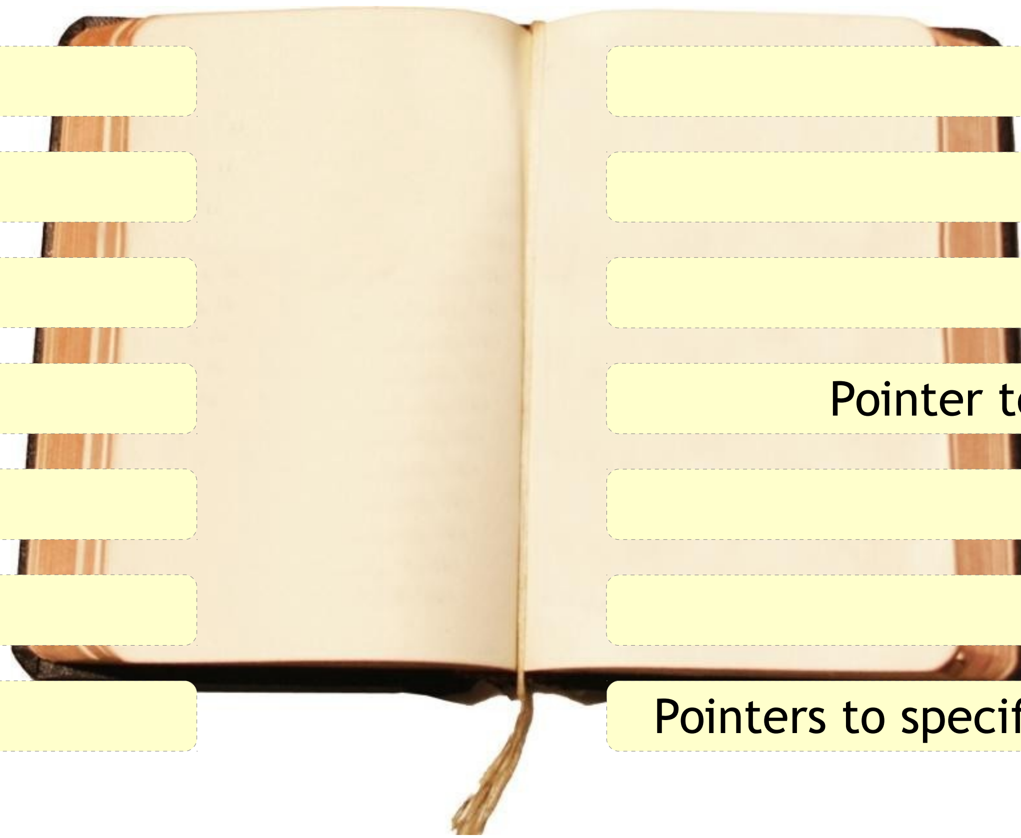
Pointers

- What's a Jargon?
  - Jargon may refer to terminology used in a certain profession, such as computer jargon, or it may refer to any nonsensical language that is not understood by most people.
  - Speech or writing having unusual or pretentious vocabulary, convoluted phrasing, and vague meaning.
- Pointer are perceived difficult
  - Because of jargonification
- So, let's de-jargonify & understand them

Advanced C

Pointers – Analogy with Book



A Book	System Memory
Contains pages	Memory Pages
Front matter	Boot Code
Table of Contents	Pointer to different programs
Chapters	Code Segments
Notes Section	Data & Stack
Indexes	Pointers to specific memory locations

Advanced C

Pointers – Computers

- Just like a book analogy, Computers contains different different sections (Code) in the memory
- All sections have different purposes
- Every section has a address and we need to point to them whenever required
- In fact everything (Instructions and Data) in a particular section has an address!!
- So the pointer concept plays a big role here

Advanced C

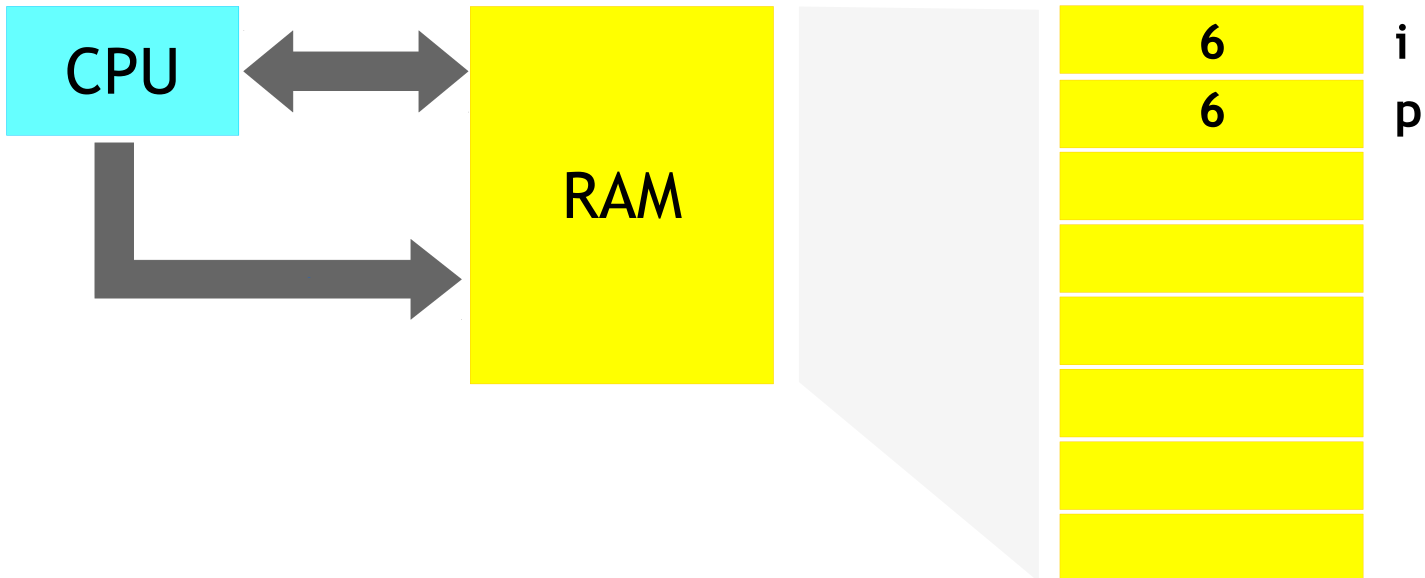
Pointers – Why?

- To have C as a low level language being a high level language
- Returning more than one value from a function
- To achieve the similar results as of ”pass by value”
- parameter passing mechanism in function, by passing the reference
- To have the dynamic allocation mechanism

- Rule 1 - Pointer is an Integer
- Rule 2 - Referencing and De-referencing
- Rule 3 - Pointing means Containing
- Rule 4 - Pointer Type
- Rule 5 - Pointer Arithmetic
- Rule 6 - Pointing to Nothing
- Rule 7 - Static vs Dynamic Allocation

Advanced C

Pointers – The 7 Rules – Rule 1



Integer i;
Pointer p;
Say:
    i = 6;
    p = 6;

Advanced C

Pointers – The 7 Rules – Rule 1

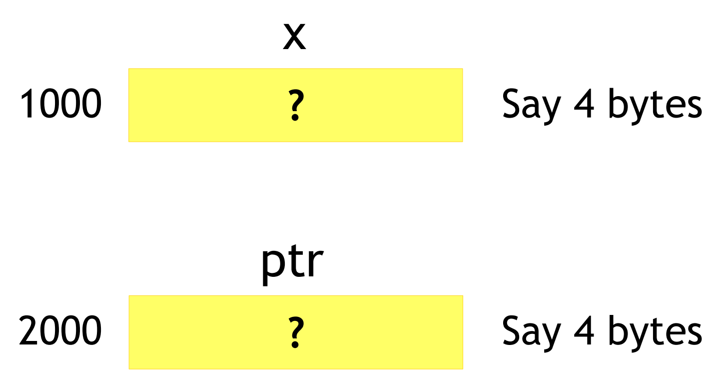
- Whatever we put in data bus is Integer
- Whatever we put in address bus is Pointer
- So, at concept level both are just numbers. May be of different sized buses
- Rule: “Pointer is an Integer”
- Exceptions:
  - May not be address and data bus of same size
  - Rule 2 (Will see why? while discussing it)

Advanced C

Pointers - Rule 1 in detail

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = 5;

    return 0;
}
```

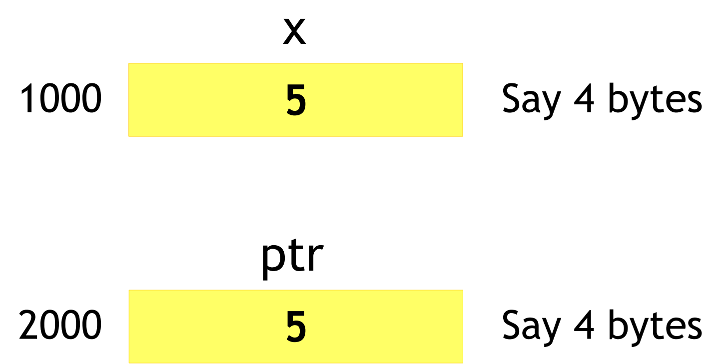


Advanced C

Pointers - Rule 1 in detail

Example

```
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = 5;

    return 0;
}
```



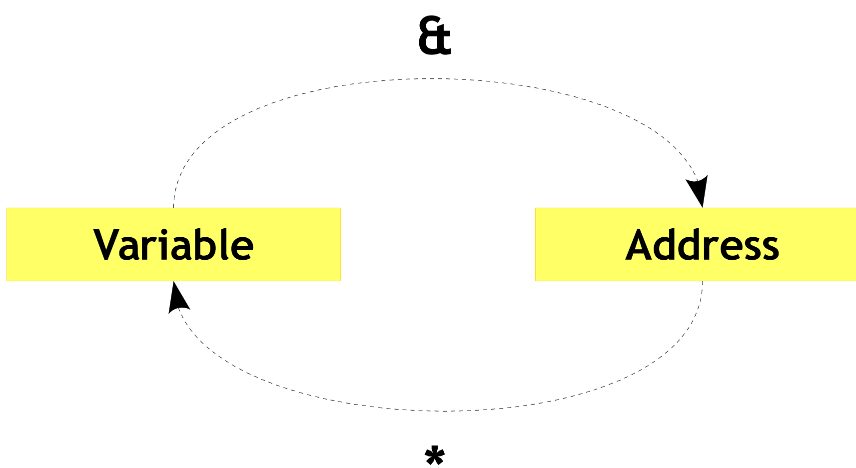
- So pointer is an integer
- But remember the “They may not be of same size”

  32 bit system = 4 Bytes

  64 bit system = 8 Bytes

Advanced C
Pointers - The 7 Rules - Rule 2

- Rule : “Referencing and Dereferencing”

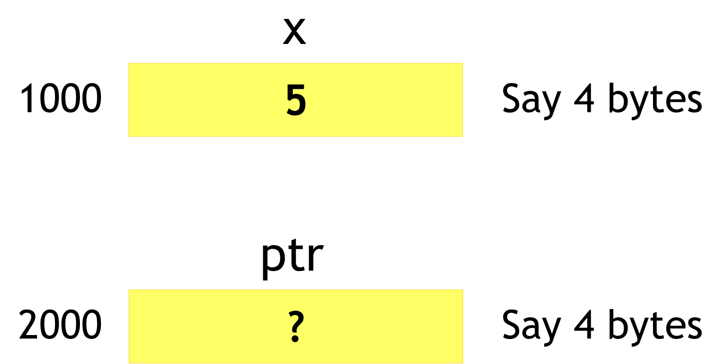


Advanced C

Pointers – Rule 2 in detail

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;

    return 0;
}
```



- Considering the image, What would the below line mean?

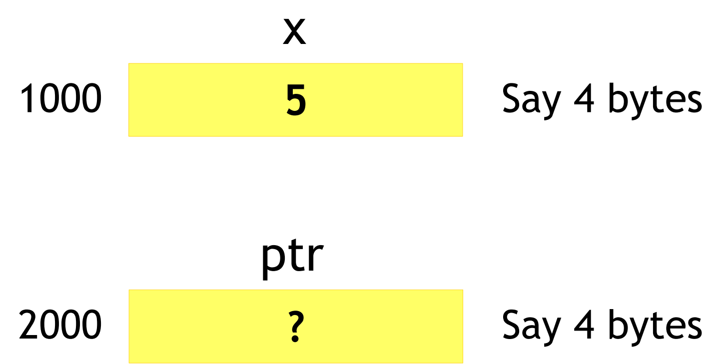
  * 1000

Advanced C

Pointers – Rule 2 in detail

Example

```
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;

    return 0;
}
```



- Considering the image, What would the below line mean?

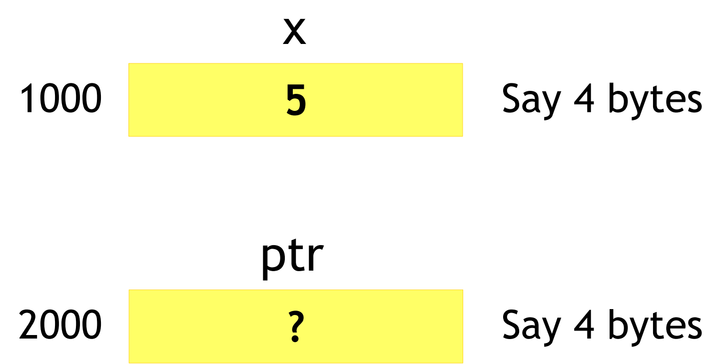
  * 1000

Advanced C

Pointers – Rule 2 in detail

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;

    return 0;
}
```



- Considering the image, What would the below line mean?

  * 1000

- Goto to the location 1000 and fetch its value, so

  * 1000 → 5

Advanced C

Pointers – Rule 2 in detail

Example

```
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = &x;

    return 0;
}
```

	x	
1000	5	Say 4 bytes
	ptr	
2000	?	Say 4 bytes

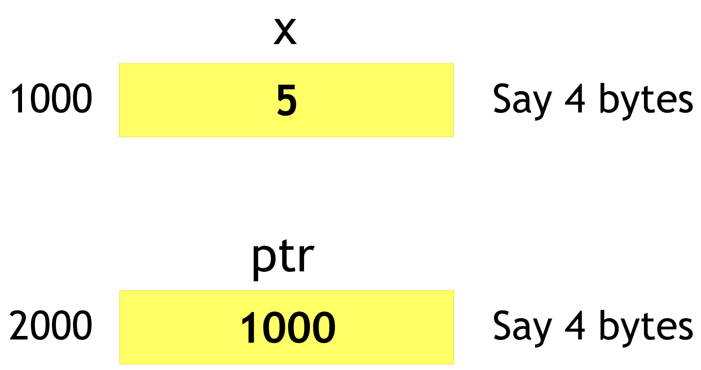
- What should be the change in the above diagram for the above code?

Advanced C

Pointers – Rule 2 in detail

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = &x;

    return 0;
}
```



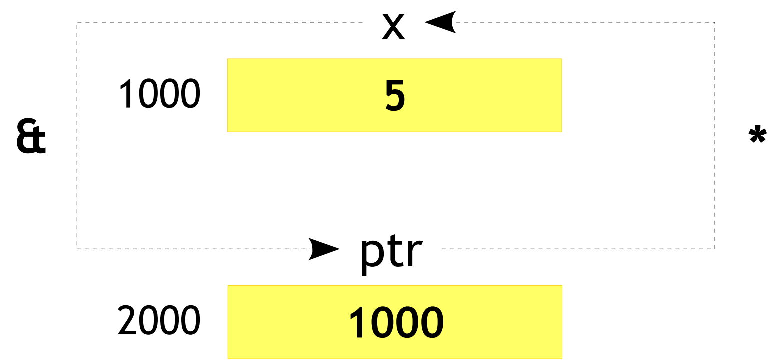
- So pointer should contain the address of a variable
- It should be a valid address

Advanced C

Pointers – Rule 2 in detail

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = &x;

    return 0;
}
```



“Prefix 'address of operator' (&) with variable (x) to get its address and store in the pointer”

“Prefix 'indirection operator' (*) with pointer to get the value of variable (x) it is pointing to”

Advanced C

Pointers – Rule 2 in detail

Example

```
#include <stdio.h>

int main()
{
    int number = 10;
    int *ptr;

    ptr = &number;

    printf("Address of number is %p\n", &number);
    printf("ptr contains %p\n", ptr);

    return 0;
}
```

Advanced C

Pointers – Rule 2 in detail

Example

```
#include <stdio.h>

int main()
{
    int number = 10;
    int *ptr;

    ptr = &number;

    printf("number contains %d\n", number);
    printf("*ptr contains %d\n", *ptr);

    return 0;
}
```

Advanced C

Pointers – Rule 2 in detail

Example

```c
#include <stdio.h>

int main()
{
    int number = 10;
    int *ptr;

    ptr = &number;
    *ptr = 100;

    printf("number contains %d\n", number);
    printf("*ptr contains %d\n", *ptr);

    return 0;
}
```

- So, from the above code we can conclude

“*ptr <=> number”

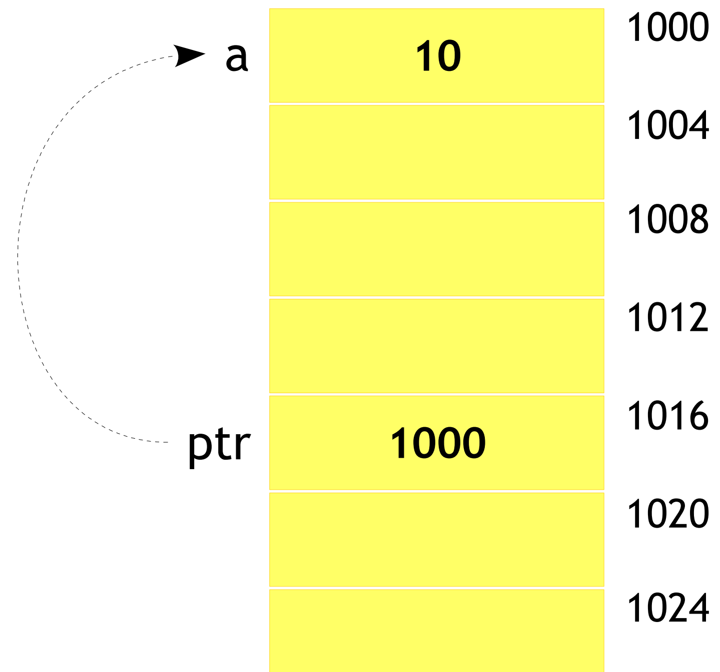
Advanced C

Pointers – The 7 Rules – Rule 3

- Pointer pointing to a Variable = Pointer contains the Address of the Variable
- Rule: “Pointing means Containing”

Example

```c
#include <stdio.h>

int main()
{
    int a = 10;
    int *ptr;

    ptr = &a;

    return 0;
}
```



Advanced C

Pointers – The 7 Rules – Rule 4

- Types to the pointers
- What??, why do we need types attached to pointers?

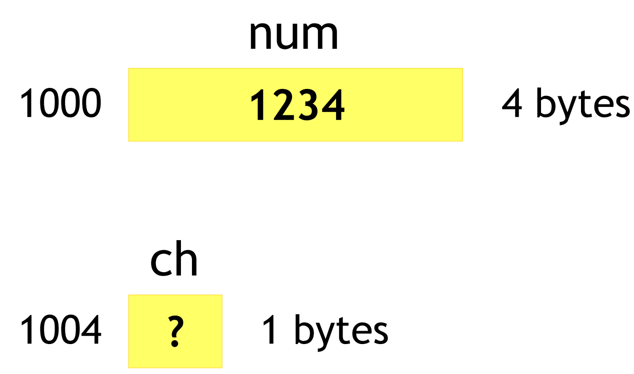
Advanced C

Pointers – Rule 4 in detail

- Does address has a type?

Example

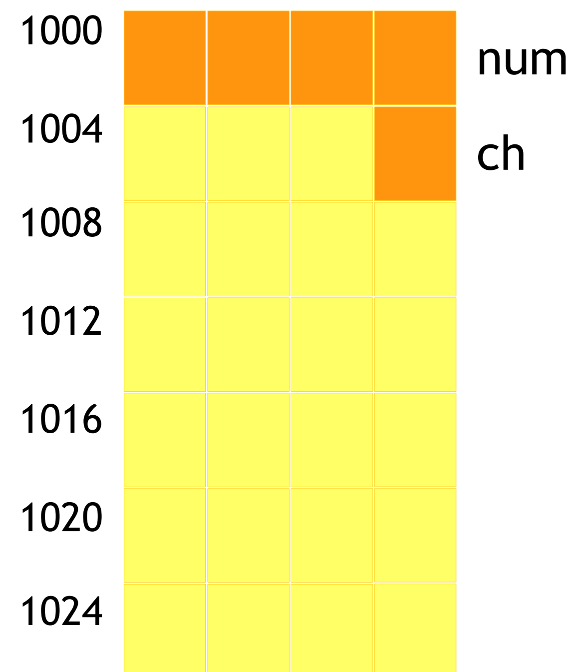
```c
#include <stdio.h>

int main()
{
    int num = 1234;
    char ch;

    return 0;
}
```



- So from the above diagram can we say &num → 4 bytes and &ch → 1 byte?

Advanced C
Pointers – Rule 4 in detail

- The answer is no!!
- Address size does not depend on type of the variable
- It depends on the system we use and remains same across all pointers
- Then a simple questions arises “why type is used with pointers?”

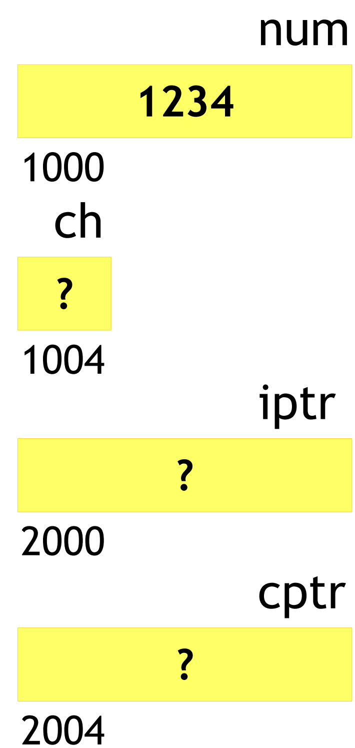


Advanced C

Pointers – Rule 4 in detail

Example

```c
#include <stdio.h>

int main()
{
    int num = 1234;
    char ch;

    int *iptr;
    char *cptr;

    return 0;
}
```



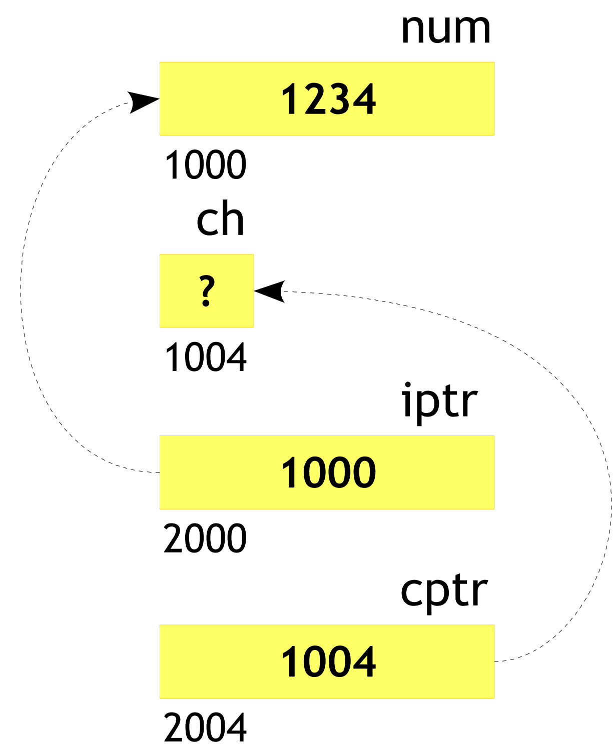
- Lets consider above example to understand it
- Say we have an integer and a character pointer

Advanced C

Pointers – Rule 4 in detail

Example

```c
#include <stdio.h>

int main()
{
    int num = 1234;
    char ch;

    int *iptr = &num;
    char *cptr = &ch;

    return 0;
}
```

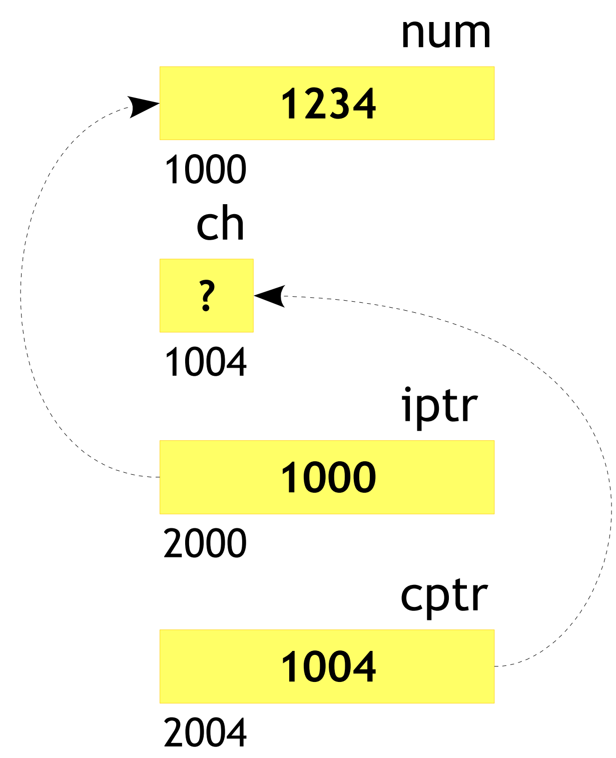


- Lets consider the above examples to understand it
- Say we have a integer and a character pointer

Advanced C

Pointers – Rule 4 in detail

- With just the address, can we know what data is stored?
- How would we know how much data to fetch for the address it is pointing to?
- Eventually the answer would be NO!!



Advanced C

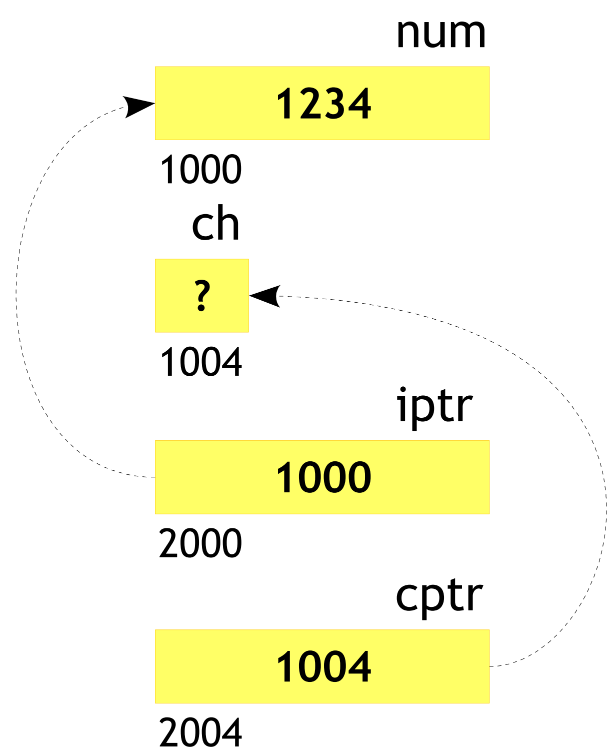
Pointers – Rule 4 in detail

- From the diagram right side we can say

  *cptr fetches a single byte

  *iptr fetches 4 consecutive bytes

- So, in conclusion we can say



(type *) → fetch sizeof(type) bytes

Advanced C

Pointers – Rule 4 in detail - Endianness

- Since the discussion is on the data fetching, its better we have knowledge of storage concept of machines
- The Endianness of the machine
- What is this now!!?
  - Its nothing but the byte ordering in a word of the machine
- There are two types
  - Little Endian – LSB in Lower Memory Address
  - Big Endian – MSB in Lower Memory Address

Advanced C

Pointers – Rule 4 in detail - Endianness

- LSB (Least Significant Byte)
  - The byte of a multi byte number with the least importance
  - The change in it would have least effect on number's value change
- MSB (Most Significant Byte)
  - The byte of a multi byte number with the most importance
  - The change in it would have larger effect on number's value change

Pointers – Rule 4 in detail - Endianness

Example

```c
#include <stdio.h>

int main()
{
    int num = 0x12345678;

    return 0;
}
```

- Let us consider the following example and how it would be stored in both machine types

Big Endian

1000	12	34	56	78	num
1004					

→

1000	1001	1002	1003
12	34	56	78

Little Endian

1000	78	56	34	12	num
1004					

→

1000	1001	1002	1003
78	56	34	12

Advanced C

Pointers – Rule 4 in detail - Endianness

- OK Fine. What now? How is it going to affect the fetch and modification?
- Let us consider the same example put in the previous slide

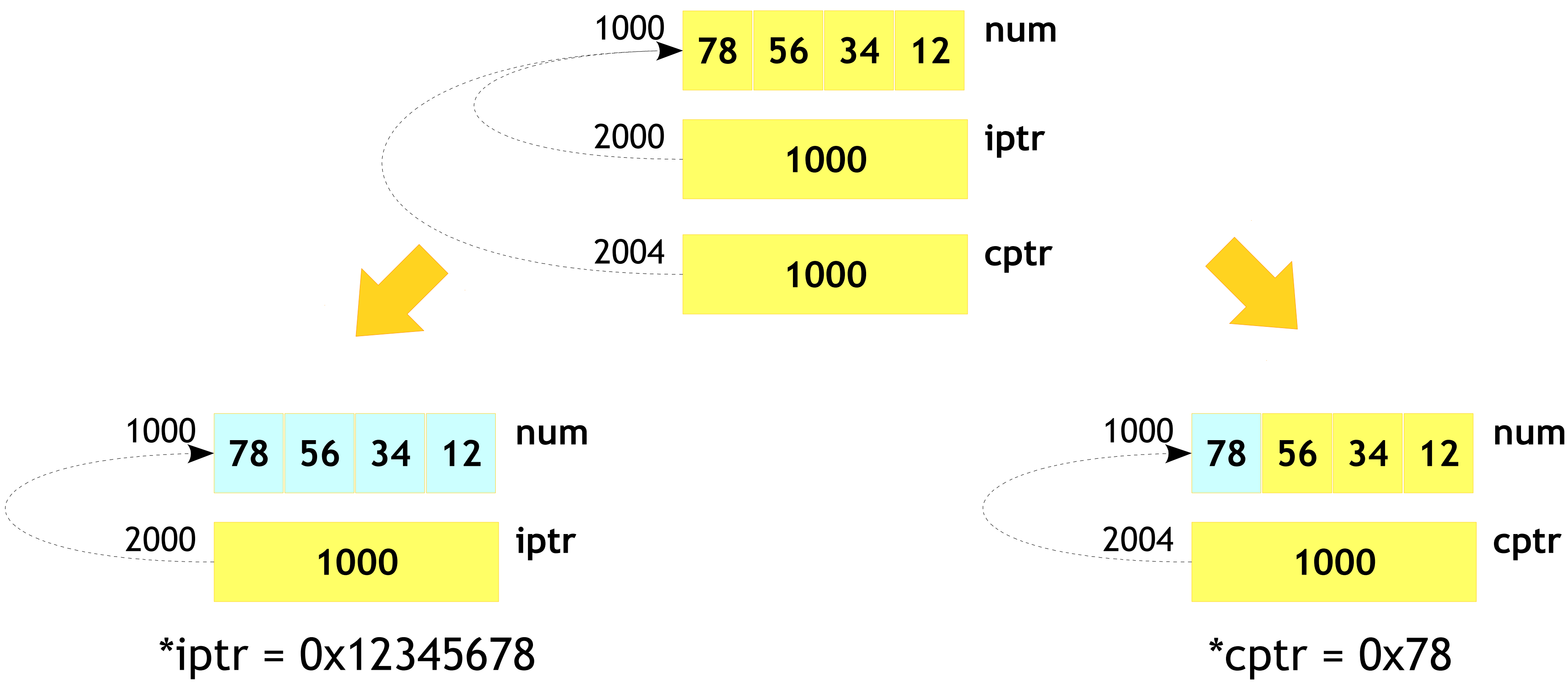
Example

```
#include <stdio.h>

int main()
{
    int num = 0x12345678;
    int *iptr, char *cptr;

    iptr = &num;
    cptr = &num;

    return 0;
}
```

- First of all is it possible to access a integer with character pointer?
- If yes, what should be the effect on access?
- Let us assume a Litte Endian system

Advanced C

Pointers – Rule 4 in detail - Endianness



- So from the above diagram it should be clear that when we do cross type accessing, the endianness should be considered

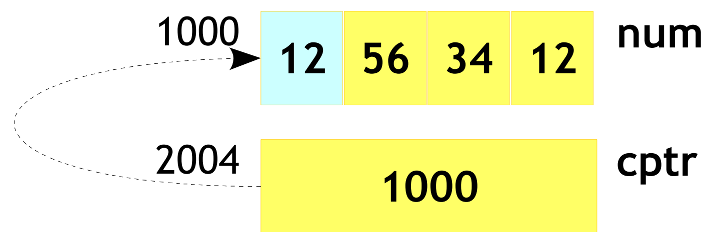
Advanced C

Pointers - The 7 Rules - Rule 4

Example

```
#include <stdio.h>

int main()
{
    int num = 0x12345678;
    char ch;

    int *iptr = &num;
    char *cptr = &num;

    *cptr = 0x12;

    return 0;
}
```

- So changing *cptr will change only the byte its pointing to



*cptr = 0x12

- So *iptr would contain 0x12345612 now!!

Advanced C

Pointers - The 7 Rules - Rule 4

- In conclusion,
  - The type of a pointer represents it's ability to perform read or write operations on number of bytes (data) starting from address its pointing to
  - Size of all different type pointers remains same

Example

```c
#include <stdio.h>

int main()
{
    if (sizeof(char *) == sizeof(long long *))
    {
        printf("Yes its Equal\n");
    }

    return 0;
}
```

- WAP to check whether a machine is Little or Big Endian

- Pointer Arithmetic

  Rule: “Value(p + i) = Value(p) + i * sizeof(*p)”

Advanced C

Pointers – The Rule 5 in detail

- Before proceeding further let us understand an array interpretation
  - Original Big Variable (bunch of variables, whole array)
  - Constant Pointer to the 1st Small Variable in the bunch (base address)
- When first interpretation fails than second interpretation applies

Pointers – The Rule 5 in detail

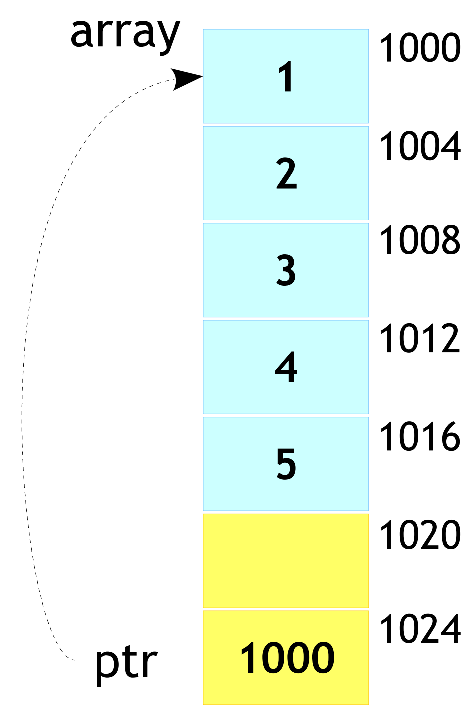
- Cases when first interpretation applies
  - When name of array is operand to sizeof operator
  - When “address of operator (&)” is used the with name of array while performing pointer arithmetic
- Following are the cases when first interpretation fails
  - When we pass array name as function argument
  - When we assign an array variable to pointer variable

Advanced C

Pointers - The Rule 5 in detail

Example

```c
#include <stdio.h>

int main()
{
    int array[5] = {1, 2, 3, 4, 5};
    int *ptr = array;

    return 0;
}
```



- So,

  Address of array = 1000

  Base address = 1000

  &array[0] = 1 → 1000

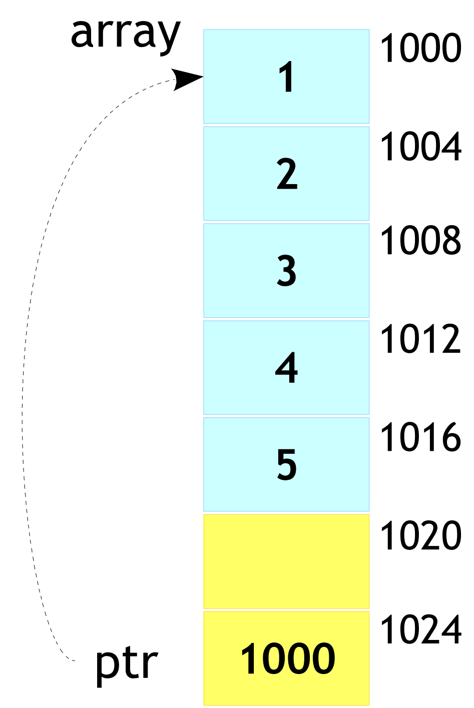
  &array[1] = 2 → 1004

Advanced C

Pointers – The Rule 5 in detail

Example

```c
#include <stdio.h>

int main()
{
    int array[5] = {1, 2, 3, 4, 5};
    int *ptr = array;

    printf(“%d\n”, *ptr);

    return 0;
}
```



- This code should print 1 as output since its points to the base address
- Now, what should happen if we do

  ptr = ptr + 1;

Advanced C

Pointers – The Rule 5 in detail

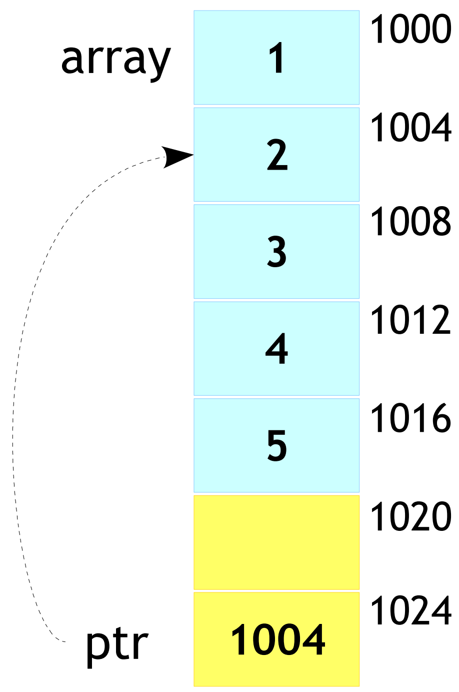
- ptr = ptr + 1;
- The above line can be discribed as follows
- ptr = ptr + 1 * sizeof(data type)
- In this example we have a integer array, so
- ptr = ptr + 1 * sizeof(int)

  = ptr + 1 * 4

  = ptr + 4
- Here ptr = 1000 so

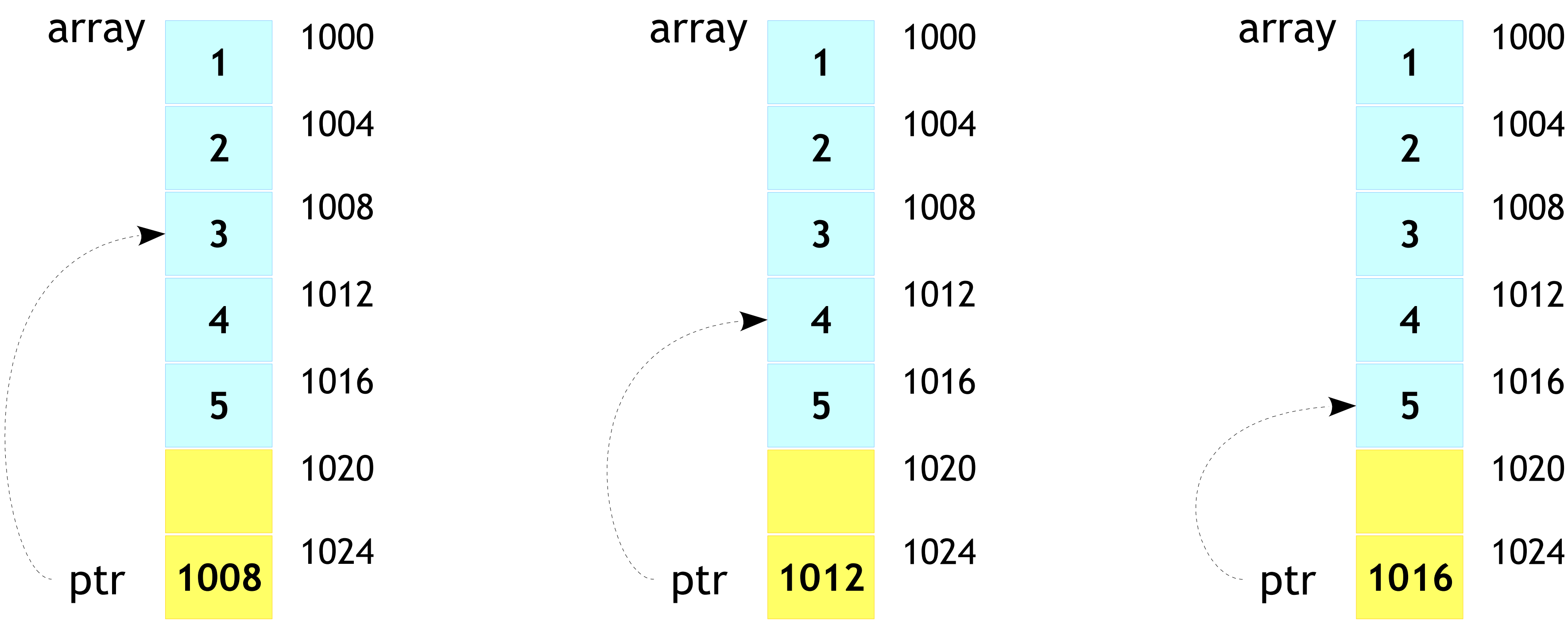
  = 1000 + 4

  = 1004



Advanced C

Pointers – The Rule 5 in detail

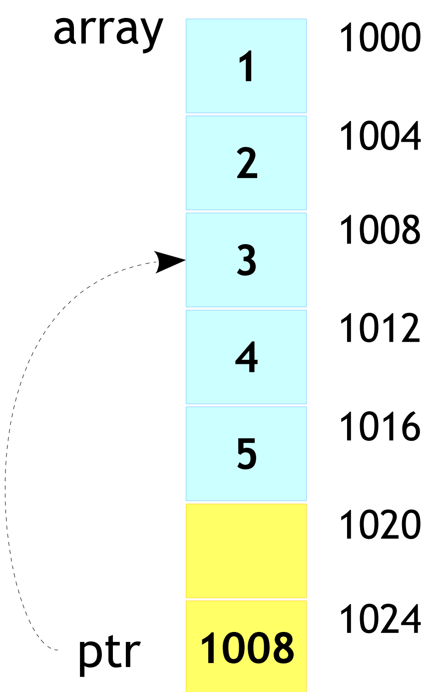


ptr = ptr + 2;   ptr = ptr + 3;   ptr = ptr + 4;

- Why does the compiler does this?. Just for convenience

Advanced C

Pointers – The Rule 5 in detail



ptr = ptr + 2;

- Relation with array can be explained as

  ptr + 2

  ptr + 2 * sizeof(int)

  1000 + 2 * 4

  1008 → &array[2]

- So,

  ptr + 2 → 1008 → &array[2]

  *(ptr + 2) → *(1008) → array[2]

Advanced C

Pointers – The Rule 5 in detail

- So to access a array element using a pointer would be

*(ptr + i) → array[i]

- This can be written as following too!!

array[i] → *(array + i)

- Which results to

ptr = array

- So as summary the below line also becomes valid because of second array interpretation

```
int *ptr = array;
```

- Wait can I write

$$*(ptr + i) \rightarrow *(i + ptr)$$

- Yes. So than can I write

$$array[i] \rightarrow i[array]$$

- Yes. You can index the element in both the ways

Advanced C

Pointers – The 7 Rules – Rule 6

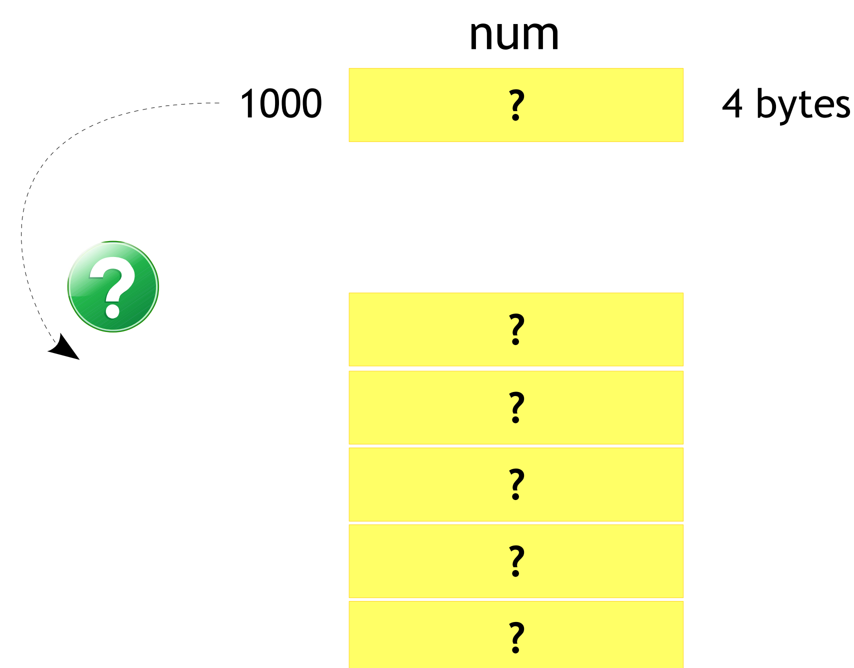
- Rule: “Pointer value of NULL or Zero = Null Addr = Null Pointer = Pointing to Nothing”

Advanced C

Pointers - Rule 6 in detail - NULL Pointer

Example

```
#include <stdio.h>

int main()
{
    int *num;

    return 0;
}
```



Where am I pointing to?

What does it Contain?

Can I read or write wherever I am pointing?

Advanced C

Pointers – Rule 6 in detail – NULL Pointer

- Is it pointing to the valid address?
- If yes can we read or write in the location where its pointing?
- If no what will happen if we access that location?
- So in summary where should we point to avoid all this questions if we don't have a valid address yet?
- The answer is Point to Nothing!!

Advanced C

Pointers – Rule 6 in detail – NULL Pointer

- Now what is Point to Nothing?
- A permitted location in the system will always give predictable result!
- It is possible that we are pointing to some memory location within our program limit, which might fail any time! Thus making it bit difficult to debug.
- An act of initializing pointers to 0 (generally, implementation dependent) at definition.
- 0??, Is it a value zero? So a pointer contain a value 0?
- Yes. On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system

- So by convention if a pointer is initialized to zero value, it is logically understood to be point to nothing.
- And now, in the pointer context, 0 is called as **NULL**
- So a pointer that is assigned NULL is called a Null Pointer which is **Pointing to Nothing**
- So dereferencing a NULL pointer is illegal and will always lead to segment violation, which is better than pointing to some unknown location and failing randomly!

Advanced C

Pointers – Rule 6 in detail – NULL Pointer

- Need for Pointing to 'Nothing'
  - Terminating Linked Lists
  - Indicating Failure by malloc, ...
- Solution
  - Need to reserve one valid value
  - Which valid value could be most useless?
  - In wake of OSes sitting from the start of memory, 0 is a good choice
  - As discussed in previous sides it is implementation dependent

Advanced C

Pointers - Rule 6 in detail - NULL Pointer

Example

```c
#include <stdio.h>

int main()
{
    int *num;

    num = NULL;

    return 0;
}
```

Example

```c
#include <stdio.h>

int main()
{
    int *num = NULL;

    return 0;
}
```

- A generic pointer which can point to data in memory
- The data type has to be mentioned while accessing the memory which has to be done by type casting

Example

```c
#include <stdio.h>

int main()
{
    void *vptr;

    return 0;
}
```

	vptr
2000	?

Advanced C

Pointers – Void Pointer - Size of void

- On gcc size of void is 1
- Hence pointer arithmetic can be performed on void pointer
- Its compiler dependent!

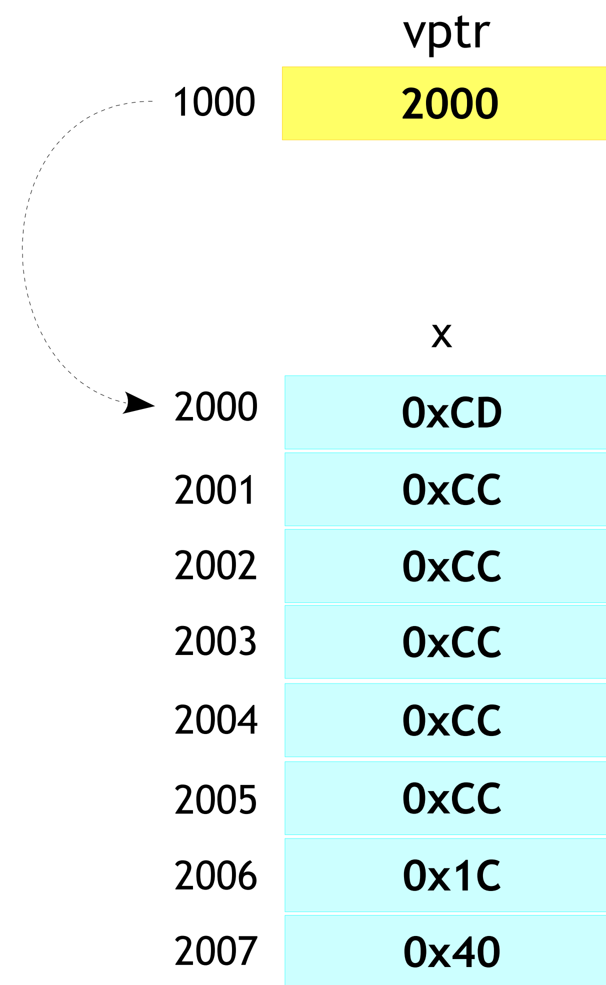
Note: To make standard compliant, compile using gcc -pedantic-errors

Advanced C

Pointers – Void Pointer

Example

```c
#include <stdio.h>

int main()
{
    double x = 7.2;
    void *vptr = &x;

    return 0;
}
```



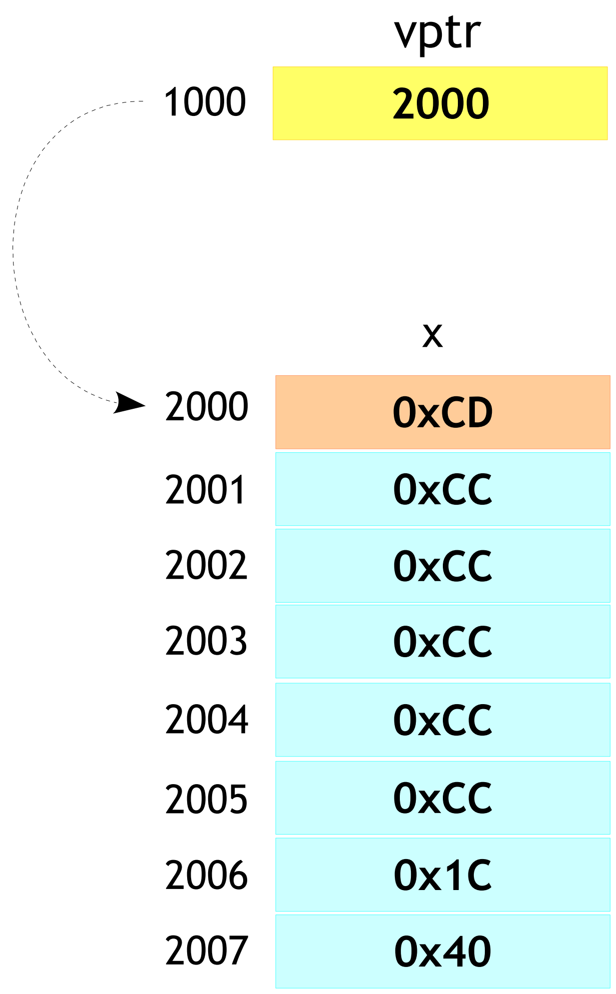
- vptr is a void pointer pointing to address of x which holds the data of type double
- These eights bytes are the legal region to the vptr
- We can access any byte(s) within this region by type casting

Advanced C

Pointers – Void Pointer

Example

```c
#include <stdio.h>

int main()
{
    double x = 7.2;
    void *vptr = &x;

    printf(“%hhx\n”, *(char *)vptr);
    printf(“%hhx\n”, *(char *)(vptr + 7));
    printf(“%hu\n”, *(short *)(vptr + 3));
    printf(“%x\n”, *(int *)(vptr + 0));

    return 0;
}
```

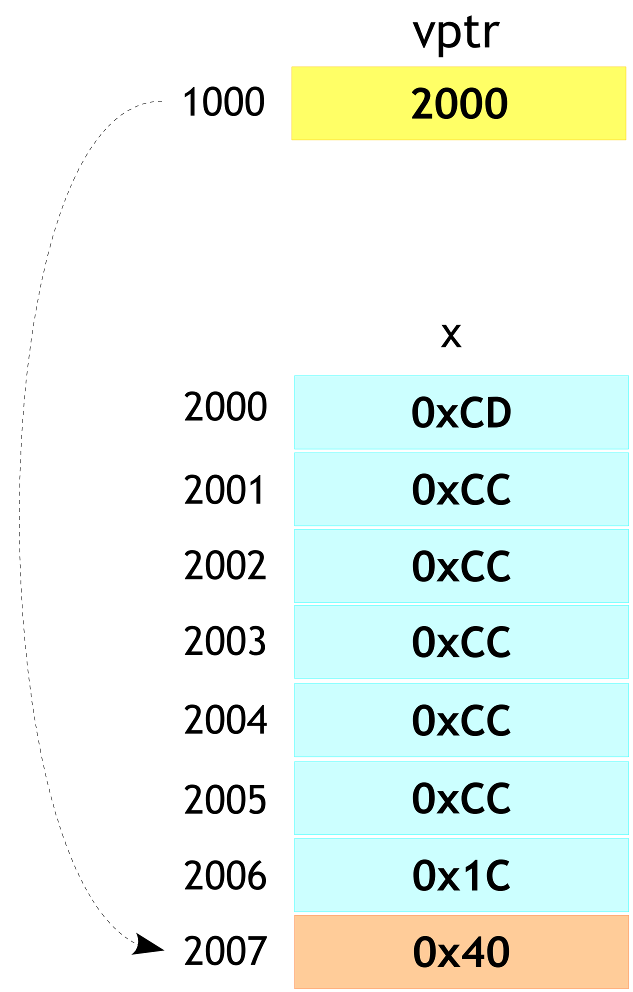


Advanced C

Pointers – Void Pointer

Example

```c
#include <stdio.h>

int main()
{
    double x = 7.2;
    void *vptr = &x;

    printf(“%hhx\n”, *(char *)vptr);
    printf(“%hhx\n”, *(char *)(vptr + 7);
    printf(“%hu\n”, *(short *)(vptr + 3);
    printf(“%x\n”, *(int *)(vptr + 0);

    return 0;
}
```



Advanced C

Pointers – Void Pointer

Example

```c
#include <stdio.h>

int main()
{
    double x = 7.2;
    void *vptr = &x;

    printf(“%hhx\n”, *(char *)vptr);
    printf(“%hhx\n”, *(char *)(vptr + 7);
    printf(“%hu\n”, *(short *)(vptr + 3);
    printf(“%x\n”, *(int *)(vptr + 0);

    return 0;
}
```

vptr

1000	2000

x

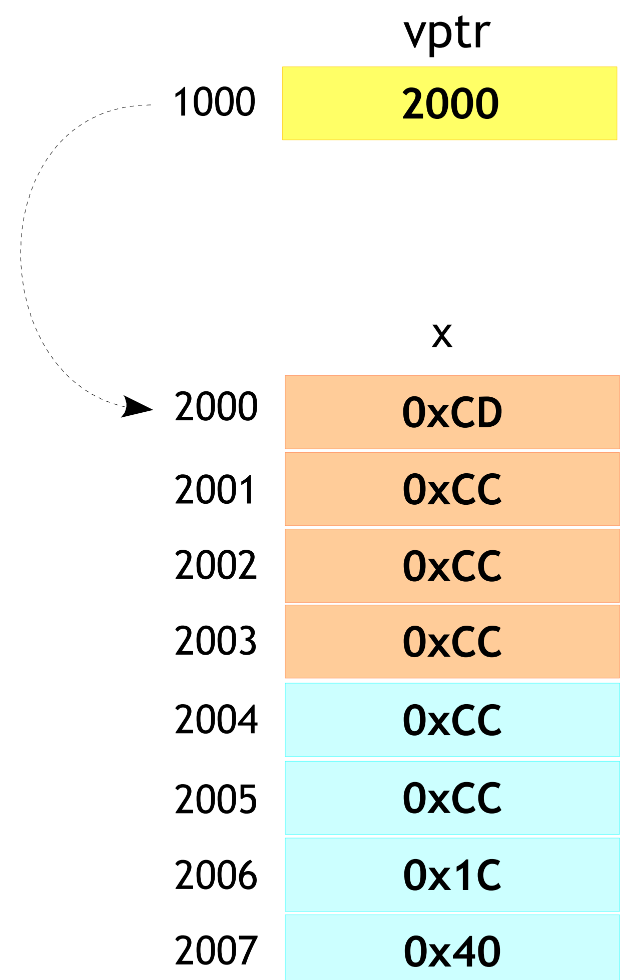
2000	0xCD
2001	0xCC
2002	0xCC
2003	0xCC
2004	0xCC
2005	0xCC
2006	0x1C
2007	0x40

Advanced C

Pointers – Void Pointer

Example

```
#include <stdio.h>

int main()
{
    double x = 7.2;
    void *vptr = &x;

    printf("%hhx\n", *(char *)vptr);
    printf("%hhx\n", *(char *)(vptr + 7));
    printf("%hu\n", *(short *)(vptr + 3));
    printf("%x\n", *(int *)(vptr + 0));

    return 0;
}
```



Advanced C

Pointers – Void Pointer

- A pointer with incomplete type
- Void pointer can't be dereferenced. You MUST use type cast operator (type) to dereference.
- DIY
  - W.A.P to swap any given data type

:GCC Extension:

6.22 Arithmetic on void- and Function-Pointers

In GNU C, addition and subtraction operations are supported on "pointers to void" and on "pointers to functions". This is done by treating the size of a void or of a function as 1.

A consequence of this is that sizeof is also allowed on void and on function types, and returns 1.

The option **-Wpointer-arith** requests a warning if these extensions are used

Advanced C

Pointers - The 7 Rules - Rule 7

- Rule: “Static Allocation vs Dynamic Allocation”

Example

```
#include <stdio.h>

int main()
{
    char array[5];

    return 0;
}
```

Example

```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    return 0;
}
```

- Named vs Unnamed Allocation = Named vs Unnamed Houses

Ok, House 1, I should go??? Oops

1 2 3 4 5 6 7 8

Ok, House 1, I should go that side ←

Advanced C

Pointers – Rule 7 in detail

- Managed by Compiler vs User
- Compiler
  - The compiler will allocate the required memory internally
  - This is done at the time of definition of variables
- User
  - The user has to allocate the memory whenever required and deallocate whenever required
  - This done by using malloc and free

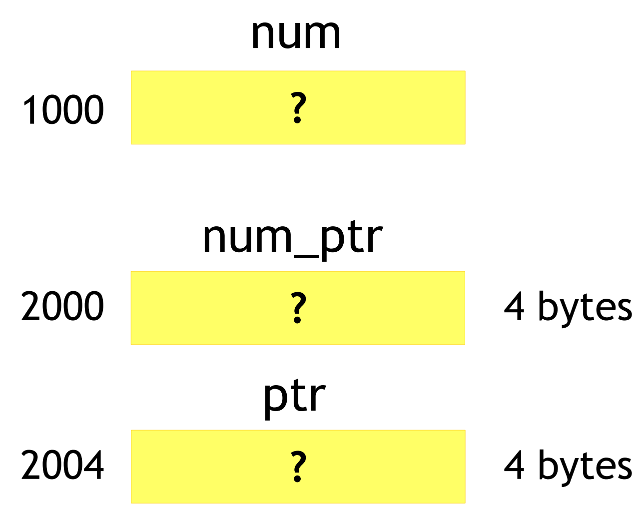
Advanced C

Pointers – Rule 7 in detail

- Static vs Dynamic

Example

```c
#include <stdio.h>

int main()
{
    int num, *num_ptr, *ptr;

    num_ptr = &num;

    ptr = malloc(1);

    return 0;
}
```



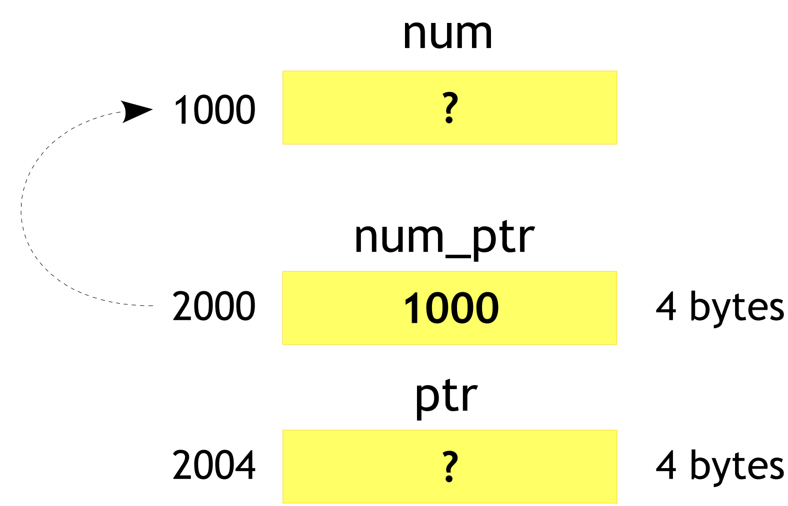
Advanced C

Pointers – Rule 7 in detail

- Static vs Dynamic

Example

```c
#include <stdio.h>

int main()
{
    int num, *num_ptr, *ptr;

    num_ptr = &num;

    ptr = malloc(1);

    return 0;
}
```



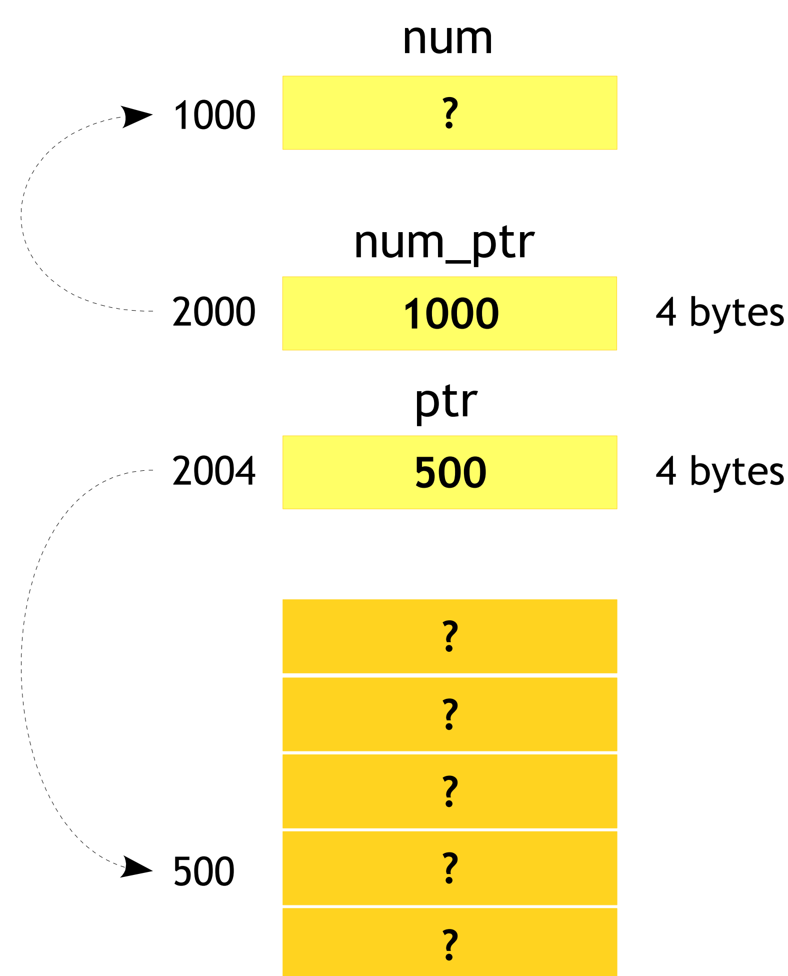
Advanced C

Pointers – Rule 7 in detail

- Static vs Dynamic

Example

```c
#include <stdio.h>

int main()
{
    int num, *num_ptr, *ptr;

    num_ptr = &num;

    ptr = malloc(1);

    return 0;
}
```



Advanced C

Pointers – Rule 7 in detail - Dynamic Allocation

- The need
  - You can decide size of the memory at run time
  - You can resize it whenever required
  - You can decide when to create and destroy it.

Advanced C

Pointers – Rule 7 – Dynamic Allocation - malloc

Prototype

```
void *malloc(size_t size);
```

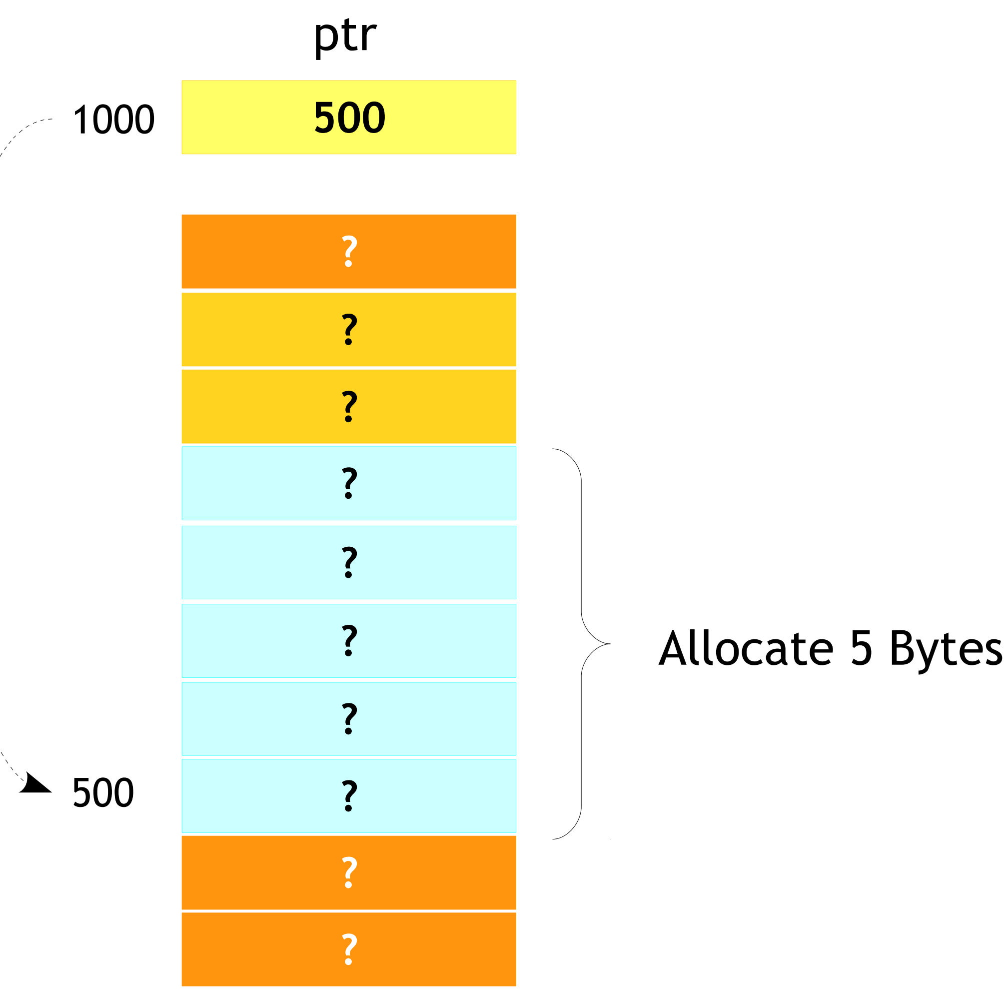
- Allocates the requested size of memory from the heap
- The size is in bytes
- Returns the pointer of the allocated memory on success, else returns NULL pointer

Advanced C

Pointers – Rule 7 – Dynamic Allocation - malloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    return 0;
}
```

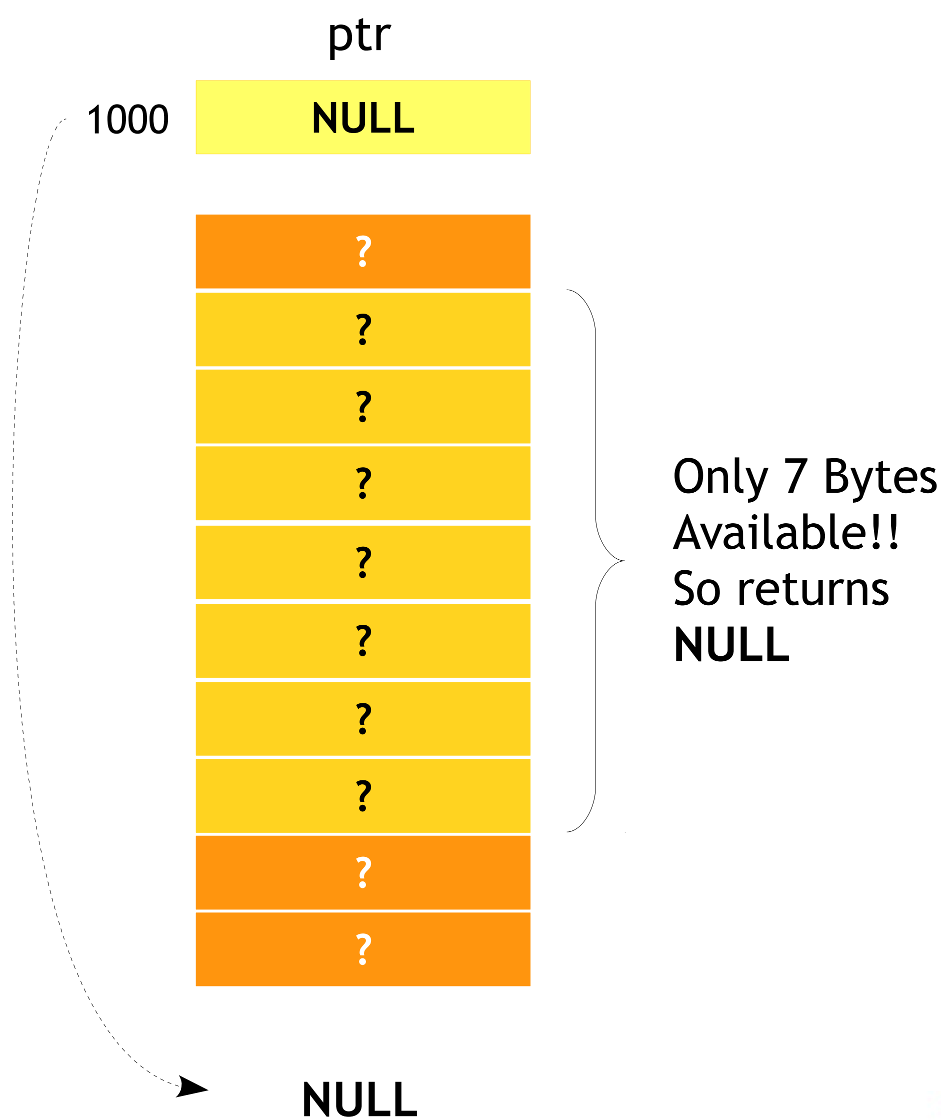


Advanced C

Pointers – Rule 7 – Dynamic Allocation - malloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(10);

    return 0;
}
```



Prototype

```
void *calloc(size_t nmemb, size_t size);
```

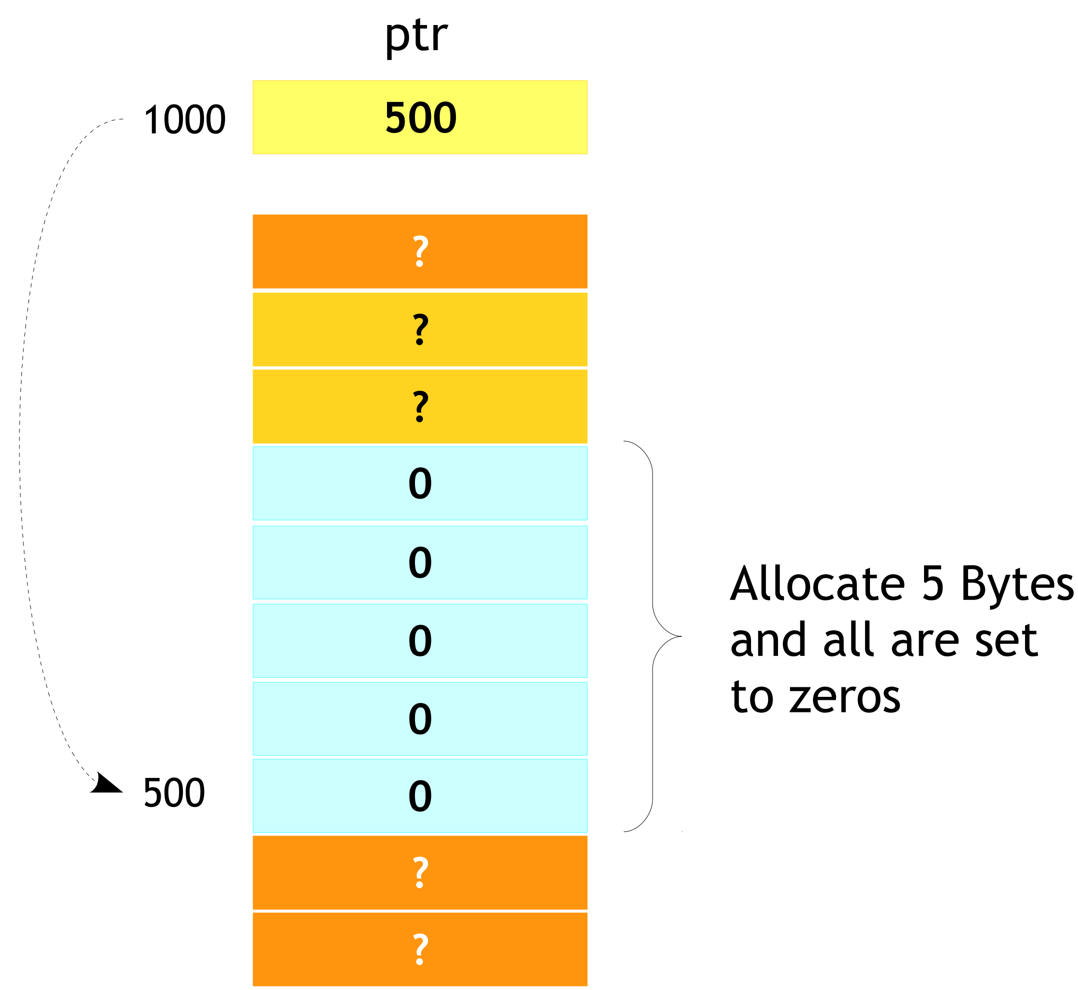
- Allocates memory blocks large enough to hold "n elements" of "size" bytes each, from the heap
- The allocated memory is set with 0's
- Returns the pointer of the allocated memory on success, else returns NULL pointer

Advanced C

Pointers – Rule 7 – Dynamic Allocation - calloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = calloc(5, 1);

    return 0;
}
```



Prototype

```
void *realloc(void *ptr, size_t size);
```

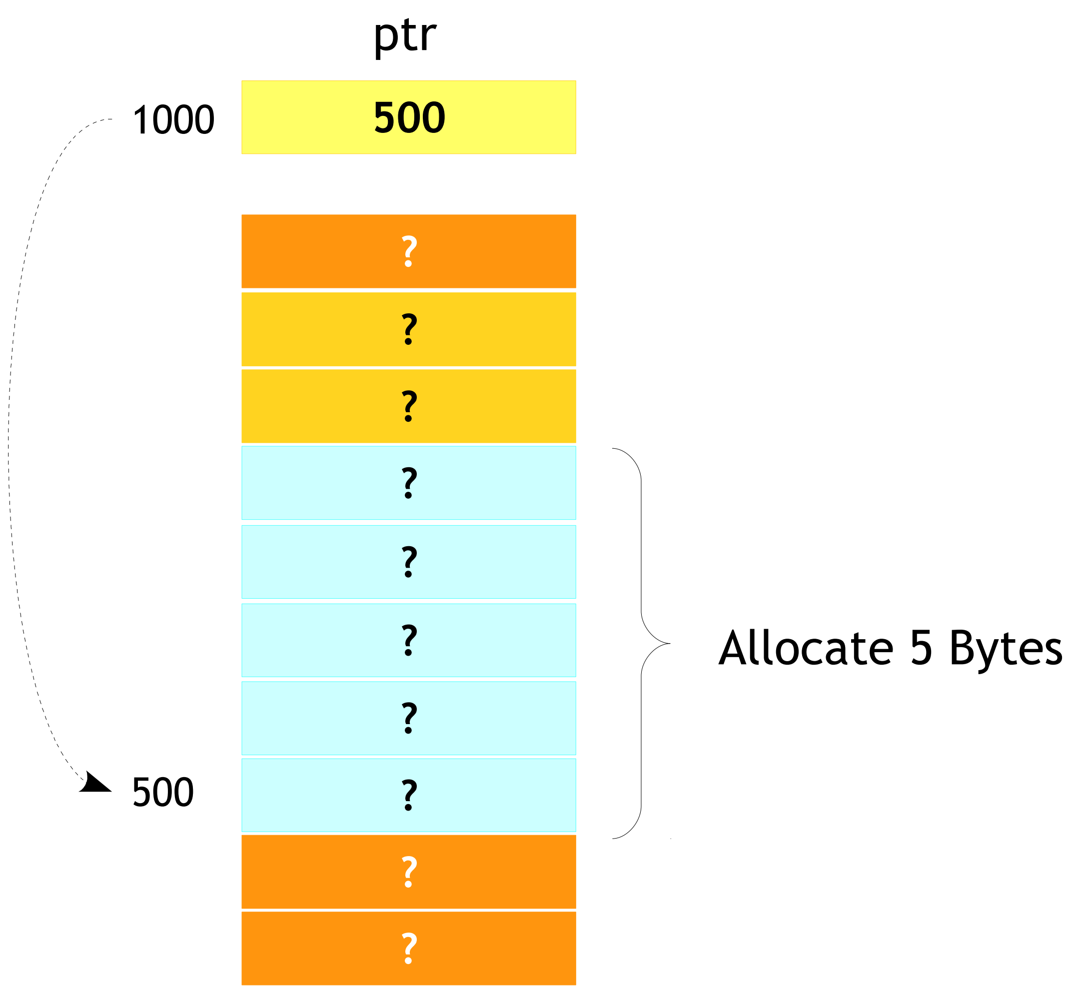
- Changes the size of the already allocated memory by malloc or calloc.
- Returns the pointer of the allocated memory on success, else returns NULL pointer

Advanced C

Pointers – Rule 7 – Dynamic Allocation - realloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    ptr = realloc(ptr, 7);
    ptr = realloc(ptr, 2);

    return 0;
}
```

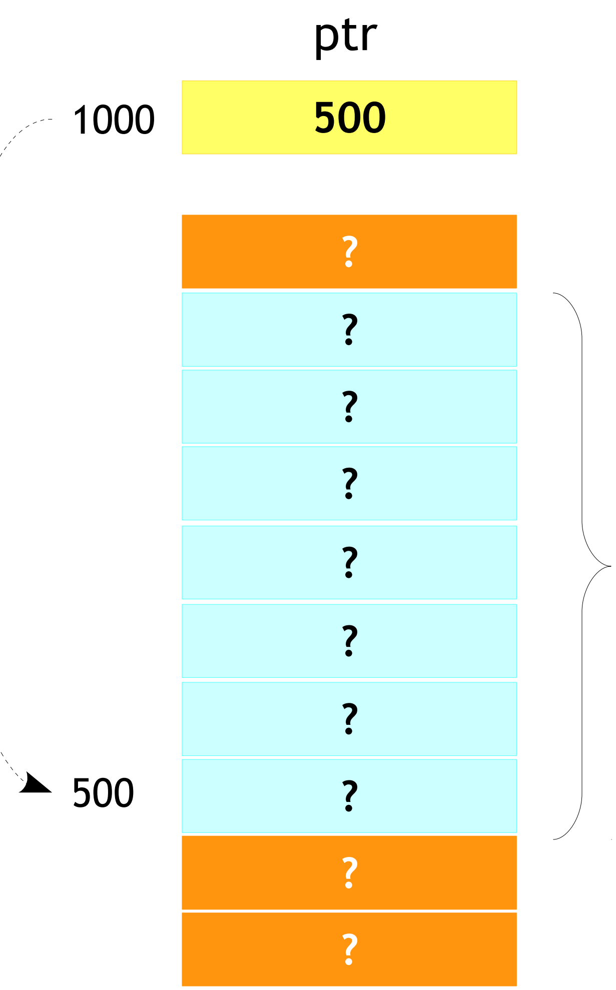


Advanced C

Pointers – Rule 7 – Dynamic Allocation - realloc

Example

```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    ptr = realloc(ptr, 7);
    ptr = realloc(ptr, 2);

    return 0;
}
```



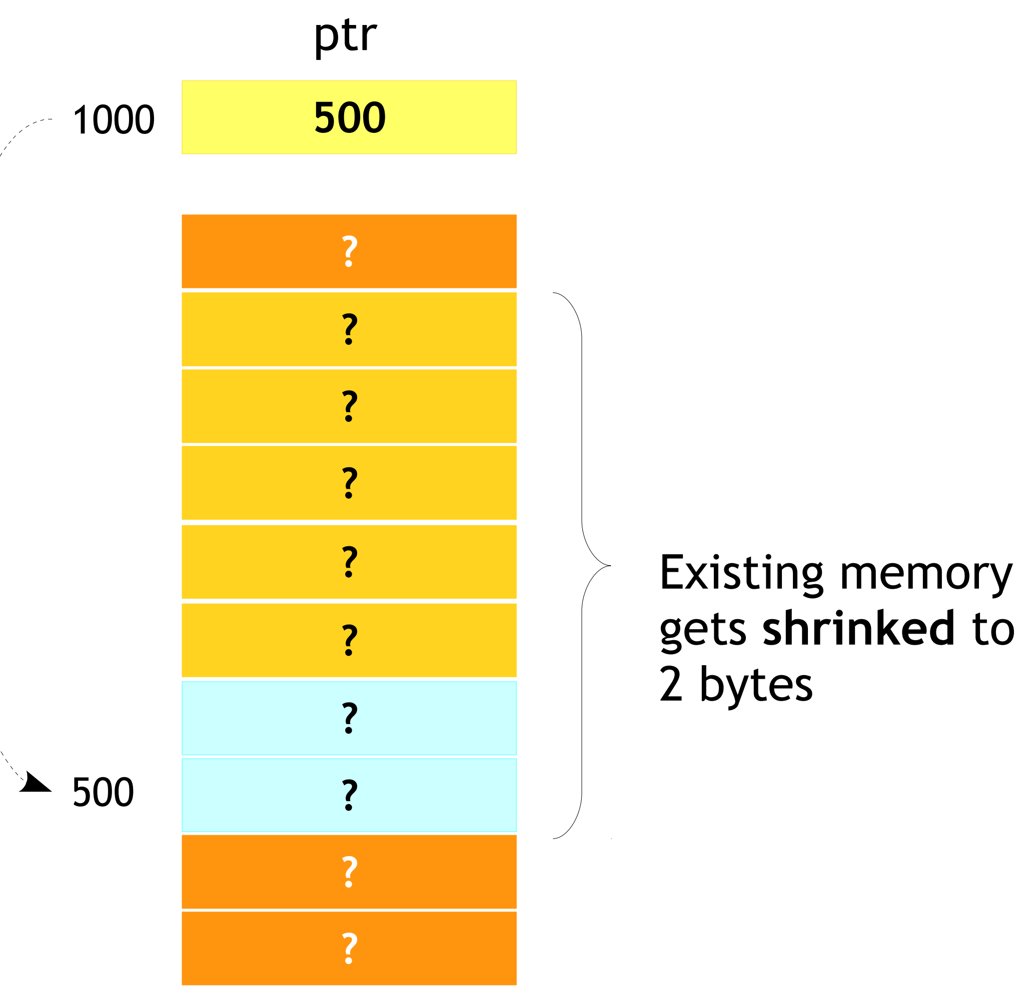
Existing memory gets **extended** to 7 bytes

Advanced C

Pointers – Rule 7 – Dynamic Allocation - realloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    ptr = realloc(ptr, 7);
    ptr = realloc(ptr, 2);

    return 0;
}
```



Advanced C

Pointers – Rule 7 – Dynamic Allocation - realloc

- Points to be noted
  - Reallocating existing memory will be like deallocating the allocated memory
  - If the requested chunk of memory cannot be extended in the existing block, it would allocate in a new free block starting from different memory!
  - If new memory block is allocated then old memory block is automatically freed by realloc function

Pointers – Rule 7 – Dynamic Deallocation - free

Prototype

```
void free(void *ptr);
```

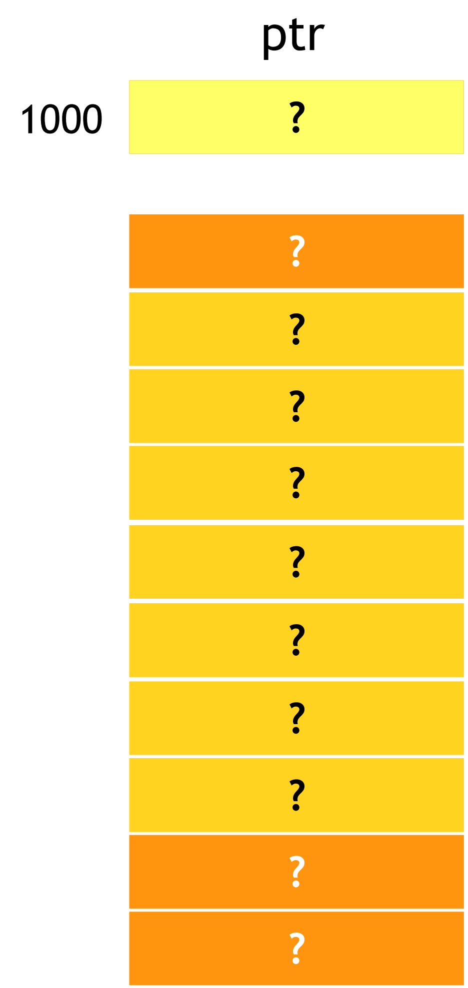
- Frees the allocated memory, which must have been returned by a previous call to malloc(), calloc() or realloc()
- Freeing an already freed block or any other block, would lead to undefined behaviour
- Freeing NULL pointer has no effect.
- If free() is called with invalid argument, might collapse the memory management mechanism
- If free is not called after dynamic memory allocation, will lead to memory leak

Advanced C

Pointers – Rule 7 – Dynamic Deallocation - free

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;
    int iter;

    ptr = malloc(5);

    for (iter = 0; iter < 5; iter++)
    {
        ptr[iter] = 'A' + iter;
    }

    free(ptr);

    return 0;
}
```

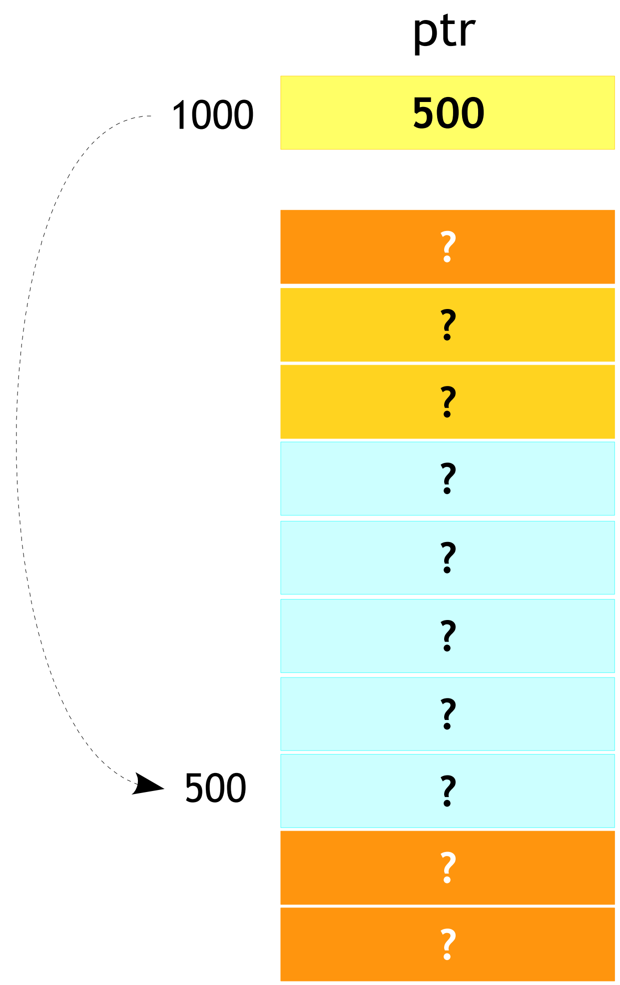


Advanced C

Pointers – Rule 7 – Dynamic Deallocation - free

Example

```
#include <stdio.h>

int main()
{
    char *ptr;
    int iter;

    ptr = malloc(5);

    for (iter = 0; iter < 5; iter++)
    {
        ptr[iter] = 'A' + iter;
    }

    free(ptr);

    return 0;
}
```

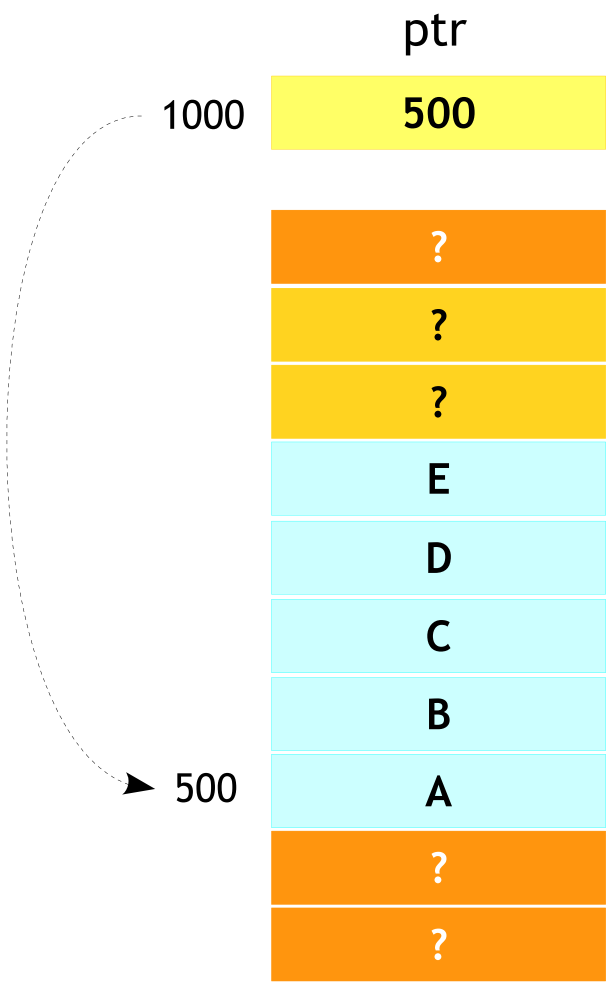


Advanced C

Pointers – Rule 7 – Dynamic Deallocation - free

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;
    int iter;

    ptr = malloc(5);

    for (iter = 0; iter < 5; iter++)
    {
        ptr[iter] = 'A' + iter;
    }

    free(ptr);

    return 0;
}
```

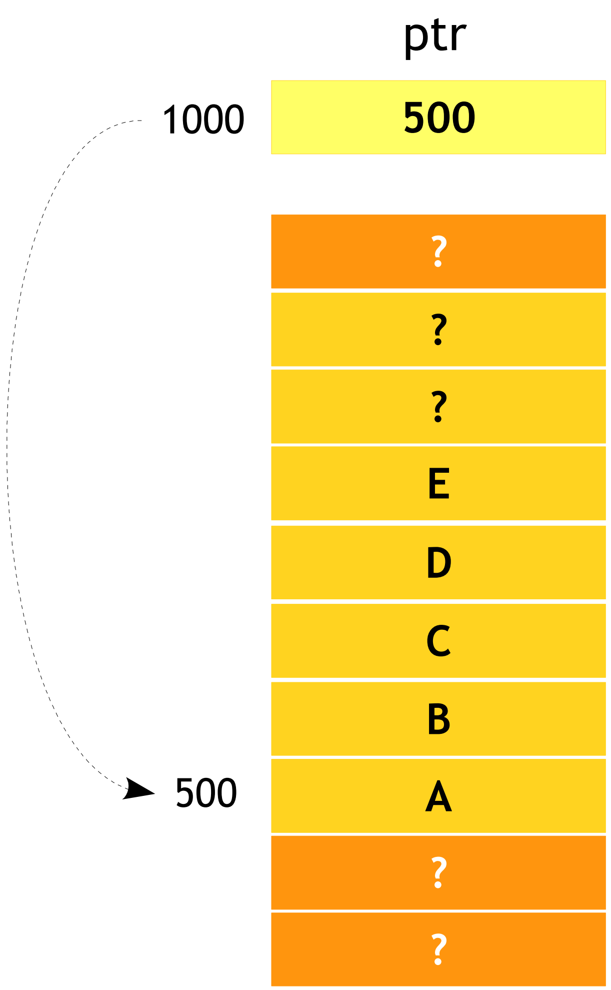


Advanced C

Pointers – Rule 7 – Dynamic Deallocation - free

Example

```
#include <stdio.h>

int main()
{
    char *ptr;
    int iter;

    ptr = malloc(5);

    for (iter = 0; iter < 5; iter++)
    {
        ptr[iter] = 'A' + iter;
    }

    free(ptr);

    return 0;
}
```



EMERTXE

- Points to be noted
  - Free releases the allocated block, but the pointer would still be pointing to the same block!!, So accessing the freed block will have undefined behaviour.
  - This type of pointer which are pointing to freed locations are called as **Dangling Pointers**
  - Doesn't clear the memory after freeing

- Implement my_strdup function

Advanced C

Pointers – Const Pointer

Example

```
#include <stdio.h>

int main()
{
    int const *num = NULL;

    return 0;
}
```

The location, its pointing to is constant

Example

```
#include <stdio.h>

int main()
{
    int * const num = NULL;

    return 0;
}
```

The pointer is constant

Example

```
#include <stdio.h>

int main()
{
    const int * const num = NULL;

    return 0;
}
```

Both constants

Example

```
#include <stdio.h>

int main()
{
    const int num = 100;
    int *iptr = &num;

    printf("Number is %d\n", *iptr);

    *iptr = 200;

    printf("Number is %d\n", num);

    return 0;
}
```

Advanced C

Pointers – Const Pointer

Example

```
#include <stdio.h>

int main()
{
    int num = 100;
    const int *iptr = &num;

    printf("Number is %d\n", num);

    num = 200;

    printf("Number is %d\n", *iptr);

    return 0;
}
```

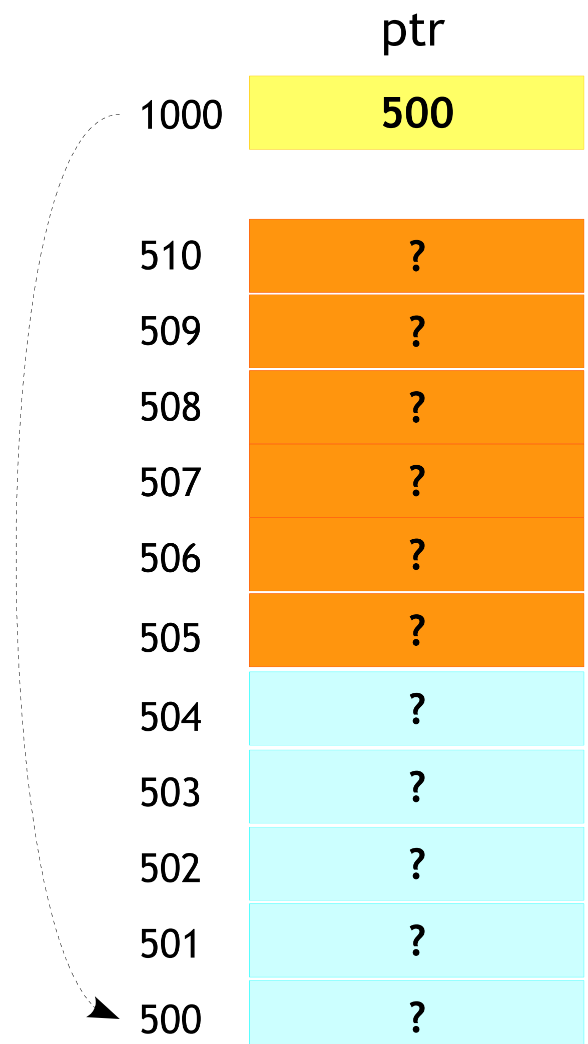
Advanced C

Pointers – Do's and Dont's

Example

```c
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

    ptr = ptr + 10; /* Yes */
    ptr = ptr - 10; /* Yes */

    return 0;
}
```

- malloc(5) allocates a block of 5 bytes as shown



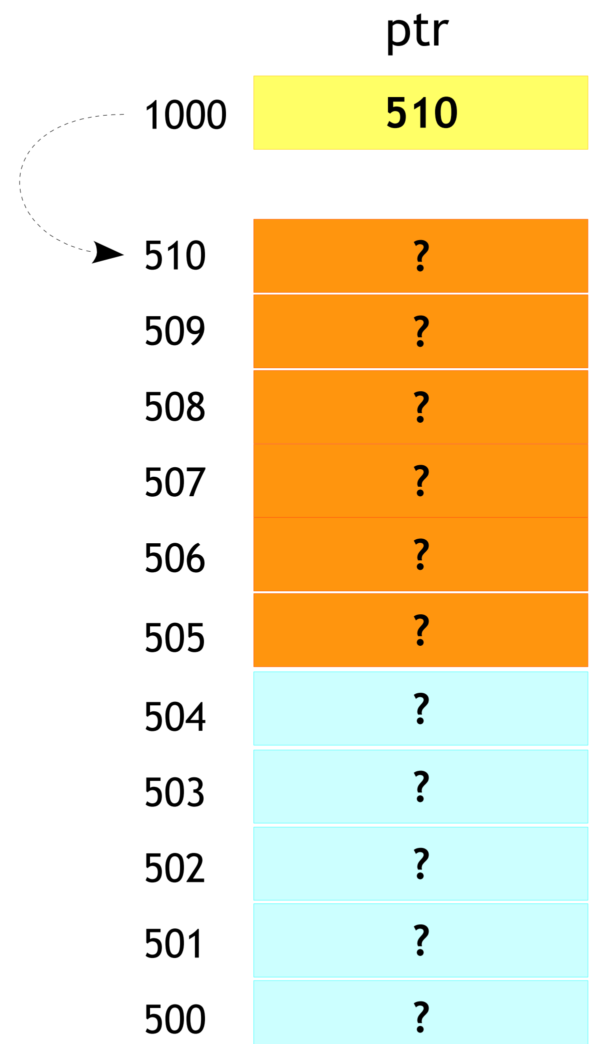
Advanced C

Pointers – Do's and Dont's

Example

```c
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

    ptr = ptr + 10; /* Yes */
    ptr = ptr - 10; /* Yes */

    return 0;
}
```

- Adding 10 to ptr we will advance 10 bytes from the base address which is illegal but no issue in compilation!!



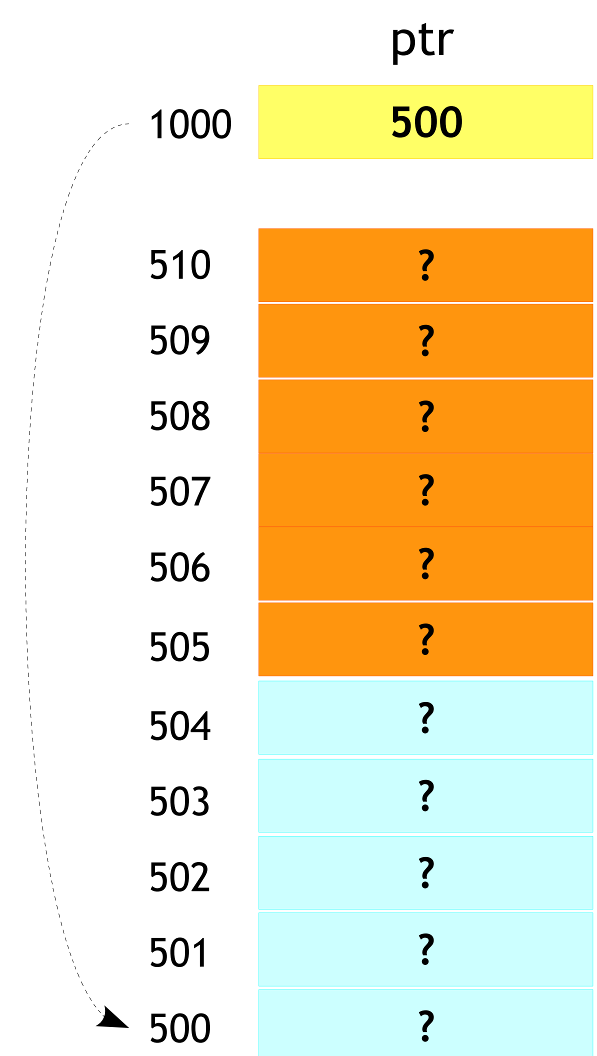
Advanced C

Pointers – Do's and Dont's

Example

```c
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

    ptr = ptr + 10; /* Yes */
    ptr = ptr - 10; /* Yes */

    return 0;
}
```

- Subtracting 10 from ptr we will retract 10 bytes to the base address which is perfectly fine



EMERTXE

Example

```
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

    ptr = ptr * 1; /* No */
    ptr = ptr / 1; /* No */

    return 0;
}
```

- All these operation on the ptr will be illegal and would lead to compiler error!!
- In fact most of the binary operator would lead to compilation error

Advanced C

Pointers – Do's and Dont's

Example

```c
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

    ptr = ptr + ptr; /* No */
    ptr = ptr * ptr; /* No */
    ptr = ptr / ptr; /* No */

    return 0;
}
```

- All these operation on the ptr will be illegal and would lead to compiler error!!
- In fact most of the binary operator would lead to compilation error

Advanced C

Pointers – Do's and Dont's

Example

```c
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

    ptr = ptr - ptr;

    return 0;
}
```

- What is happening here!?
- Well the value of ptr would be 0, which is nothing but NULL (Most of the architectures) so it is perfectly fine
- The compiler would compile the code with a warning though

Advanced C

Pointers – Pitfalls – Segmentation Fault

- A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed.

Example

```
#include <stdio.h>

int main()
{
    int num = 0;

    printf("Enter the number\n");
    scanf("%d", num);

    return 0;
}
```

Example

```
#include <stdio.h>

int main()
{
    int *num = 0;

    printf("The number is\n", *num);

    return 0;
}
```

EMERTXE

- A dangling pointer is something which does not point to a valid location any more.

Example

```
#include <stdio.h>

int main()
{
    int *num_ptr;

    num_ptr = malloc(4);
    free(num_ptr);

    *num_ptr = 100;

    return 0;
}
```

Example

```
#include <stdio.h>

int *foo()
{
    int num_ptr;

    return &num_ptr;
}

int main()
{
    int *num_ptr;

    num_ptr = foo();

    return 0;
}
```

Advanced C

Pointers – Pitfalls – Wild Pointer

- An uninitialized pointer pointing to a invalid location can be called as an wild pointer.

Example

```
#include <stdio.h>

int main()
{
    int *num_ptr_1; /* Wild Pointer */
    static int *num_ptr_2; / Not a wild pointer */


    return 0;
}
```

Advanced C

Pointers – Pitfall - Memory Leak

- Improper usage of the memory allocation will lead to memory leaks
- Failing to deallocating memory which is no longer needed is one of most common issue.
- Can exhaust available system memory as an application runs longer.

Advanced C

Pointers – Pitfall - Memory Leak

Example

```
#include <stdio.h>

int main()
{
    int *num_array, sum = 0, no_of_elements, iter;

    while (1)
    {
        printf("Enter the number of elements: \n");
        scanf("%d", &no_of_elements);
        num_array = malloc(no_of_elements);

        sum = 0;
        for (iter = 0; iter < no_of_elements; iter++)
        {
            scanf("%d", &num_array[iter]);
            sum += num_array[iter];
        }

        printf("The sum of array elements are %d\n", sum);
        /* Forgot to free!! */
    }
    return 0;
}
```

- A bus error is a fault raised by hardware, notifying an operating system (OS) that, a process is trying to access memory that the CPU cannot physically address: an invalid address for the address bus, hence the name.

Example

```
#include <stdio.h>

int main()
{
    char array[sizeof(int) + 1];
    int *ptr1, *ptr2;

    ptr1 = &array[0];
    ptr2 = &array[1];

    scanf("%d %d", ptr1, ptr2);

    return 0;
}
```