings 1s.



00000 00001 ... 01111

10000 ... 11110 11111

- What is -00000 ? Answer: 11111
- How many positive numbers in N bits?
- How many negative numbers?

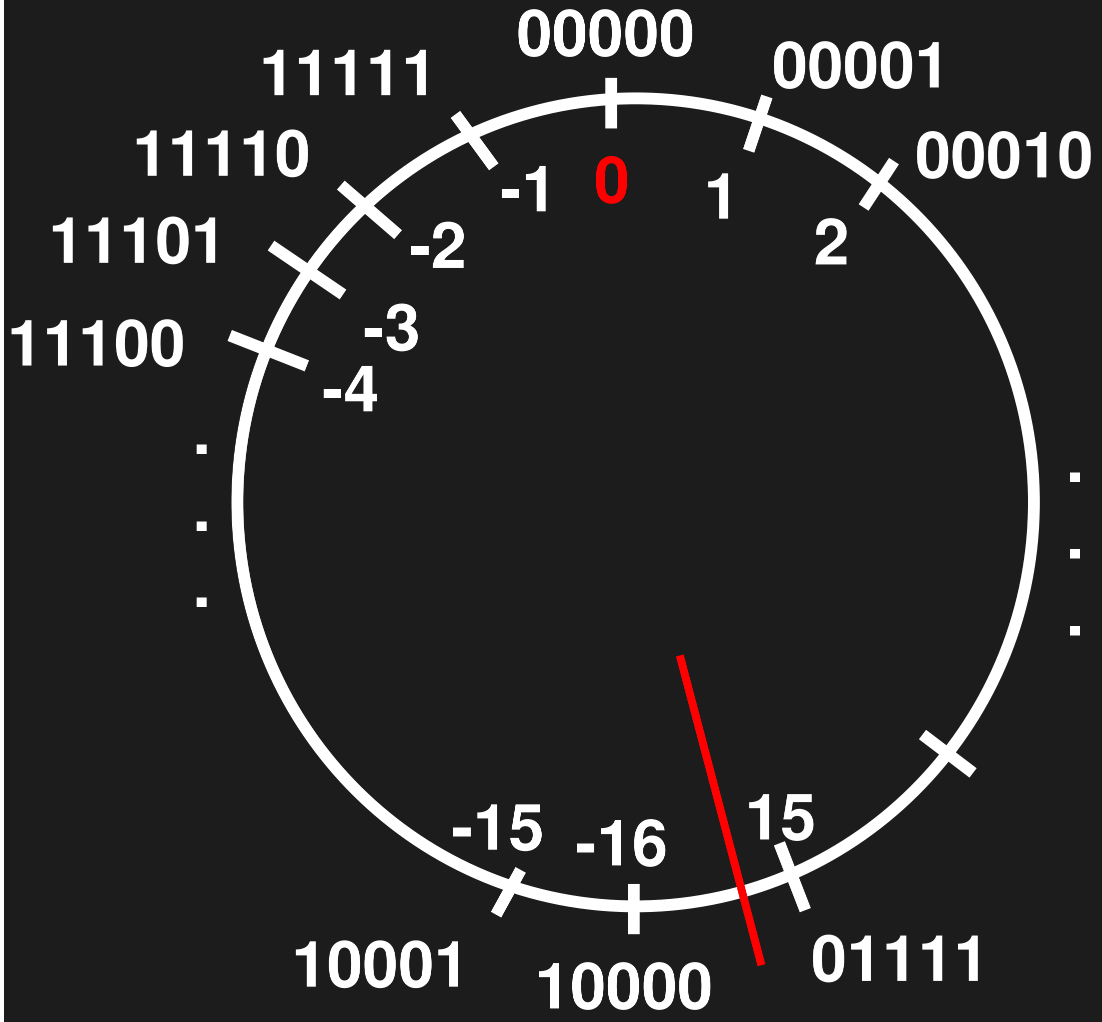
Shortcomings of One’s complement?

- Arithmetic still somewhat complicated.
- Still two zeros
  - `0x00000000` = $+0_{\text{ten}}$
  - `0xFFFFFFFF` = -0_{ten}
- Although used for awhile on some computer products, one’s complement was eventually abandoned because another solution was better.

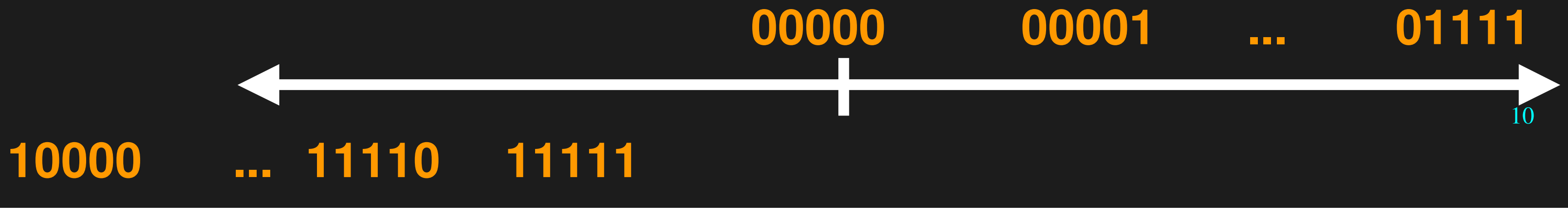
Standard Negative Number Representation

- What is result for unsigned numbers if tried to subtract large number from a small one?
  - Would try to borrow from string of leading 0s, so result would have a string of leading 1s
    - 3 - 4 ⇒ 00…0011 – 00…0100 = 11…1111
  - With no obvious better alternative, pick representation that made the hardware simple
  - As with sign and magnitude, leading 0s ⇒ positive, leading 1s ⇒ negative
    - 000000...xxx is ≥ 0, 111111...xxx is < 0
    - except 1…1111 is -1, not -0 (as in sign & mag.)
- This representation is Two's Complement

2’s Complement Number “line”: N = 5



- 2^{N-1} non-negatives
- 2^{N-1} negatives
- one zero
- how many positives?



Two’s Complement for N=32

$0000 \ldots 0000\ 0000\ 0000\ 0000_{\text{two}} =$	0_{ten}
$0000 \ldots 0000\ 0000\ 0000\ 0001_{\text{two}} =$	1_{ten}
$0000 \ldots 0000\ 0000\ 0000\ 0010_{\text{two}} =$	2_{ten}
. . .	
$0111 \ldots 1111\ 1111\ 1111\ 1101_{\text{two}} =$	$2{,}147{,}483{,}645_{\text{ten}}$
$0111 \ldots 1111\ 1111\ 1111\ 1110_{\text{two}} =$	$2{,}147{,}483{,}646_{\text{ten}}$
$0111 \ldots 1111\ 1111\ 1111\ 1111_{\text{two}} =$	$2{,}147{,}483{,}647_{\text{ten}}$
$1000 \ldots 0000\ 0000\ 0000\ 0000_{\text{two}} =$	$-2{,}147{,}483{,}648_{\text{ten}}$
$1000 \ldots 0000\ 0000\ 0000\ 0001_{\text{two}} =$	$-2{,}147{,}483{,}647_{\text{ten}}$
$1000 \ldots 0000\ 0000\ 0000\ 0010_{\text{two}} =$	$-2{,}147{,}483{,}646_{\text{ten}}$
. . .	
$1111 \ldots 1111\ 1111\ 1111\ 1101_{\text{two}} =$	-3_{ten}
$1111 \ldots 1111\ 1111\ 1111\ 1110_{\text{two}} =$	-2_{ten}
$1111 \ldots 1111\ 1111\ 1111\ 1111_{\text{two}} =$	-1_{ten}

- One zero; 1st bit called sign bit
- 1 “extra” negative:no positive $2{,}147{,}483{,}648_{\text{ten}}$

Two’s Complement Formula

- Can represent positive and negative numbers in terms of the bit value times a power of 2:

  d_{31} x $-(2^{31})$ + d_{30} x 2^{30} + ... + d_2 x 2^2 + d_1 x 2^1 + d_0 x 2^0

- Example: 1101_{two}

  $= 1\text{x}-(2^3) + 1\text{x}2^2 + 0\text{x}2^1 + 1\text{x}2^0$

  $= -2^3 + 2^2 + 0 + 2^0$

  $= -8 + 4 + 0 + 1$

  $= -8 + 5$

  $= -3_{ten}$

Two's Complement Formula

- Can represent positive and negative numbers in terms of the bit value times a power of 2:

$$d_{31} \text{ x } -(2^{31}) + d_{30} \text{ x } 2^{30} + \ldots + d_2 \text{ x } 2^2 + d_1 \text{ x } 2^1 + d_0 \text{ x } 2^0$$

- Example: 1101_{two}

$= 1\text{x}-(2^3) + 1\text{x}2^2 + 0\text{x}2^1 + 1\text{x}2^0$

$= -2^3 + 2^2 + 0 + 2^0$

$= -8 + 4 + 0 + 1$

$= -8 + 5$

$= -3_{\text{ten}}$

Two’s Complement shortcut: Negation

Two's Complement shortcut: Negation

- Change every 0 to 1 and 1 to 0 (invert or complement), then add 1 to the result

Two’s comp. shortcut: Sign extension

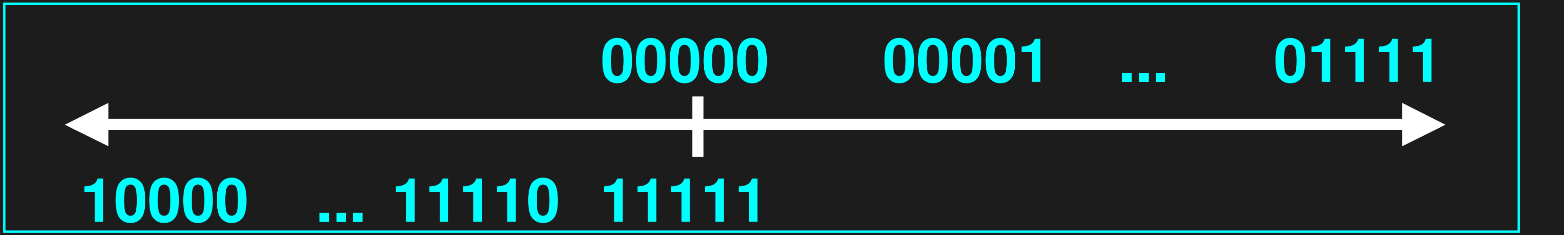
- Convert 2’s complement number rep. using n bits to more than n bits
- Simply replicate the most significant bit (sign bit) of smaller to fill new bits
  - 2’s comp. positive number has infinite 0s
  - 2’s comp. negative number has infinite 1s
  - Binary representation hides leading bits; sign extension restores some of them
  - 16-bit -4_{ten} to 32-bit:

$$1111\ 1111\ 1111\ 1100_{two}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$$

Number summary...

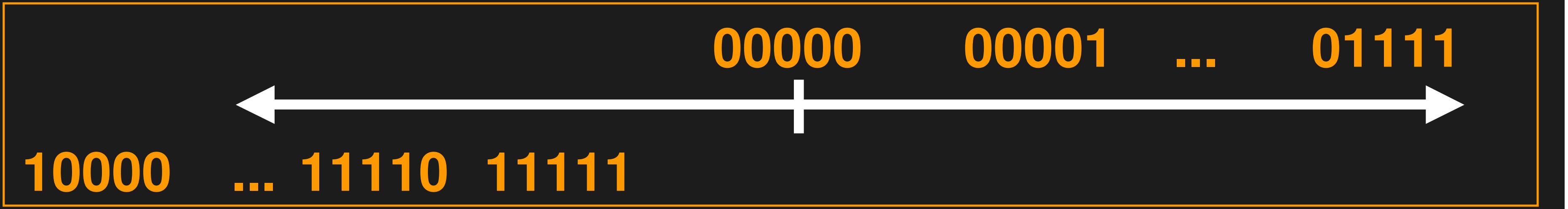
- We represent "things" in computers as particular bit patterns: N bits ⇒ 2^N
- Decimal for human calculations, binary for computers, hex to write binary more easily
- 1's complement - mostly abandoned



00000 00001 ... 01111

10000 ... 11110 11111

- 2's complement universal in computing: cannot avoid, so learn



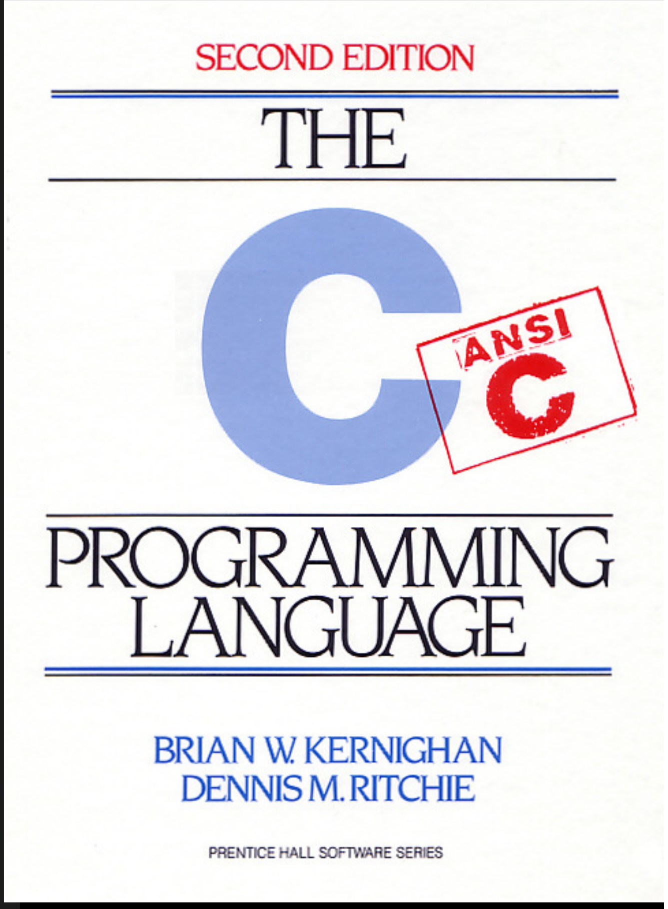
00000 00001 ... 01111

10000 ... 11110 11111

- Overflow: numbers ∞; computers finite,errors!

C Overview

- Systems software
  - Unix, Linux, MacOSX
- Small Language
- Standardized C Preprocessor for
  - macro def
  - source code file inclusion
  - conditional compilation
- gcc can cross compile to MIPS et al



ANSI C

- K&R was the informal “spec” for many years
  - K&R C
  - 2nd edition covers ANSI C std
- ANSI: Superset of K&R *C*
- After K&R published
  - **`void`** functions
  - functions returning **`struct`** or **`union`**
  - assignment for **`struct`**
  - **`const`** qualifier

```
int main(argc, argv)
 int argc;
 char* argv[];
 { . . .  }
```

```
int main(int argc, char* argv[])
{ . . .}
```

C99

- inline functions
  - eliminate function call; expand inline
- variables declared anywhere
- New data types
  - long long int
  - boolean data type
  - complex types for complex numbers
- Header files added, notably
  - `<stdbool.h>`
  - `<tgmath.h>` type generic math functions
- gcc not fully compliant (http://gcc.gnu.org/c99status.html)

Standard Headers

<assert.h>

<complex.h>

<ctype.h>

<errno.h>

<fenv.h>

<float.h>

<inttypes.h>

<iso646.h>

<limits.h>

<locale.h>

<math.h>

<setjmp.h>

<signal.h>

<stdarg.h>

<stdbool.h>

<stddef.h>

<stdint.h>

<stdio.h>

<stdlib.h>

<string.h>

<tgmath.h>

<time.h>

<wchar.h>

<wctype.h>

C vs Java

C	*Java*
Relatively Fast	*Relatively Slow*
Procedural	*OOP*
Platform Dependent	*Platform Independent*
Arrays initialize to garbage	*Arrays initialize to zero*
Small libraries	*Huge libraries*
Small executable	*Larger runnable files*
x.c => x.exe or *x.out*	*x.java => x.class*
Preprocessor	*No preprocessor*
No memory management	*Garbage Collections*
Pointers	*No global variables*
	Variable Declarations

C Syntax: variable declaration

- Similar to Java
- ANSI requires declarations to go at start of block
    - Java allows anywhere

C Syntax

- Java has booleans to represent true/false

- C has nonzeroes which evaluate to “true”
  - false
    - if(0)… //always false
    - if (NULL) …//always false
    - if(‘0’) …//always true

C Syntax: Flow Control

- Decision structures of C almost identical to Java.
  - if-else
  - switch
  - while and for
  - do- while

Memory: Conceptual

- Think of memory as one HUGE array
    - we will talk about how a “2D” array maps onto one linear space
- Each cell can be “*addresed*”
    - Address signed or unsigned?
- Commonly work with reference-type variables, i.e., pointers

Pointers

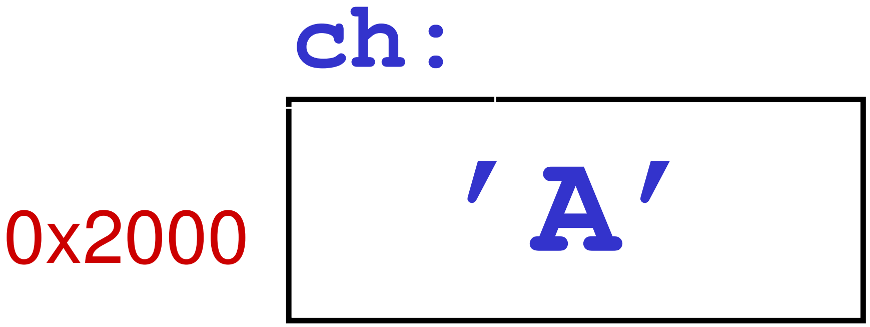
Memory Address of a Variable

Memory Address of a Variable

```
char ch = ’A’;
```

Memory Address of a Variable

```
char ch = 'A';
```



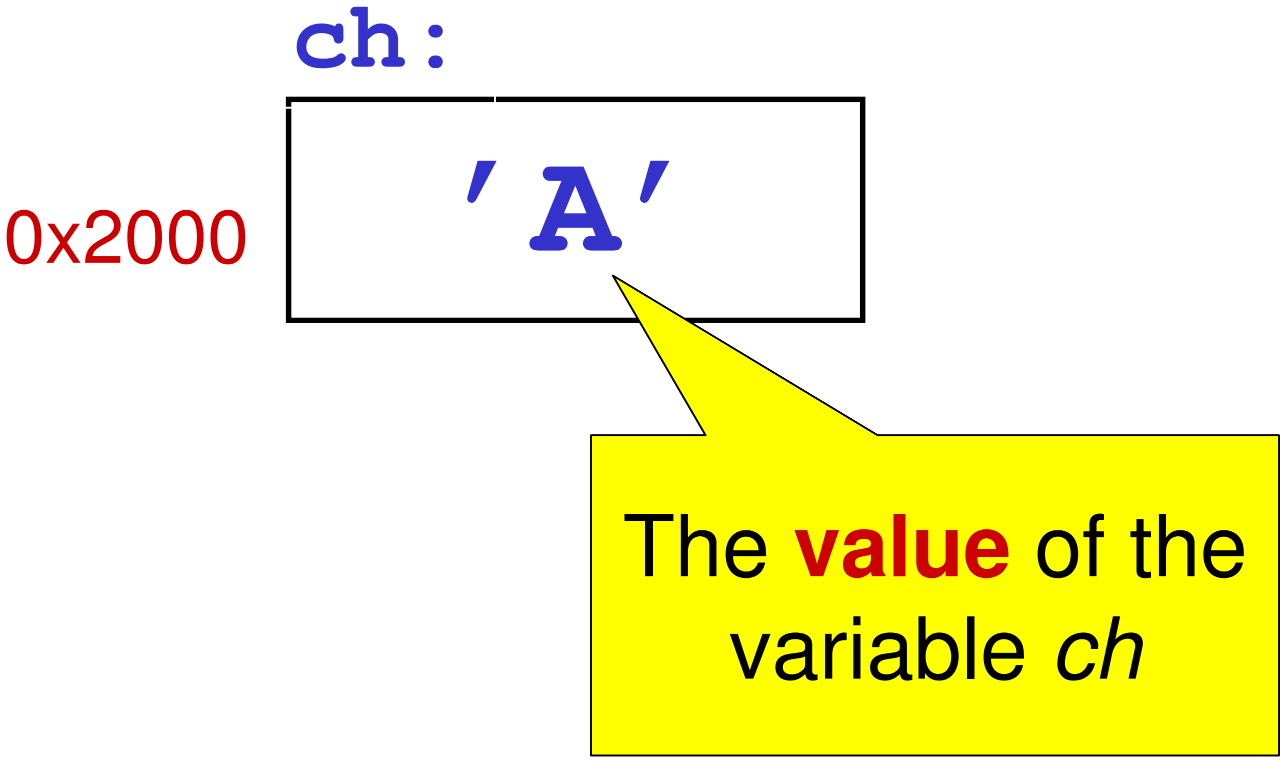
ch:

'A'

0x2000

Memory Address of a Variable

```
char ch = 'A';
```



ch:

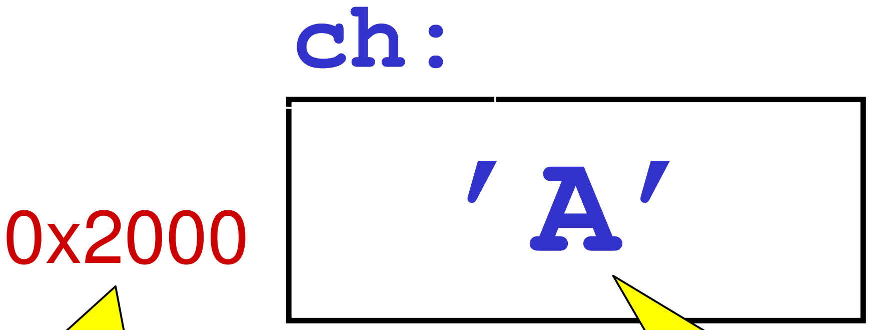
0x2000

'A'

The **value** of the variable *ch*

Memory Address of a Variable

```
char ch = ’A’;
```



The **memory address** of the variable *ch*

The **value** of the variable *ch*

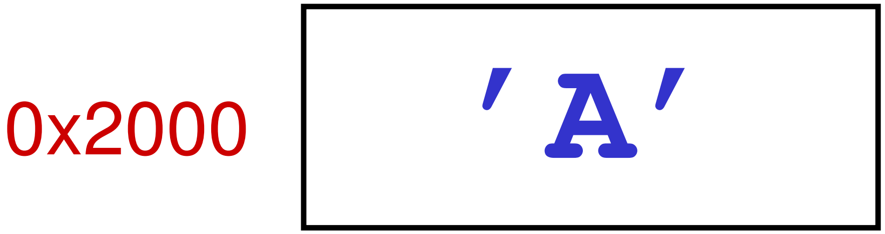
The & Operator

- Gives the memory address of an object

The & Operator

- Gives the memory address of an object

```
char ch = 'A';
```



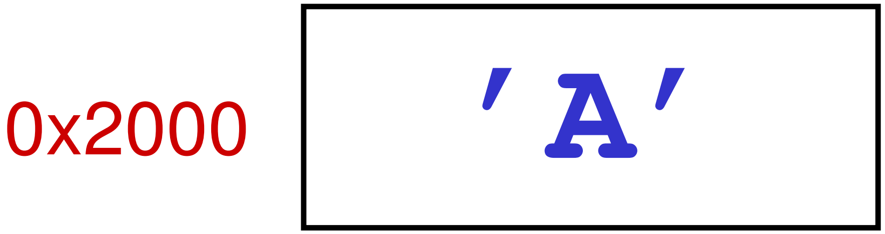
0x2000

'A'

The & Operator

- Gives the memory address of an object

```
char ch = ’A’;
```

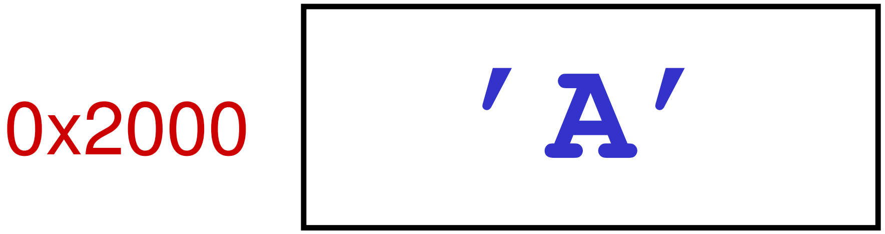


`&ch` yields the value 0x2000

The & Operator

- Gives the memory address of an object

```
char ch = 'A';
```



`&ch` yields the value 0x2000

- Also known as the "address operator"

***Example*:**

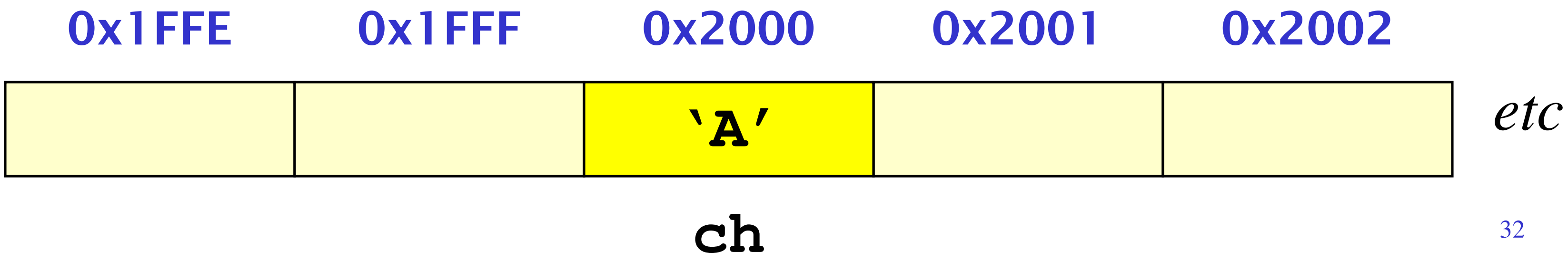
```
char ch;

printf(“%p”, &ch);
```

***Example*:**

```
char ch;
printf(“%p”, &ch);
```

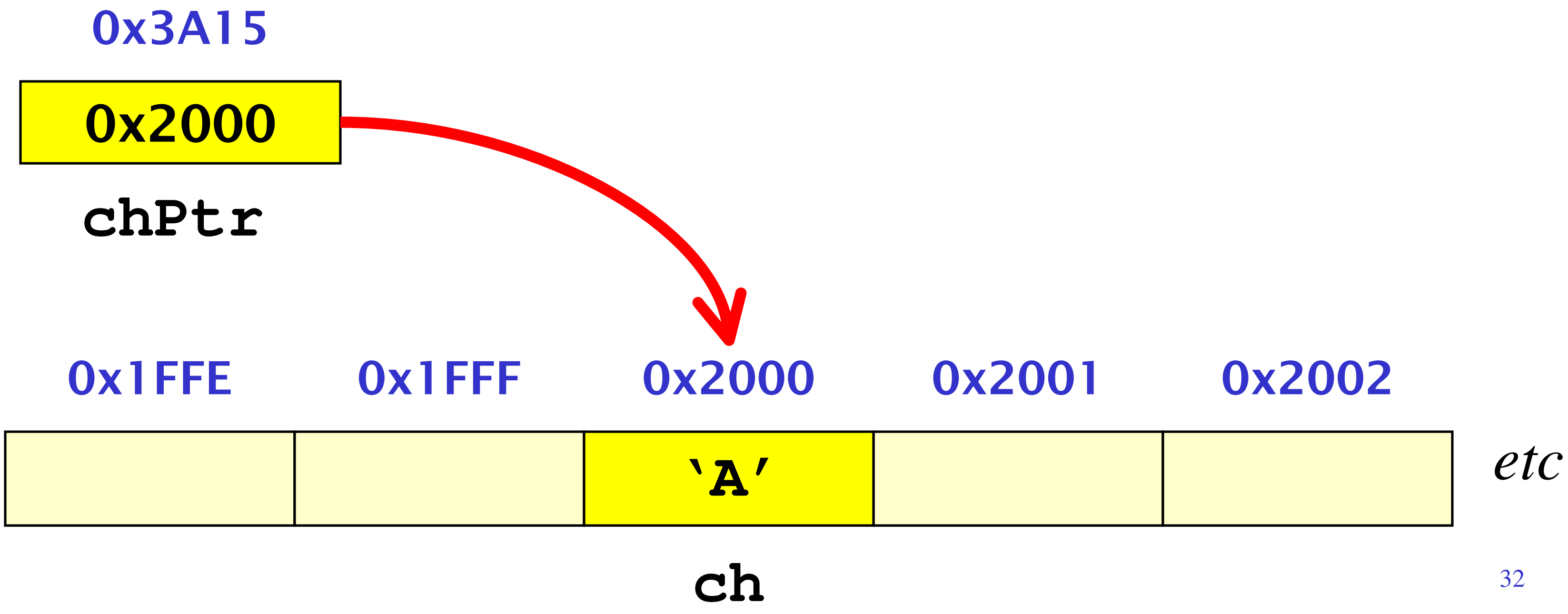
“conversion specifier” for printing a memory address

Pointers



0x1FFE	0x1FFF	0x2000	0x2001	0x2002	
		`'A'`			*etc*
		`ch`			

Pointers



0x3A15

0x2000

chPtr

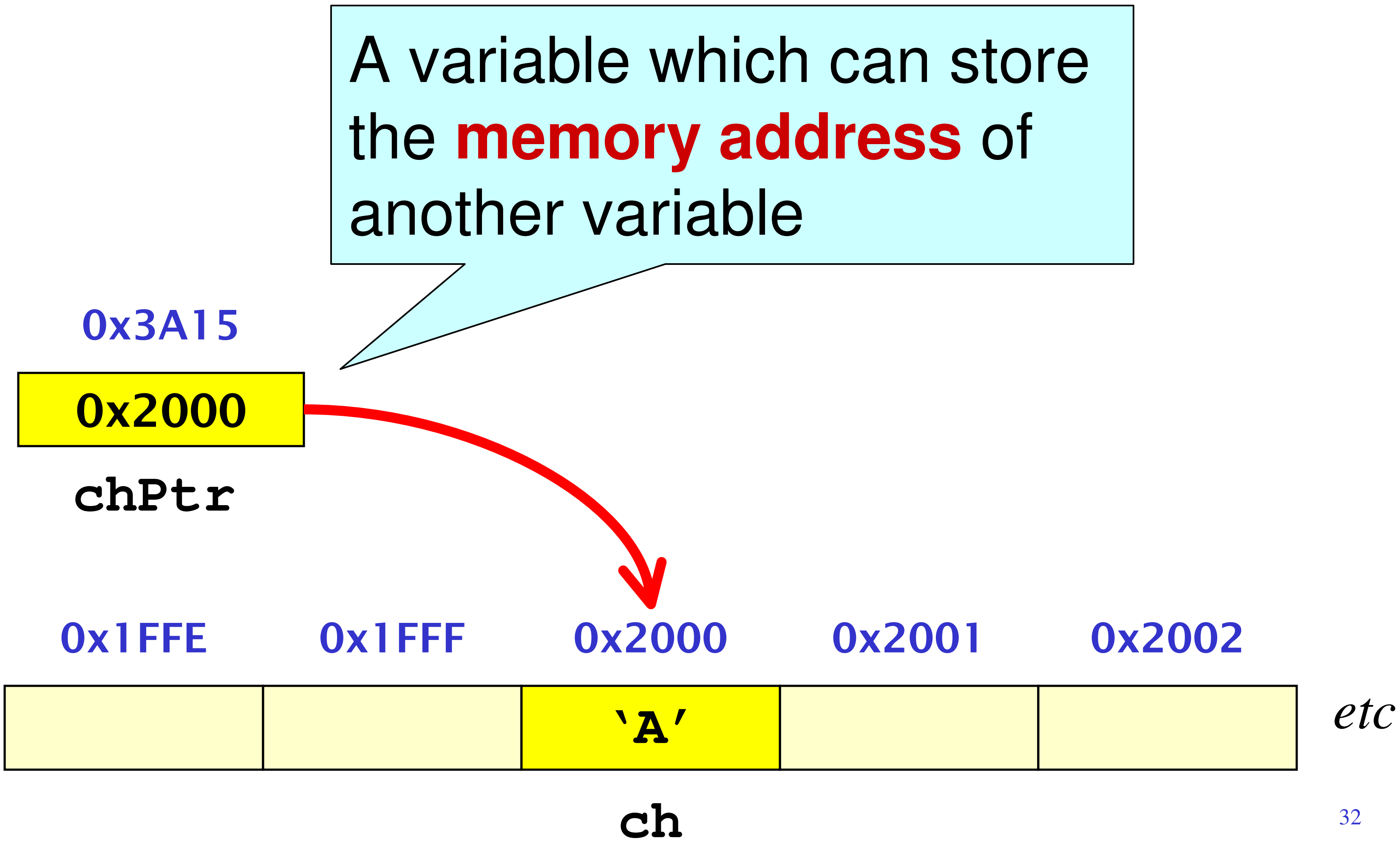
0x1FFE 0x1FFF 0x2000 0x2001 0x2002

‘A’

etc

ch

Pointers



A variable which can store the **memory address** of another variable

Pointers

Pointers

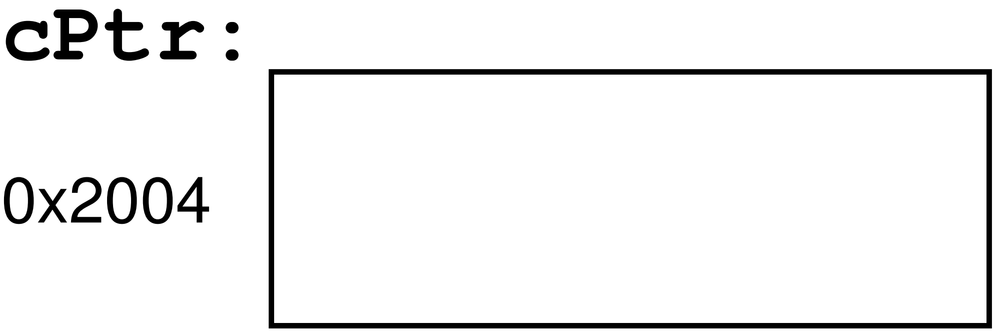
- A pointer is a variable which…
  - Contains a memory address
  - Points to a specific data type

Pointers

- A pointer is a variable which…
  - Contains a memory address
  - Points to a specific data type
- Pointer variables are usually named ***`varPtr`***

Example:

```
char* cPtr;
```

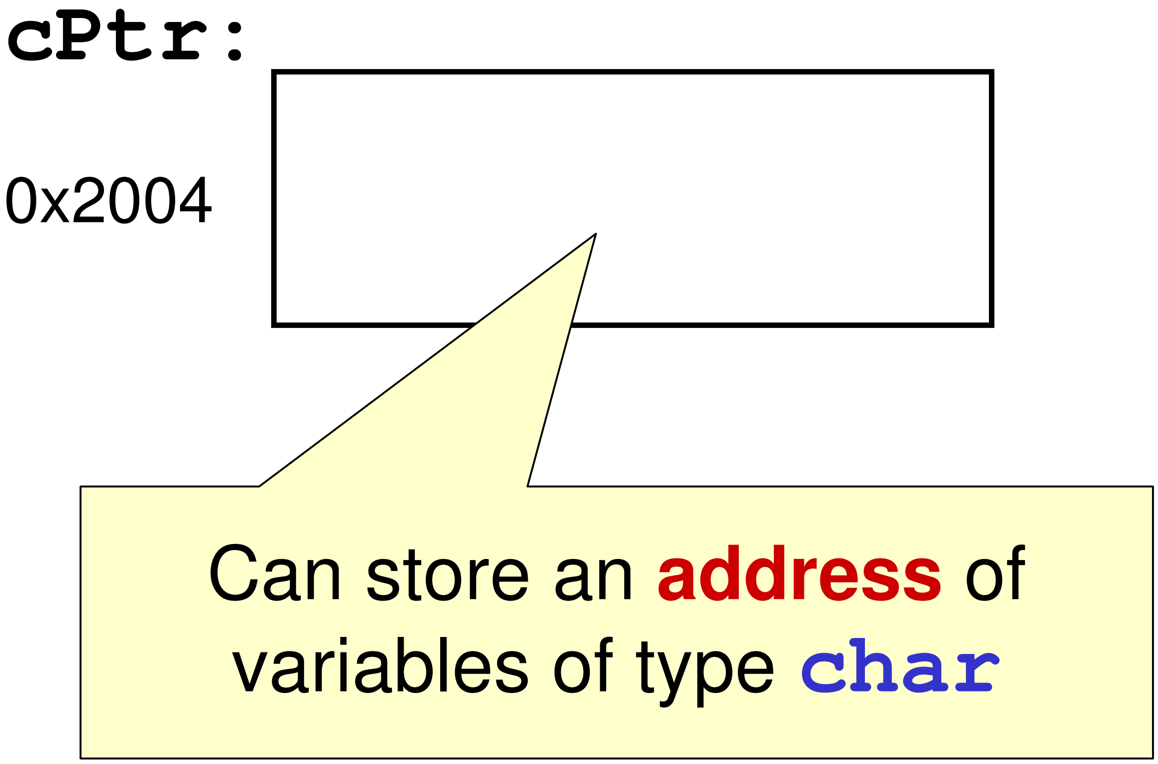


cPtr:

0x2004

Example:

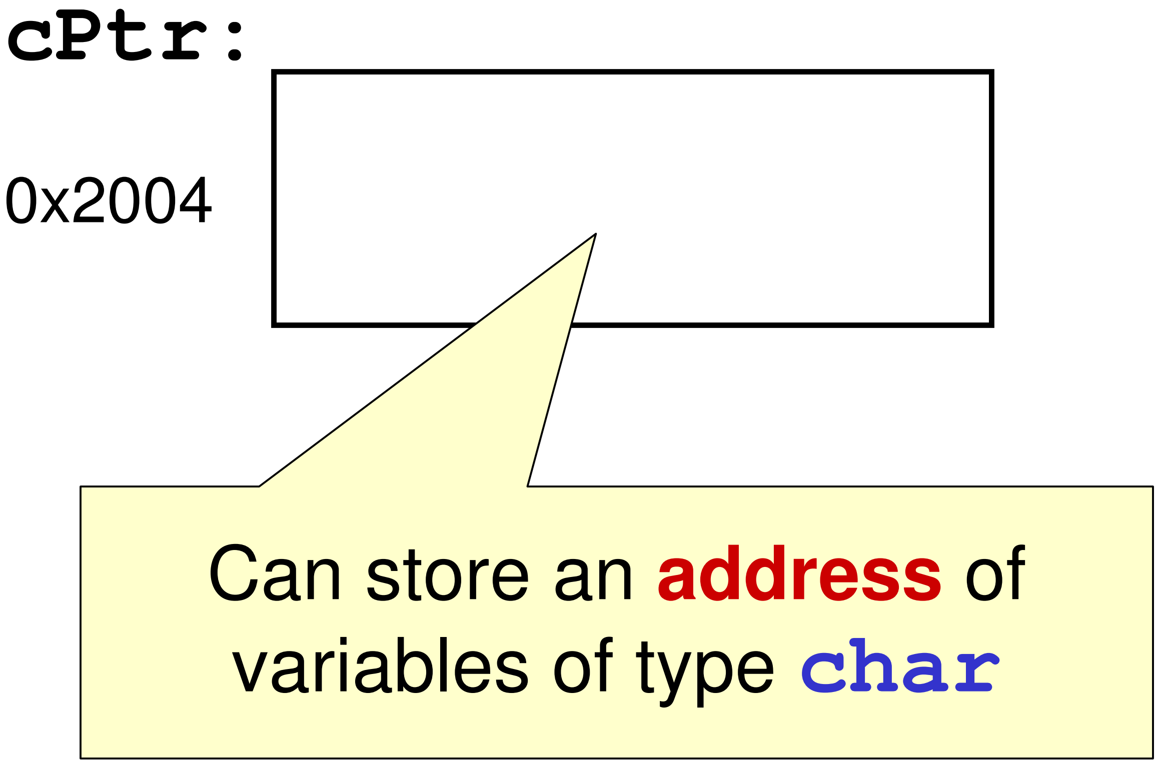
```
char* cPtr;
```



Can store an **address** of variables of type `char`

Example:

```
char* cPtr;
```



`cPtr:`

0x2004

Can store an **address** of variables of type **`char`**

- We say *cPtr* is a ***pointer*** to a character

Pointers and the & Operator

***Example*:**

Pointers and the & Operator

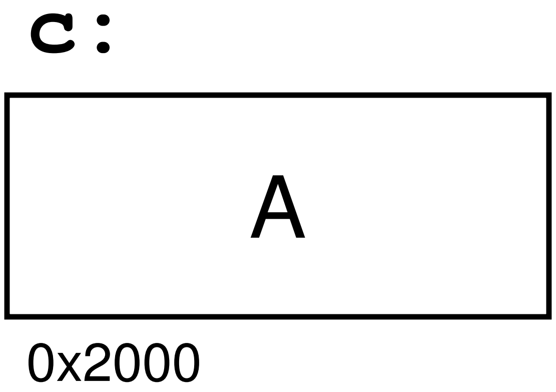
Example:

```
char c = ’A’;
```

Pointers and the & Operator

Example:

```
char c = 'A';
```



c:

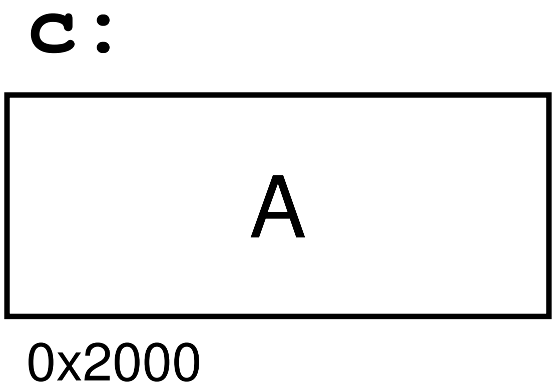
A

0x2000

Pointers and the & Operator

Example:

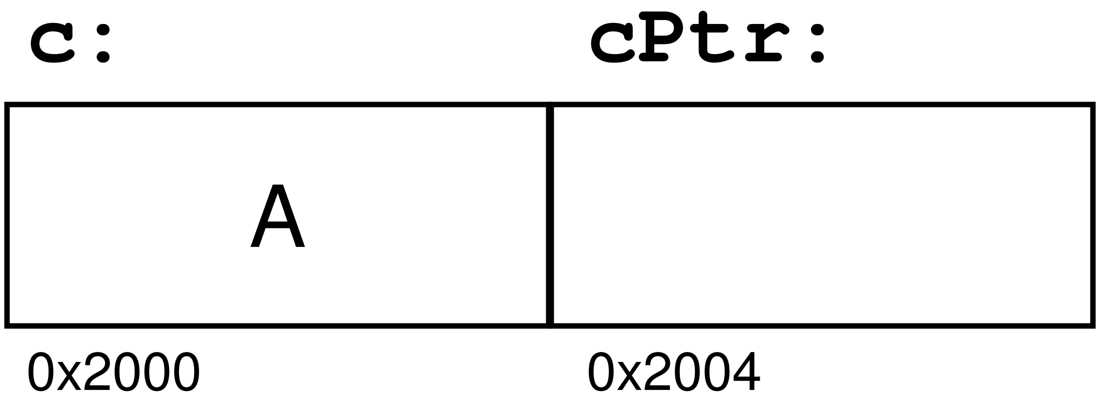
```
char c = 'A';
char *cPtr;
```



Pointers and the & Operator

Example:

```
char c = 'A';
char *cPtr;
```



c:

cPtr:

A

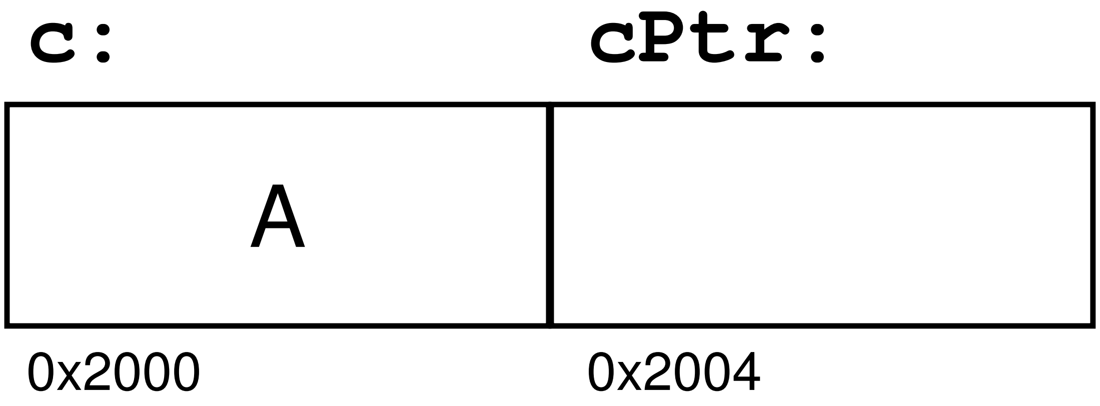
0x2000

0x2004

Pointers and the & Operator

Example:

```
char c = ’A’;
char *cPtr;


cPtr = &c;
```

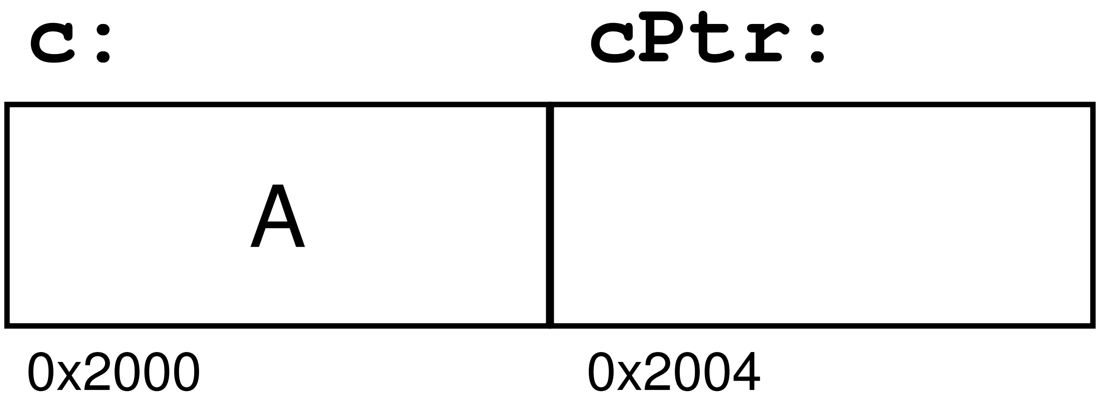


Pointers and the & Operator

Example:

```
char c = ’A’;
char *cPtr;


cPtr = &c;
```

Assigns the address of c to cPtr

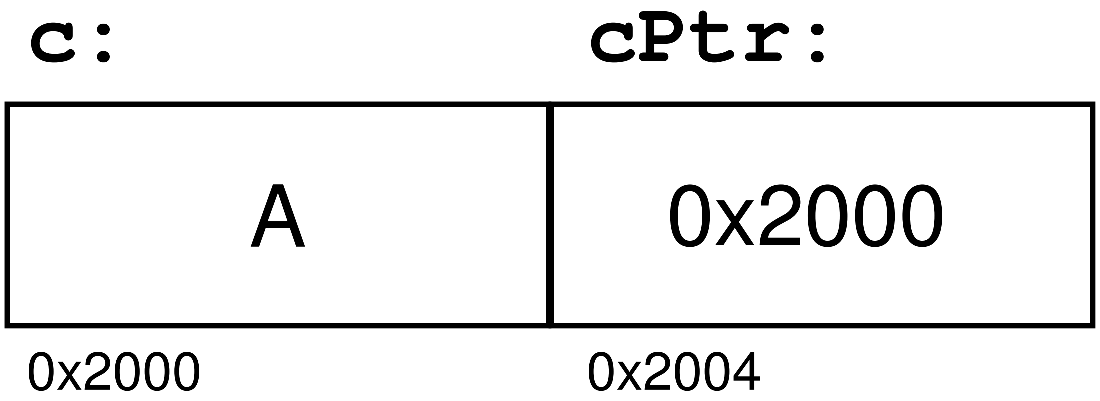


Pointers and the & Operator

Example:

```
char c = 'A';
char *cPtr;


cPtr = &c;
```

Assigns the address of c to cPtr



c:

A

0x2000

cPtr:

0x2000

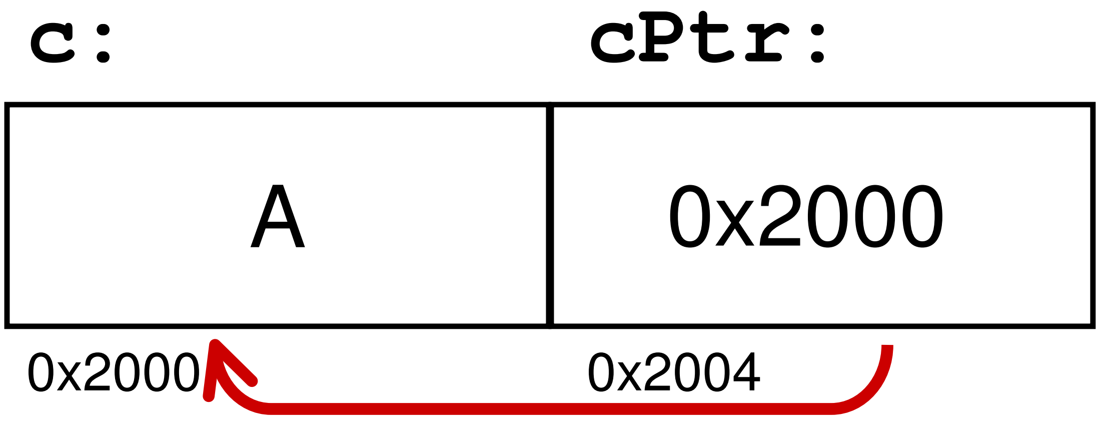
0x2004

Pointers and the & Operator

Example:

```
char c = 'A';
char *cPtr;


cPtr = &c;
```

Assigns the address of c to cPtr



Notes on Pointers

Notes on Pointers

- We can have pointers to any data type

***Example*:**

```
int*   numPtr;
float*  xPtr;
```

Notes on Pointers

- We can have pointers to any data type

Example:

```
int*    numPtr;
float*   xPtr;
```

- The * can be anywhere between the type and the variable

Example:

```
int     *numPtr;
float * xPtr;
```

Notes on Pointers (cont)

Notes on Pointers (cont)

- You can assign the address of a variable to a "compatible" pointer using the **&** operator

***Example*:**

```
int   aNumber;
int   *numPtr;

numPtr = &aNumber;
```

Notes on Pointers (cont)

- You can assign the address of a variable to a “compatible” pointer using the `&` operator

Example:

```
int   aNumber;
int   *numPtr;

numPtr = &aNumber;
```

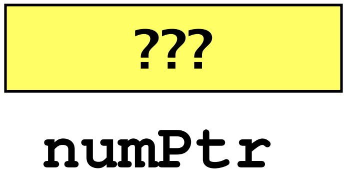
- You can print the address stored in a pointer using the `%p` conversion specifier

Example:

```
printf(“%p”, numPtr);
```

Notes on Pointers (cont)

```
int  *numPtr;
```



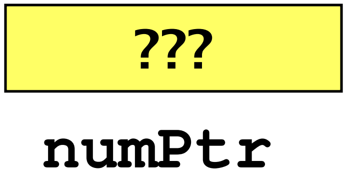
???

numPtr

Notes on Pointers (cont)

```
int   *numPtr;
```

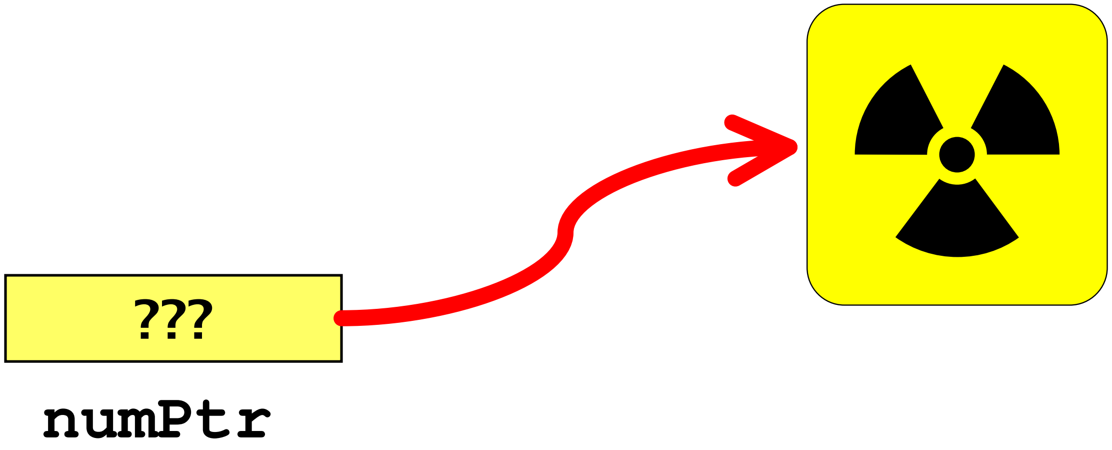
Beware of pointers which are not initialized!



Notes on Pointers (cont)

```
int  *numPtr;
```

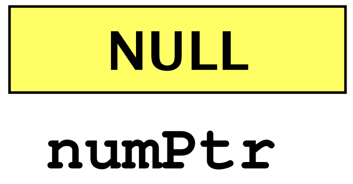
Beware of pointers which are not initialized!



Notes on Pointers (cont)

- When declaring a pointer, it is a good idea to always initialize it to **NULL** (a special pointer constant)

```
int  *numPtr = NULL;
```



The * Operator

The * Operator

- Allows pointers to access variables they point to

The * Operator

- Allows pointers to access variables they point to
- Also known as “dereferencing operator”

The * Operator

- Allows pointers to access variables they point to
- Also known as "dereferencing operator"
- Should not be confused with the * in the pointer declaration

Pointers and the * Operator

***Example*:**

Pointers and the * Operator

Example:

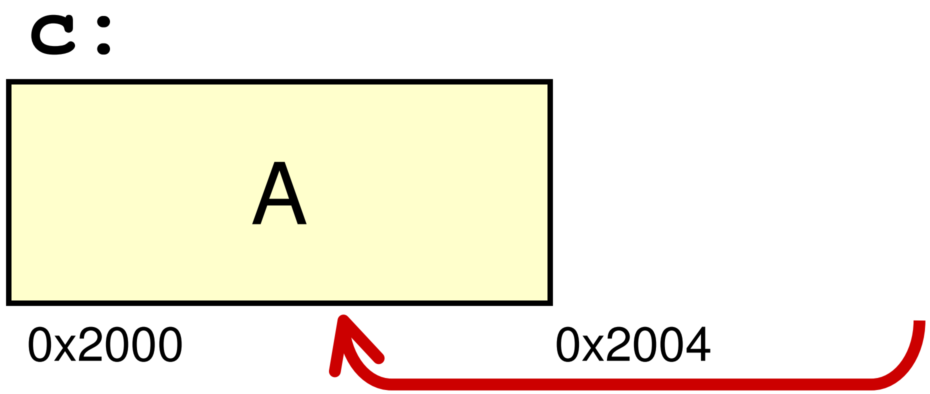
```
char c = 'A';
```

0x2004

Pointers and the * Operator

***Example*:**

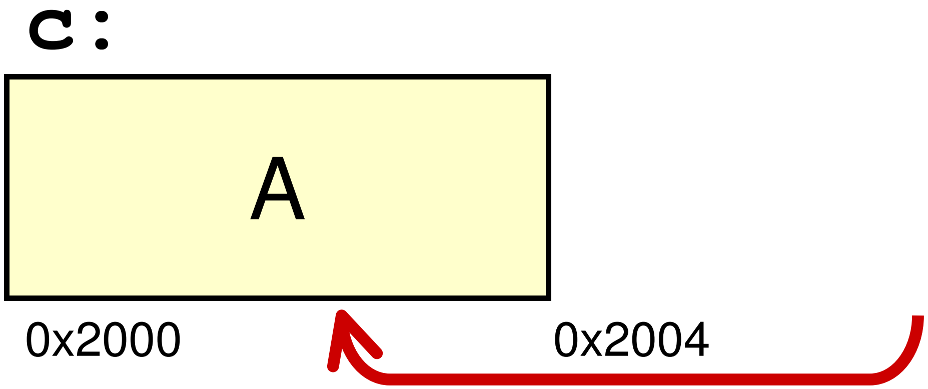
```
char c = 'A';
```



Pointers and the * Operator

Example:

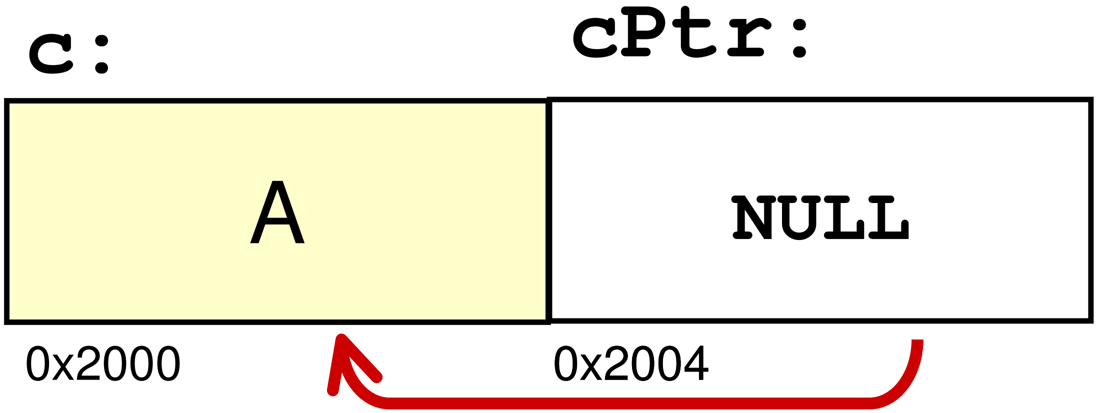
```
char c = 'A';
char *cPtr = NULL;
```



Pointers and the * Operator

***Example*:**

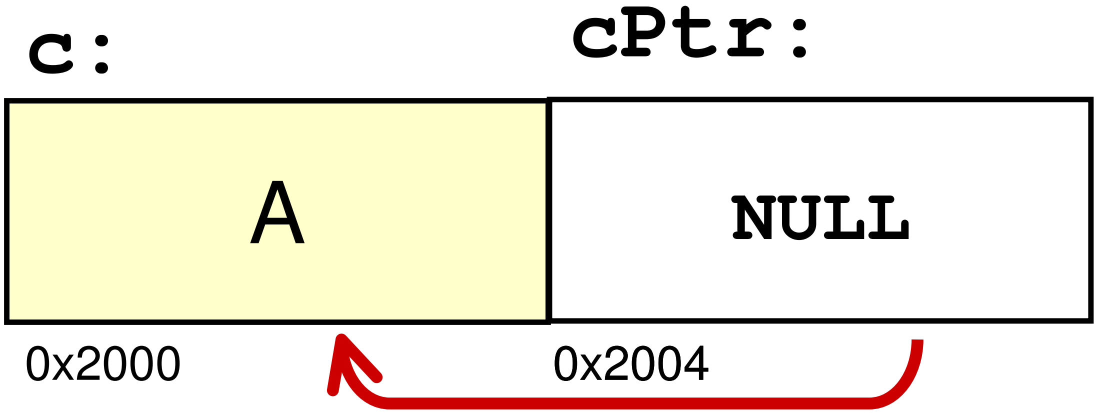
```
char c = 'A';
char *cPtr = NULL;
```



Pointers and the * Operator

***Example*:**

```
char c = 'A';
char *cPtr = NULL;


cPtr = &c;
```



c:

cPtr:

A

NULL

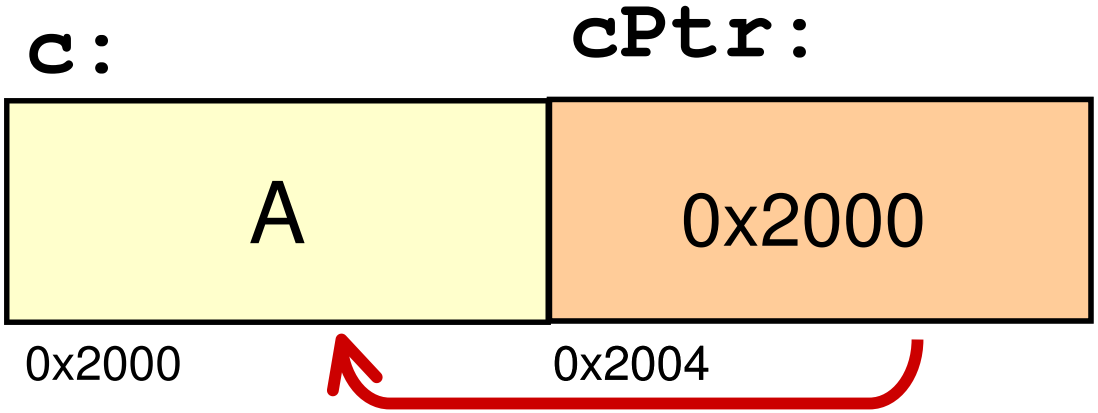
0x2000

0x2004

Pointers and the * Operator

***Example*:**

```
char c = ’A’;
char *cPtr = NULL;


cPtr = &c;
```



c:

cPtr:

A

0x2000

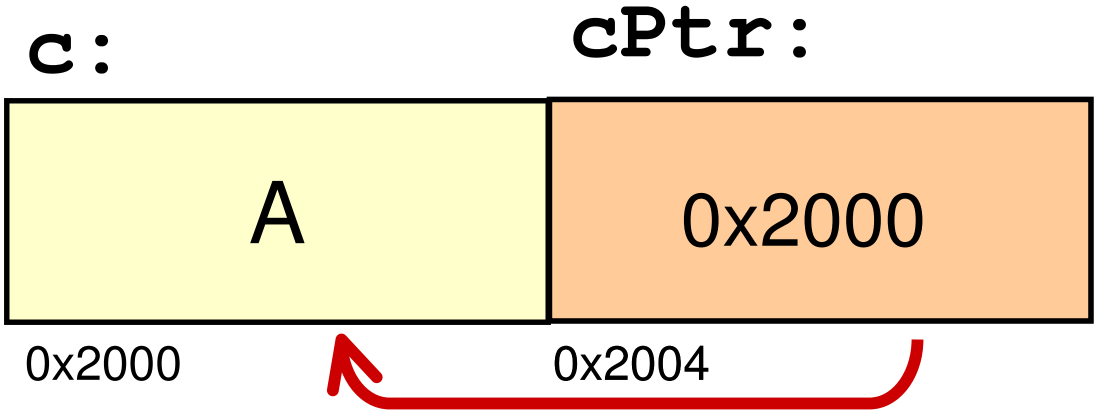
0x2000

0x2004

Pointers and the * Operator

Example:

```
char c = ’A’;
char *cPtr = NULL;

cPtr = &c;
*cPtr = ’B’;
```



c:

cPtr:

A

0x2000

0x2000

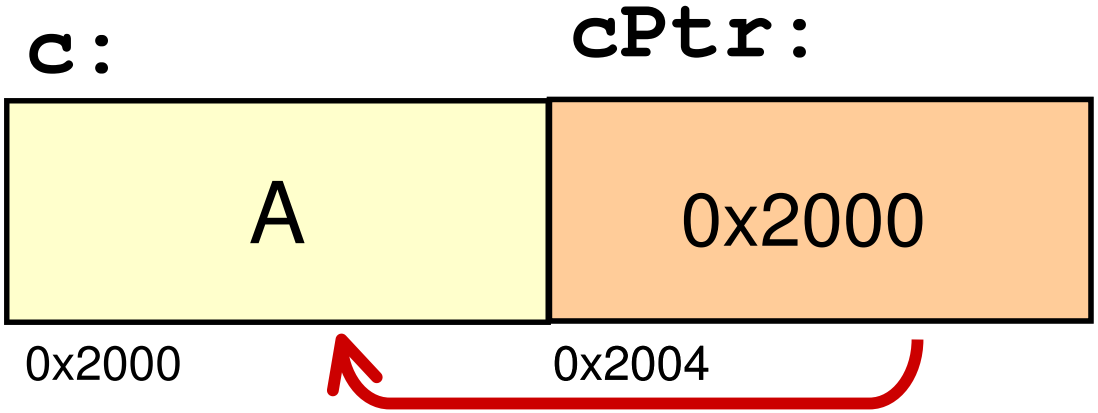
0x2004

Pointers and the * Operator

Example:

```
char c = 'A';
char *cPtr = NULL;


cPtr = &c;
*cPtr = 'B';
```

Changes the value of the variable which cPtr points to

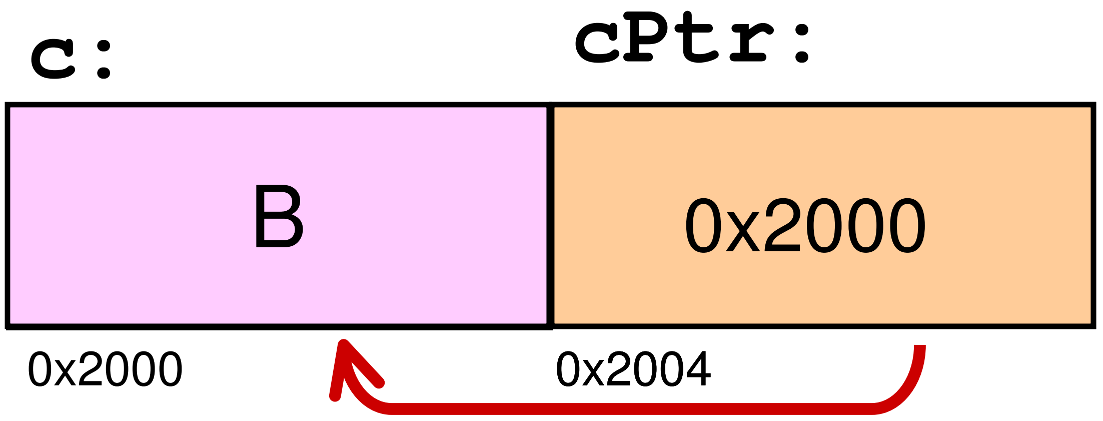


Pointers and the * Operator

***Example*:**

```
char c = 'A';
char *cPtr = NULL;


cPtr = &c;
*cPtr = 'B';
```

Changes the value of the variable which* cPtr *points to



c:

cPtr:

B

0x2000

0x2000

0x2004

Lessons

- A pointer is simply a variable that contains an address