Pointers, References, and all that fun in C/C++! ☺

Yogesh Virkar

Department of Computer Science
University of Colorado, Boulder.

Motivation

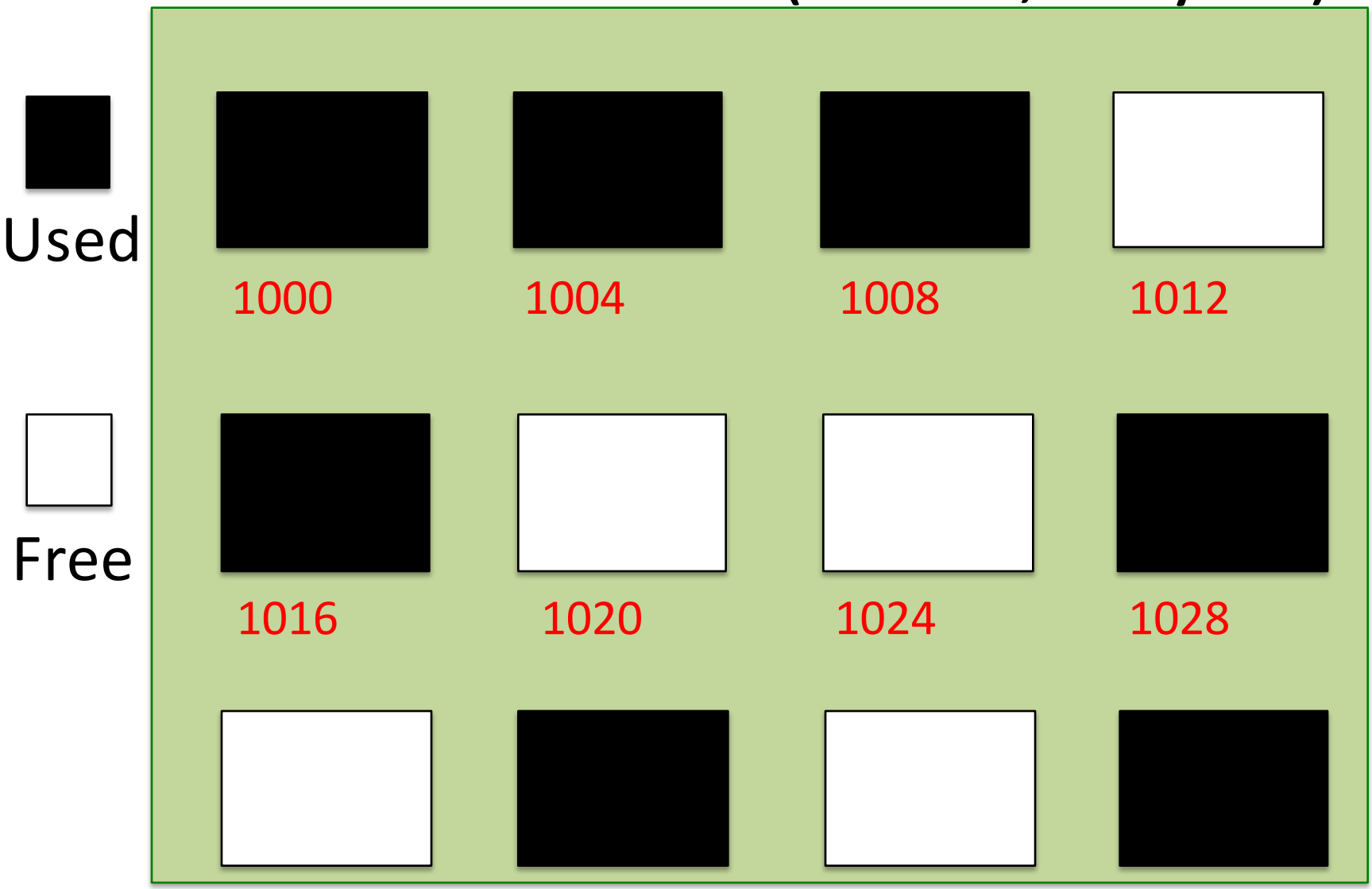
- A pointer variable stores the address of any given variable.
  - Knowing the address of a variable helps when we need to update its value in a function call.
- However, there are many subtleties of this simple concept that are hard to understand if we neglect some aspects of computer systems and architecture.
- In this presentation, we use memory (RAM) depictions to elucidate what pointers and references mean.
- Note:
  - Any variable in a computer system holds some numeric value that is a set of ones and zeros (in binary representation).
  - The way we store or interpret this numeric value defines whether the variable is an int, float, double, char, or even a pointer.

PART 0

1) What is a pointer?

2) Why do pointer variables have type?

Memory: Addresses are shown in red below each block (of size, 4 bytes).



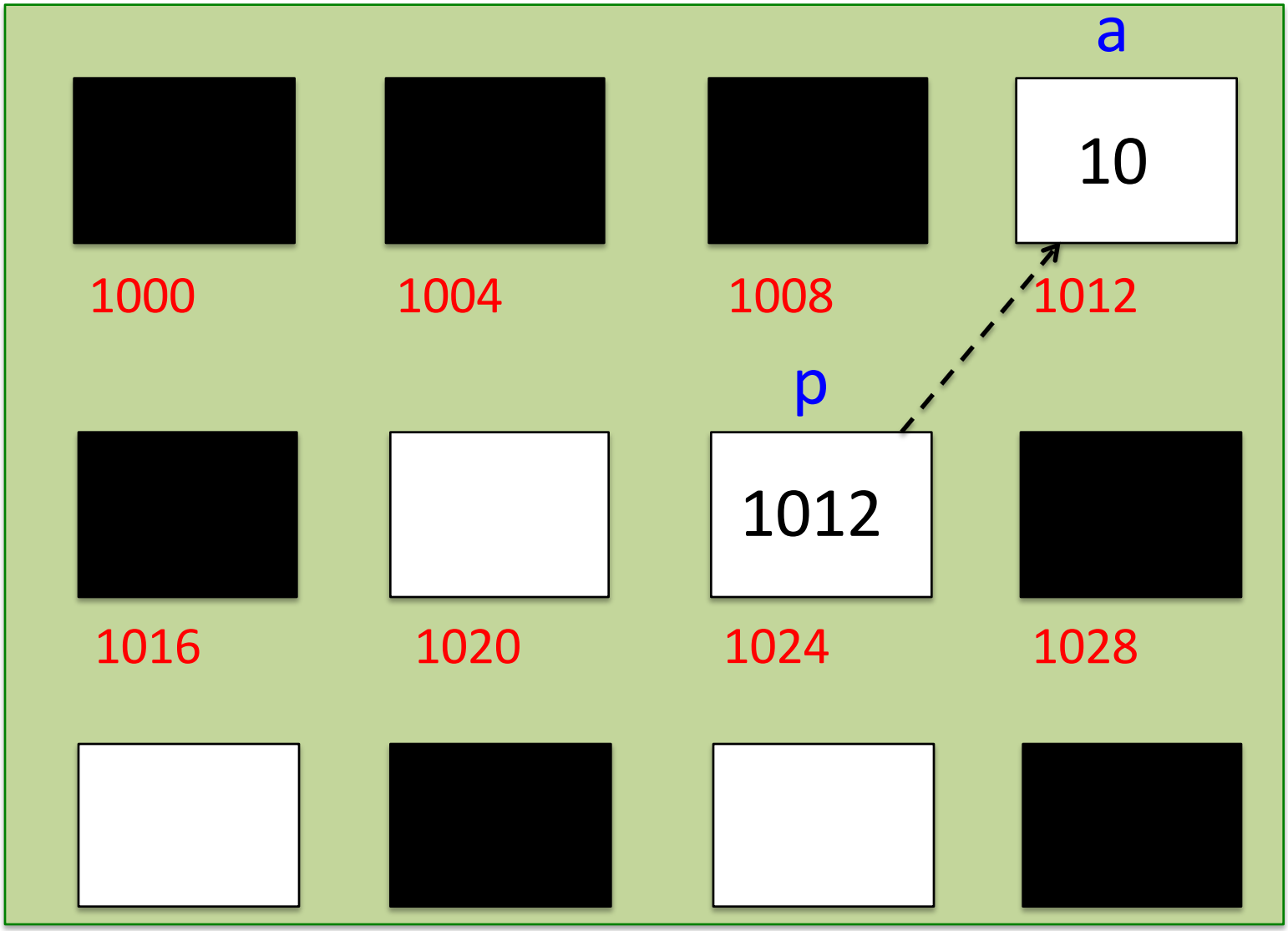
What is a pointer variable?

- int a = 10;
- int* p = &a;

- p is a pointer variable.
  - stores the address of a.

- & ➔ "address of" operator
  - Note: Cannot have address of a constant literal.
  - int* p = &10; // is meaningless.
  - int* p = &(&a); // also meaningless for same reason!

Pointer Example



What is a pointer variable? (2)

- int a = 10;
- int* p = &a;

- Thus pointers hold addresses.

- printf("\n %x \n", p);
- // This prints the address of p as ---
  - 0xFC109E21

- Usually address values are viewed in hexadecimal representation (%x stands for hexadecimal).
  - For simplicity, we view them as integers in the memory diagrams

Source: xkcd


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

Why do pointers have types?

- For a 32-bit system, a memory address is 32 bit or 4 byte long.

- If address of any variable (whether it is an int, or a float or a double or a char or a struct) is 4-byte long
  - Then why do pointers have type?
    - E.g. int* p or double* p.

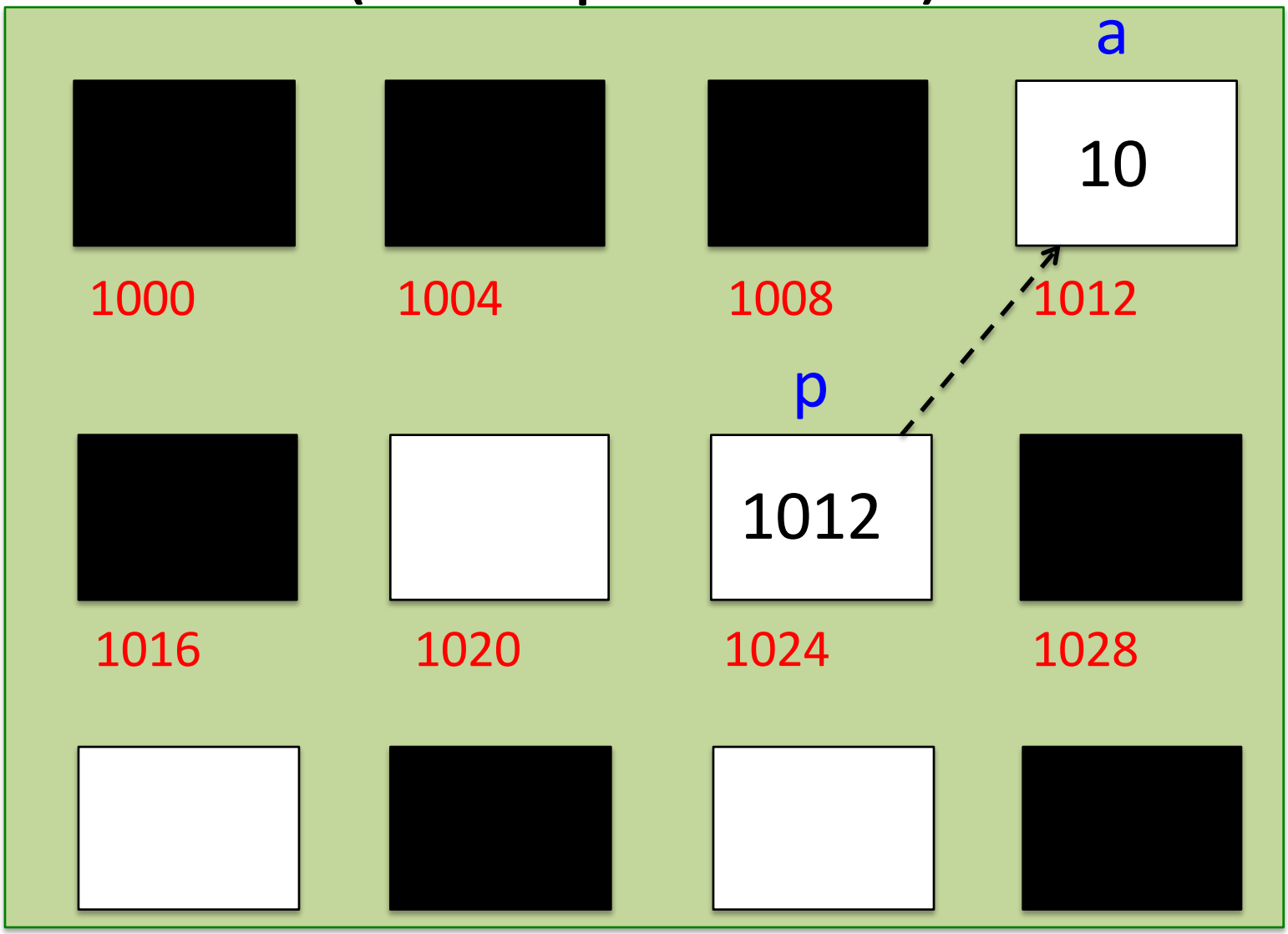
Why do pointers have types? (2)

- int a = 10;
- int* p = &a;
- // sizeof(int) is 4 bytes

- Dereferencing a pointer variable,
  - printf("\n %d \n", *p); // Prints 10.

- * is the dereference operator.
  - Gives us the content at the location given by 'p'.

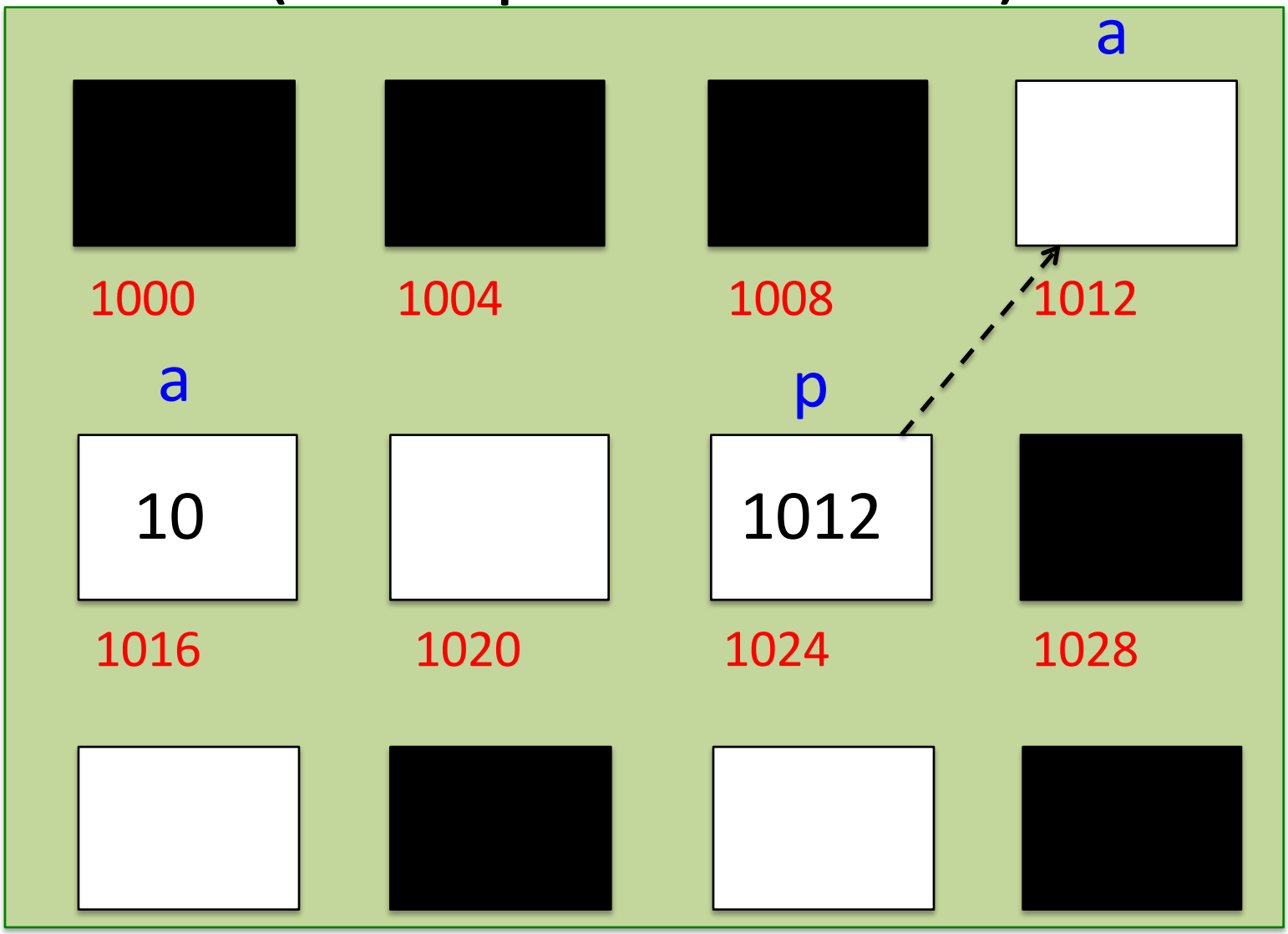
p gives us 4 bytes from address 1012 (since p is a int)



Why do pointers have types? (2)

- double a = 10;
- double* p = &a;
- // sizeof(double) is 8 bytes

p gives us 8 bytes from address 1012 (since p is a double)



Why do pointers have types? (3)

- Starting from the address given by the pointer variable, the dereference operator, i.e. *, needs to know how many bytes to read
- Hence pointers have types.

- Note:
- sizeof(double*) is the same as sizeof(int*) and is the same as the sizeof(struct ABC*).
- All are 4 byte on a 32-bit system.

PART 1

1) What is the difference between pass by value, pass by reference and pass by pointer?

2) What are double and n pointer variables?

Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Value example.

- int fun1 (int x) {
  return 2*x;
  }

- void main () {
  int a = 10;
  a = fun1(a);
  }

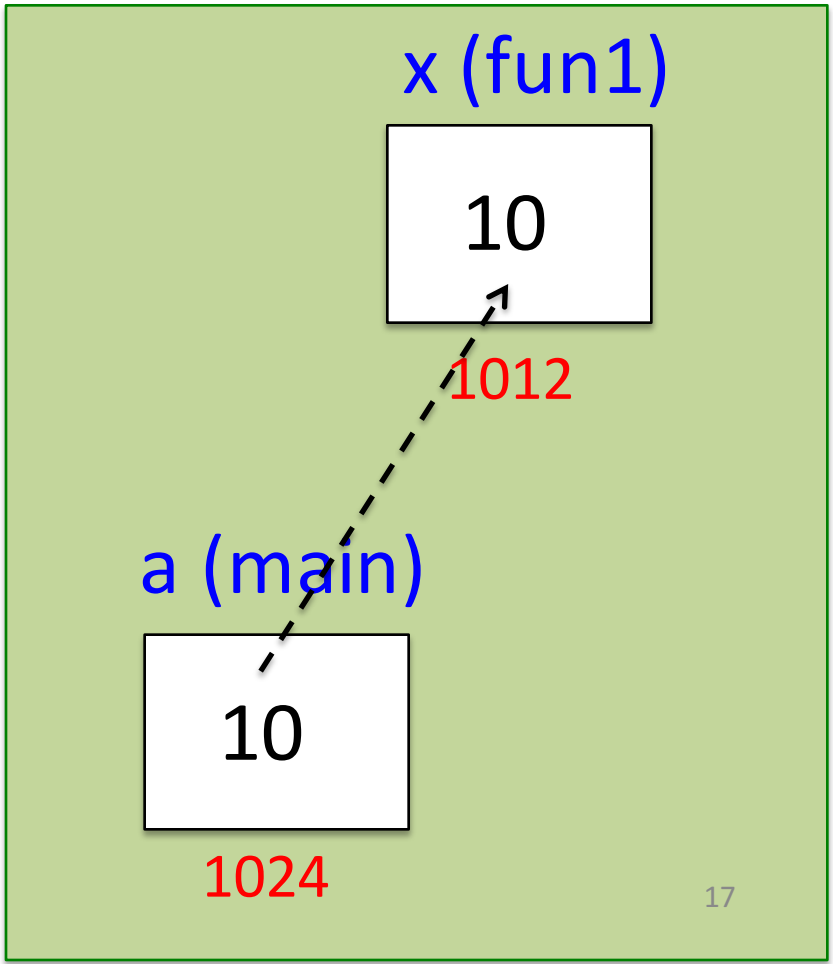
Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Value example.

```
int fun1 (int x) {
return 2*x;
}
```

```
void main () {
int a = 10;
a = fun1(a);
}
```

RAM: Inside fun1()

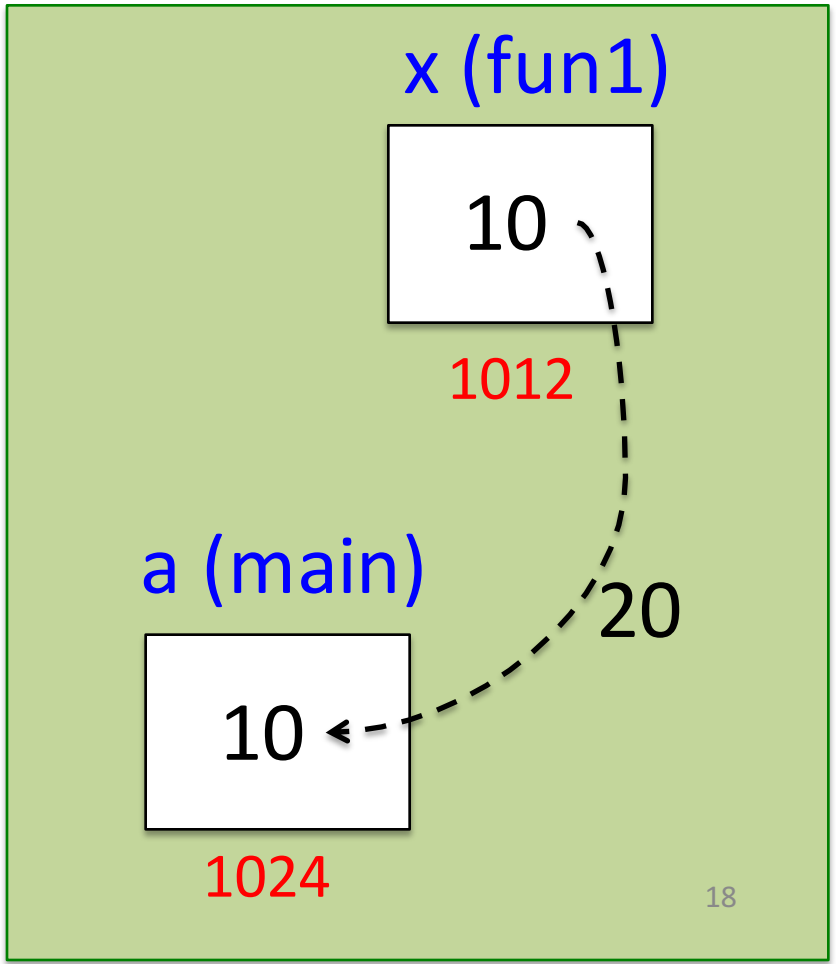


Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Value example.

- int fun1 (int x) {
  return 2*x;
  }

- void main () {
  int a = 10;
  a = fun1(a);
  }



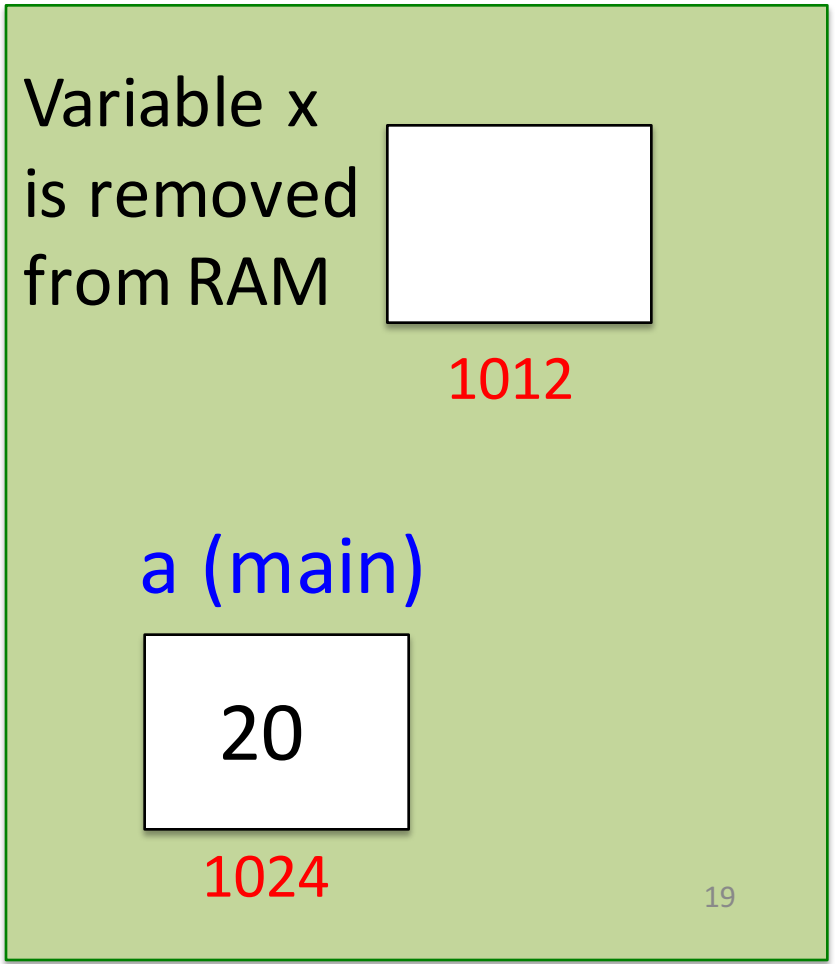
Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Value example.

- int fun1 (int x) {
  return 2*x;
  }

- void main () {
  int a = 10;
  a = fun1(a);
  }

RAM: Inside main()



Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Reference example (aliasing)

- int fun2 (int& x) {
  x = x*2;
  }

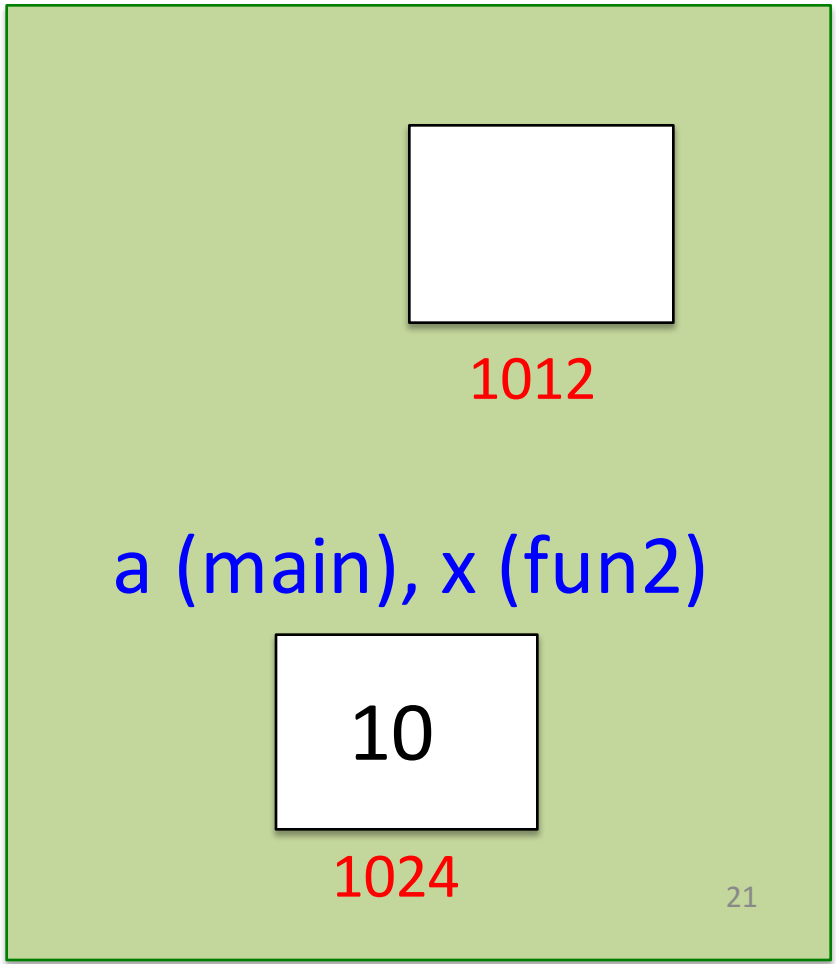
- void main () {
  int a = 10;
  fun2(a);
  }

Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Reference example (aliasing)

```
int fun2 (int& x) {
x = x*2;
}
```

-
```
void main () {
int a = 10;
fun2(a);
}
```

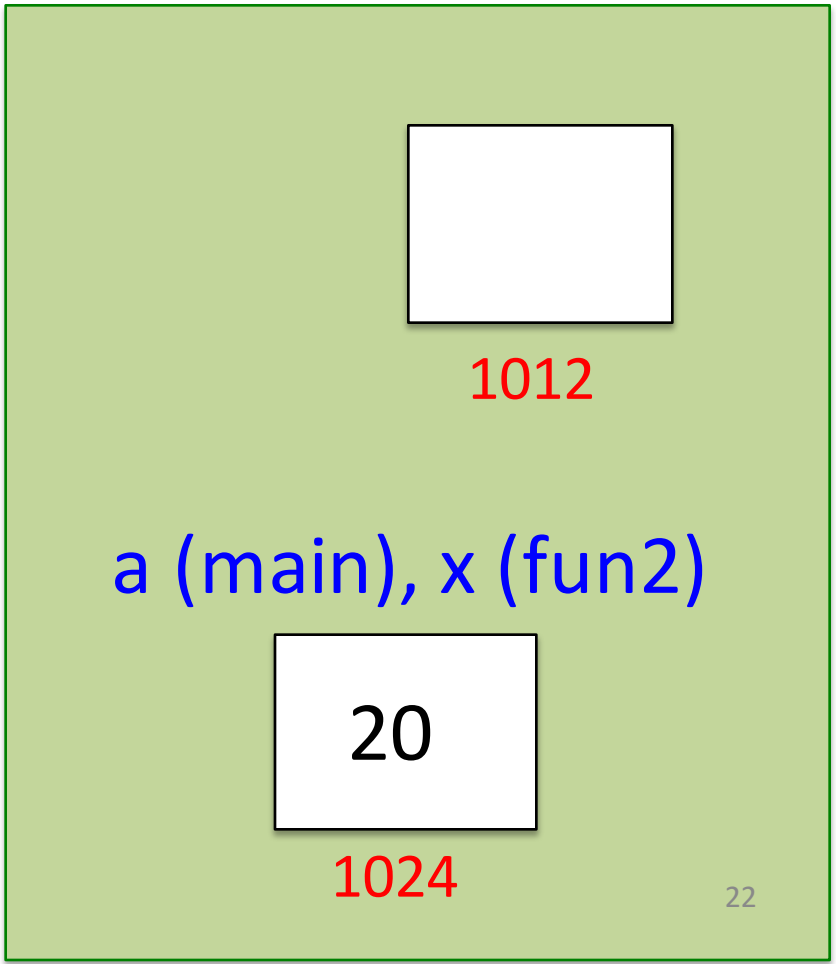


Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Reference example (aliasing)

- int fun2 (int& x) {
  x = x*2;
  }

- void main () {
  int a = 10;
  fun2(a);
  }



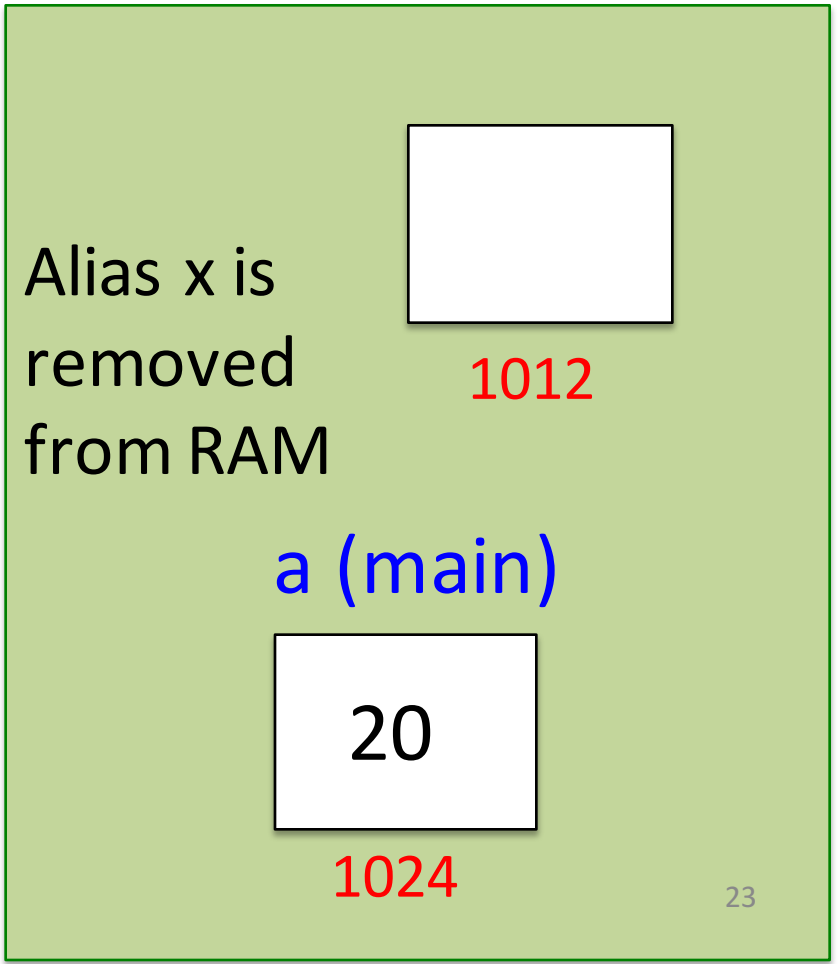
Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Reference example (aliasing)

- int fun2 (int& x) {
  x = x*2;
  }

- void main () {
  int a = 10;
  fun2(a);
  }

RAM: Inside main()



Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Pointer example

- int fun3 (int* x) {
  *x = *x * 2;
  }

- void main () {
  int a = 10;
  fun3(&a);
  }

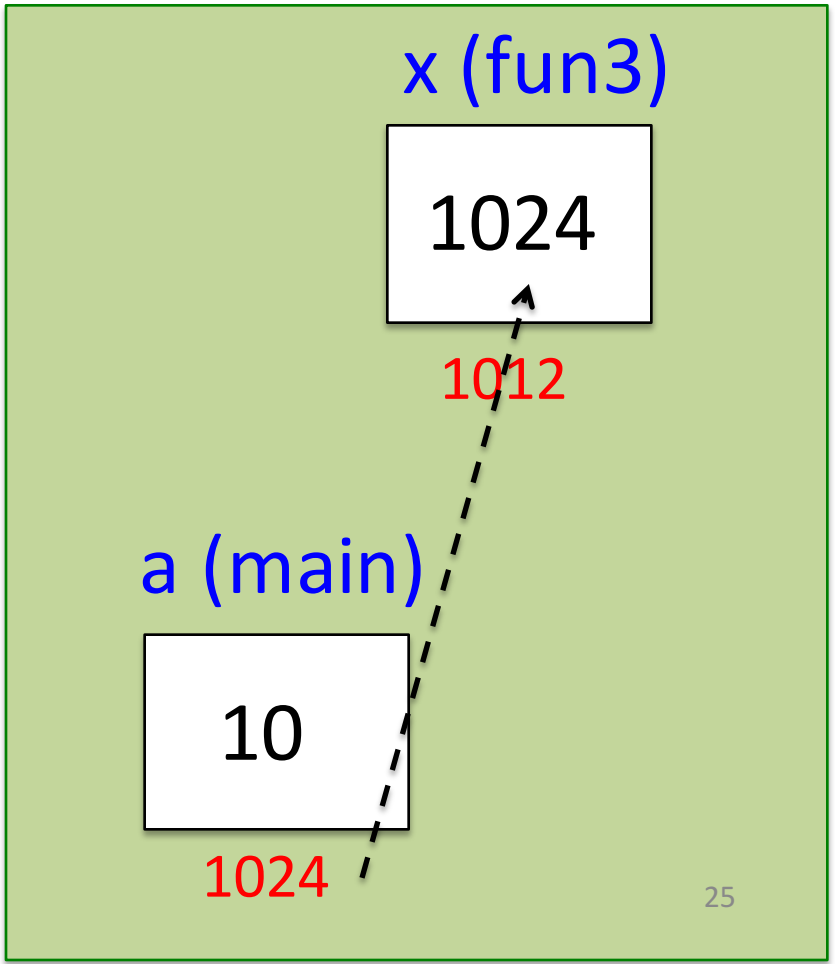
Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Pointer example

```
int fun3 (int* x) {
*x = *x * 2;
}
```

-
```
void main () {
int a = 10;
fun3(&a);
}
```

RAM: Inside fun3()



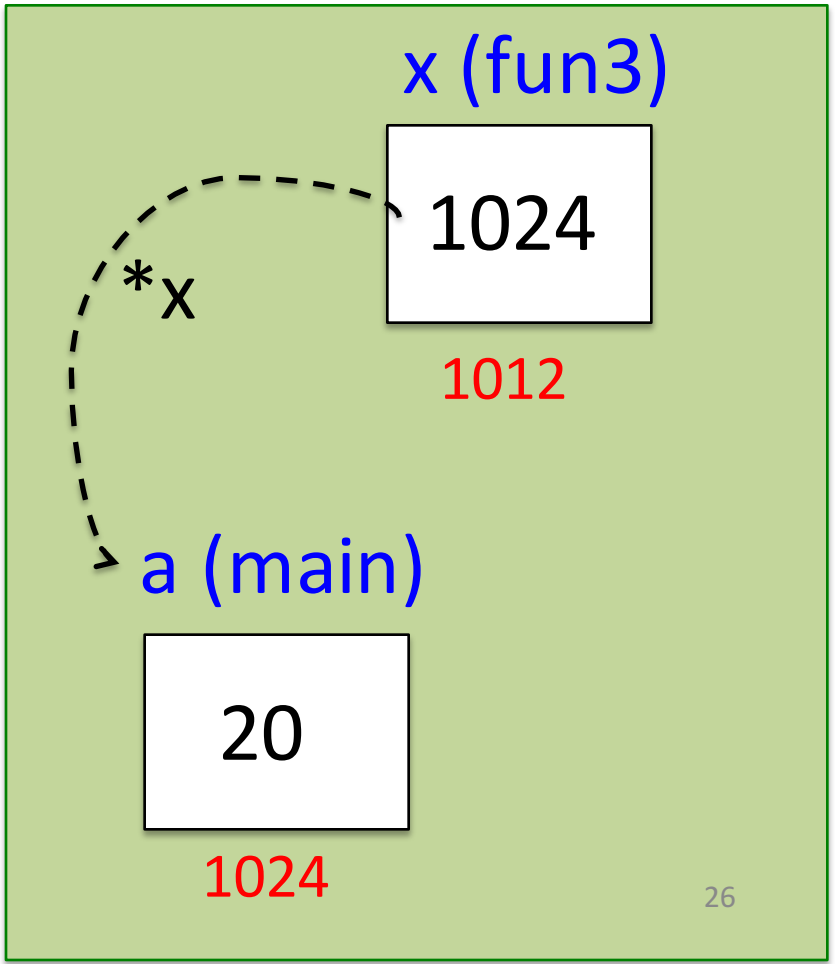
Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Pointer example

- int fun3 (int* x) {
  *x = *x * 2;
  }

- void main () {
  int a = 10;
  fun3(&a);
  }

RAM: Inside fun3()



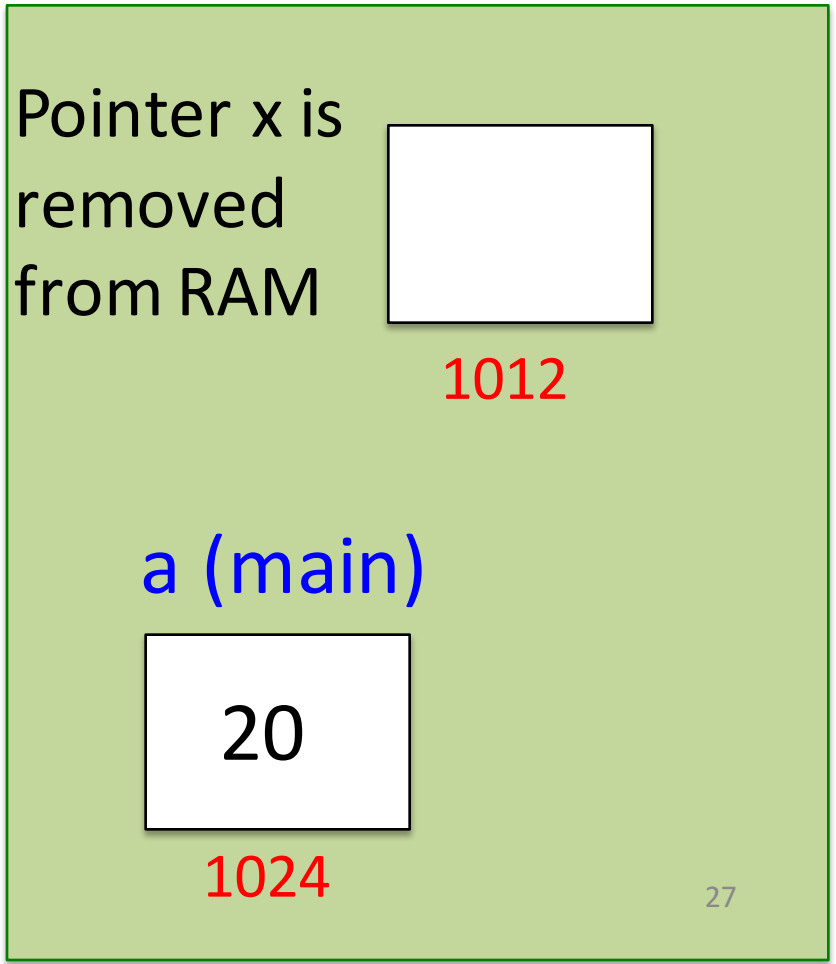
Pass by Value/Reference/Pointer: A dumb value doubling function

- Pass by Pointer example

- int fun3 (int* x) {
  *x = *x * 2;
  }

- void main () {
  int a = 10;
  fun3(&a);
  }

RAM: Inside fun3()



Some take away points

- If you want to change the value of any variable using a function call, you have three options:
  - Pass by value and then have the function return a value.
    - Advantage: Simplicity of understanding.
    - Drawback: You can return only one value. We cannot change more than one variable using this approach.
  - The function creates an alias of the given variable and changes it directly (pass by reference).
    - Advantage: We can change more than one variable in one function. We save memory. **Hence arrays by default are passed by reference!**
    - Drawback: Can be confusing.
  - The function takes the address of the variable it needs to change (pass by pointer).
    - Advantage: We can change more than one variable in one function. Is perhaps less confusing than pass by reference?
    - Drawback: We take a little more memory to hold the pointer variable and dereferencing takes time. Is perhaps the most confusing of all three? `(**Depends on what you are comfortable with!**)

Exercise 1: Can you draw the memory depiction for the following function?

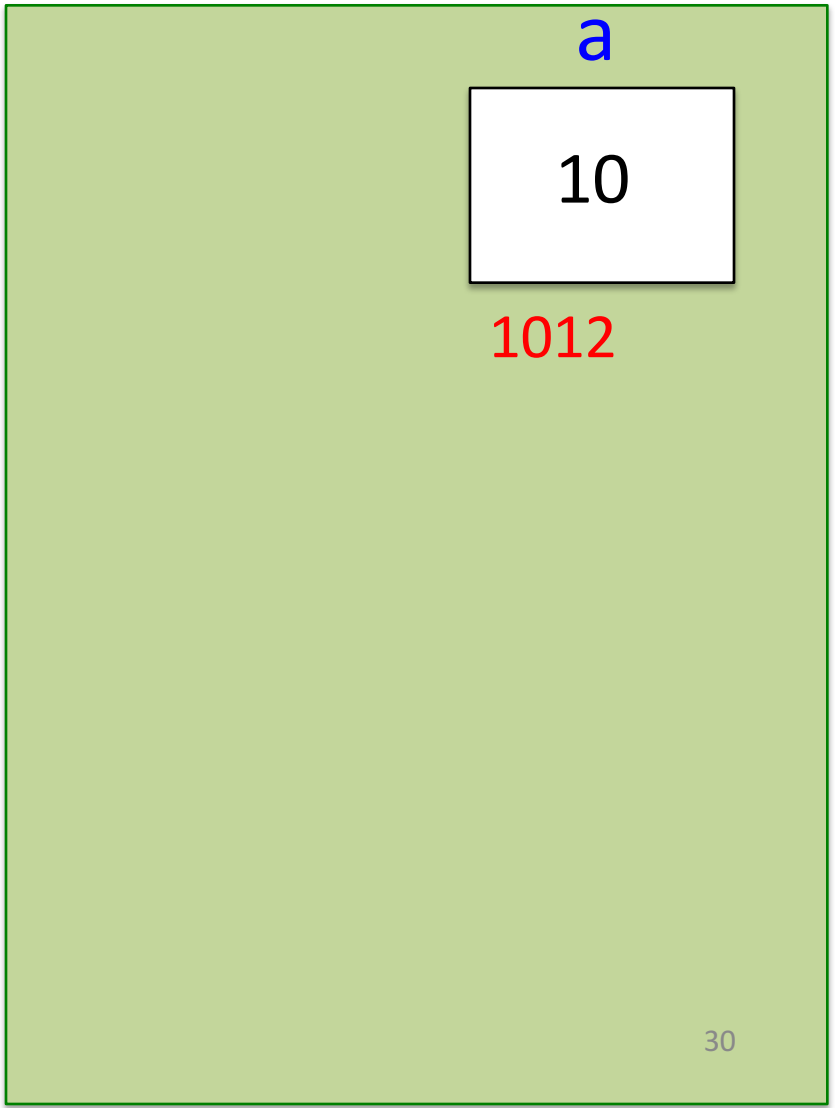
- ```
  void fun4 (int*& q) {
  q = new int[3];
  }
  ```

- ```
  void main () {
  int a = 10;
  int* p = &a;
  fun4(p);
  }
  ```

- Tips:
  - First think about the variables in the main function. Draw them.
  - Next, think about what int*& could mean.
  - Google the new command of c++ and understand what that line does.
  - Answers to all exercises are after the final slide.

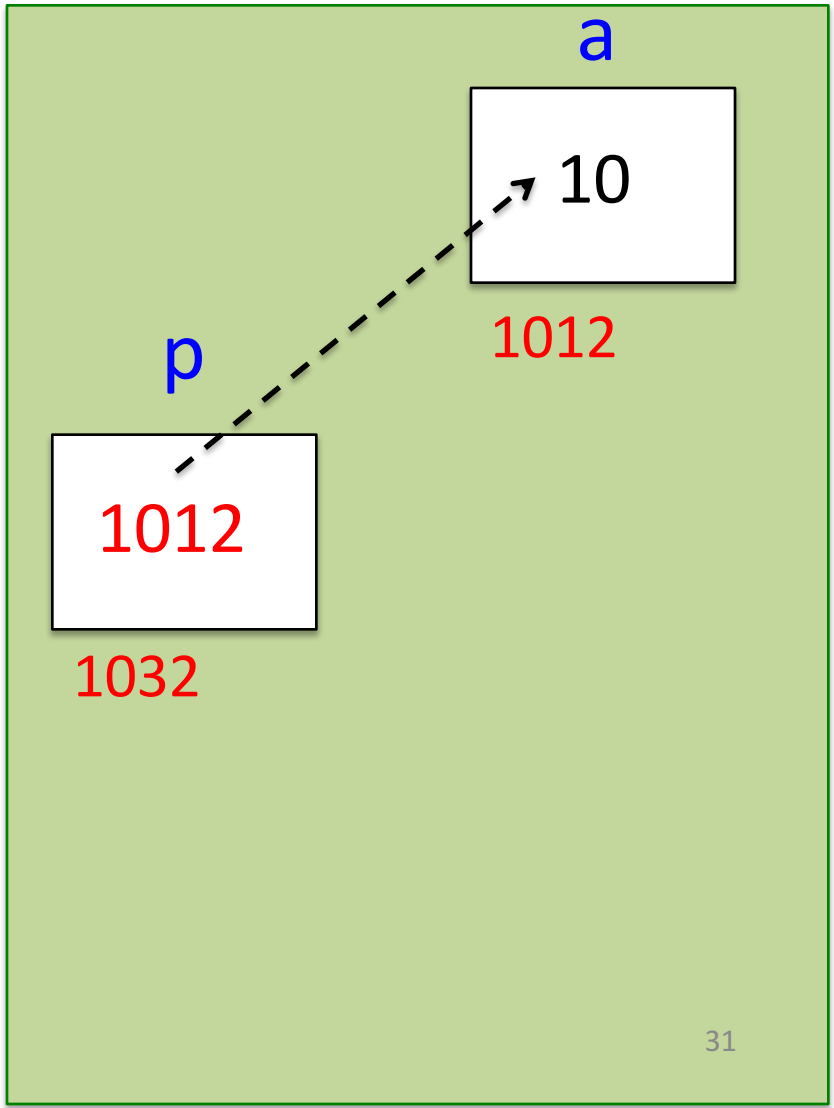
Single, Double and n pointer

- int a = 10;



Single, Double and n pointer

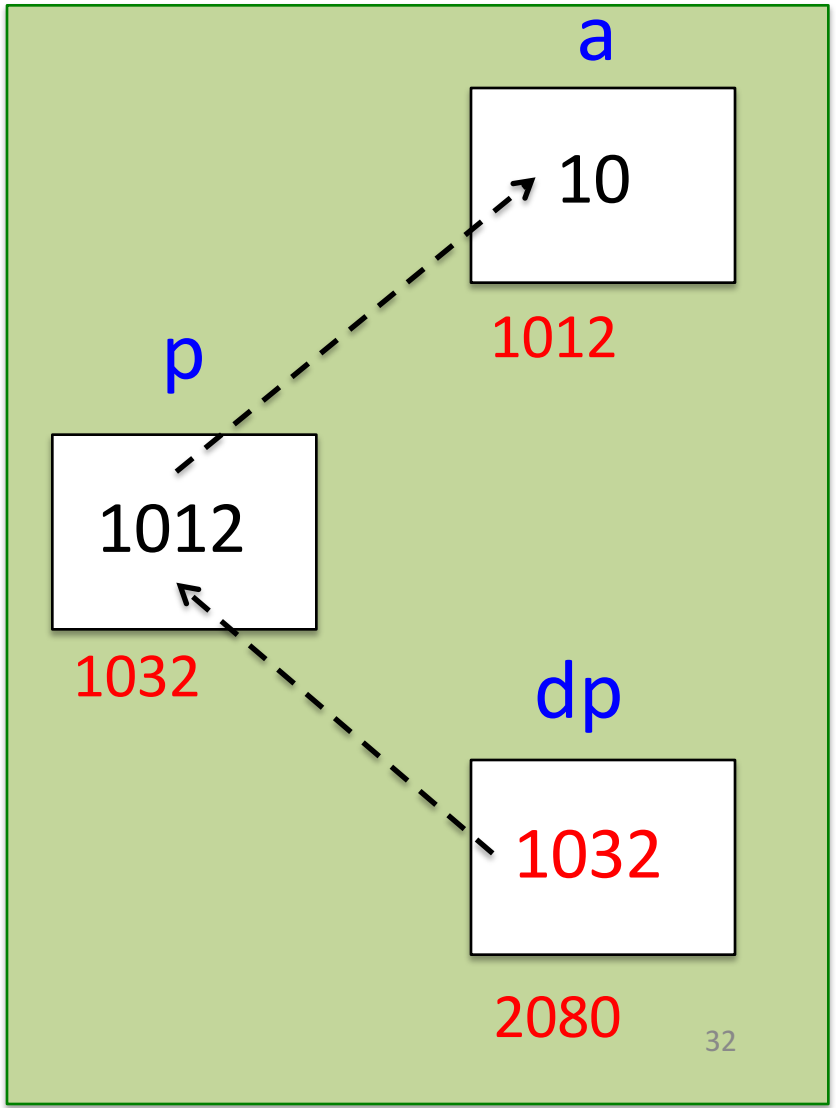
- int a = 10;
- int* p = &a;



Single, Double and n pointer

- int a = 10;
- int* p = &a;
- int** dp = &p;

- A double pointer stores the address of a single pointer



Single, Double and n pointer

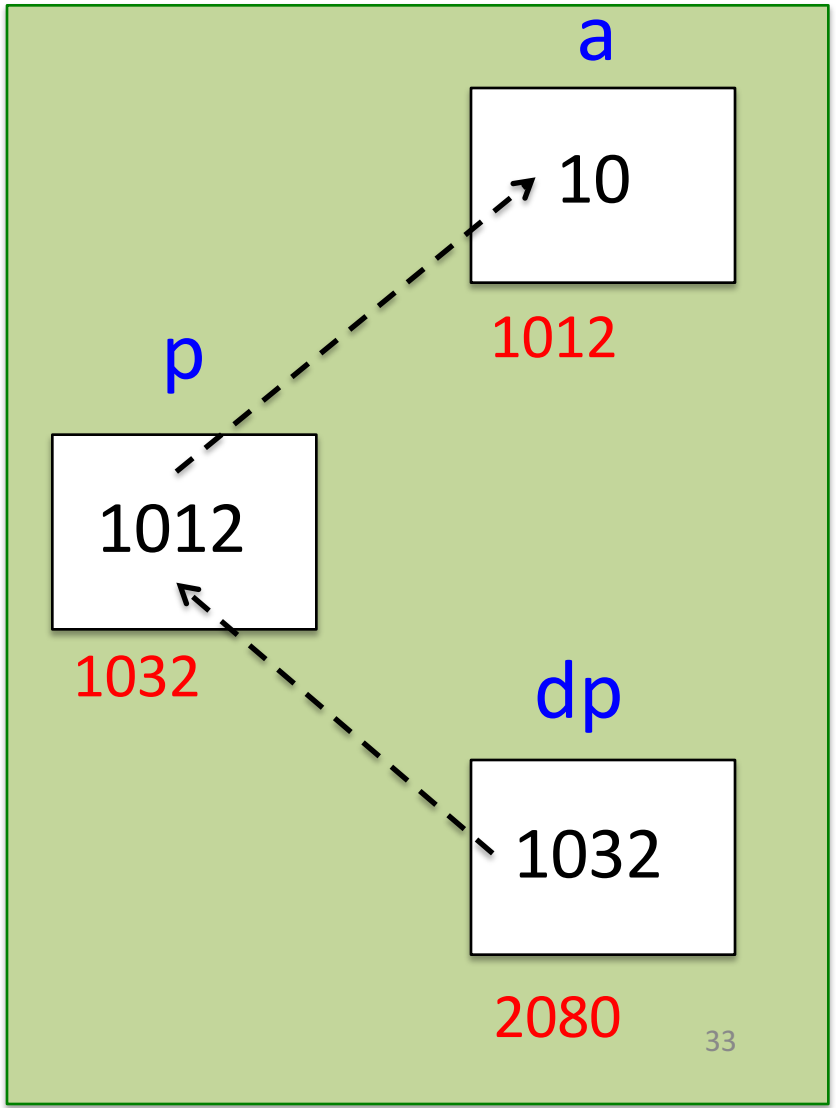
- int a = 10;
- int* p = &a;
- int** dp = &p;

- printf("a = %d\n", a);
- printf("*p = %d\n", *p);
- printf("**dp = %d\n", **dp);

- All printfs print 10 as the result.

- A double pointer implies two indirections or hops to actual data.

- Moral: n pointer implies n indirections or hops to the actual payload or data.



Exercise 2: Given the data, which of the options will correctly compile and execute.

- Given:
  - int a = 10;
  - int* p;
  - p = &a; // initializing pointer after declaration

1. First:
   - int** dp;
   - *dp = a;

2. Second:
   - int** dp = &a;

3. Third:
   - int** dp = &p;
   - int***tp = &(&p);

4. Fourth:
   - int** dp = &p;
   - int*** tp = &dp;

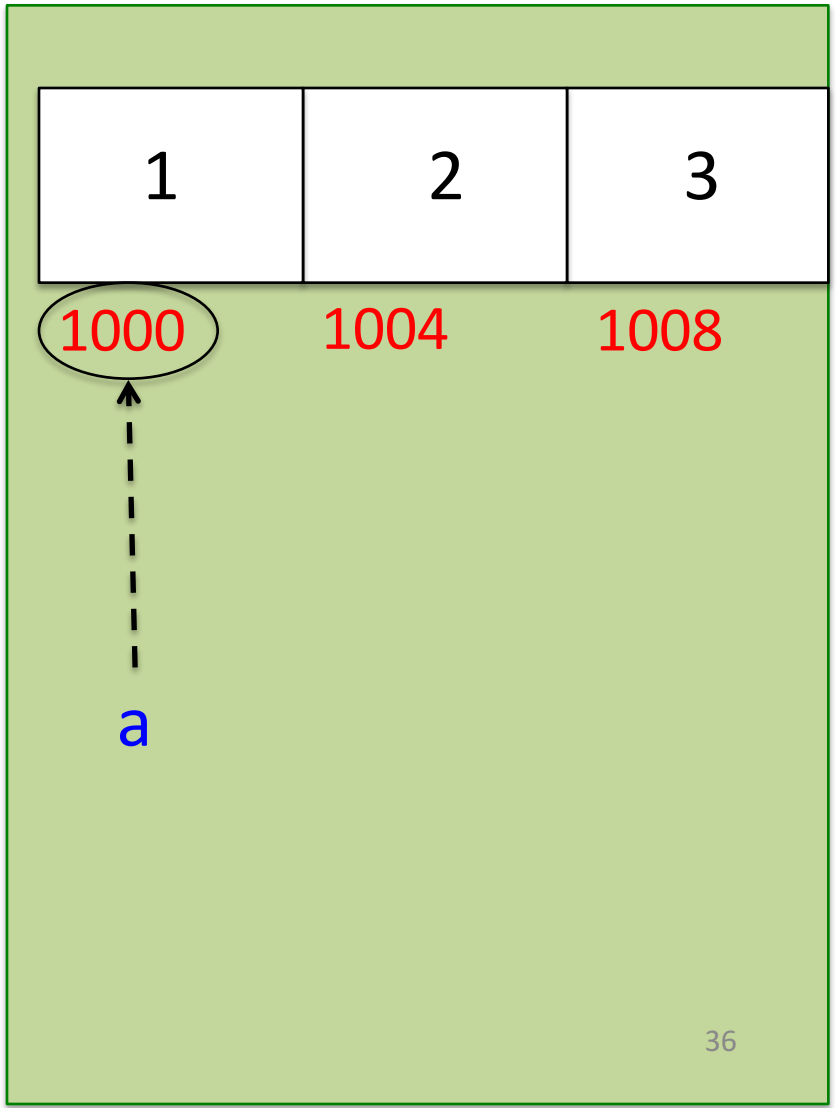
PART 2

1) What is pointer arithmetic?

2) Is an array variable essentially a pointer variable?

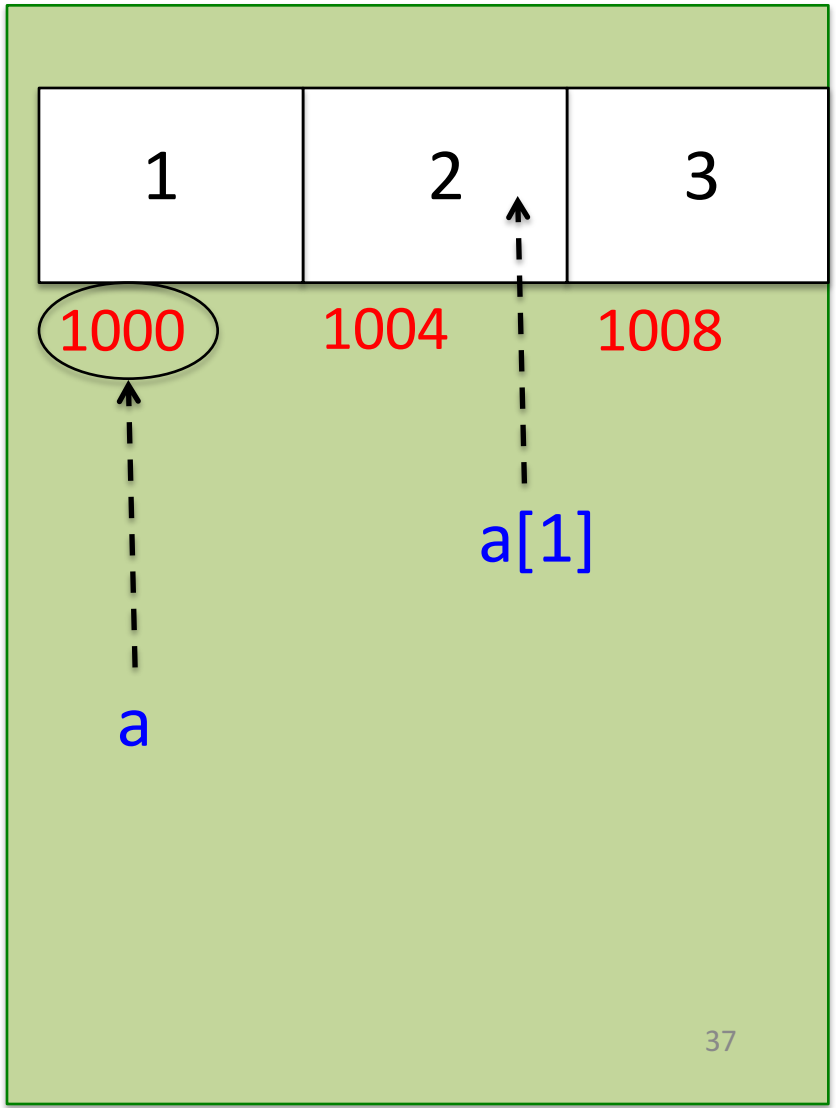
Pointer arithmetic

- int a[3] = {1, 2, 3};
- //sizeof(int) = 4.



Pointer arithmetic (2)

- int a[3] = {1, 2, 3};
- printf("%d\n", a[1]);
- // This prints 2.

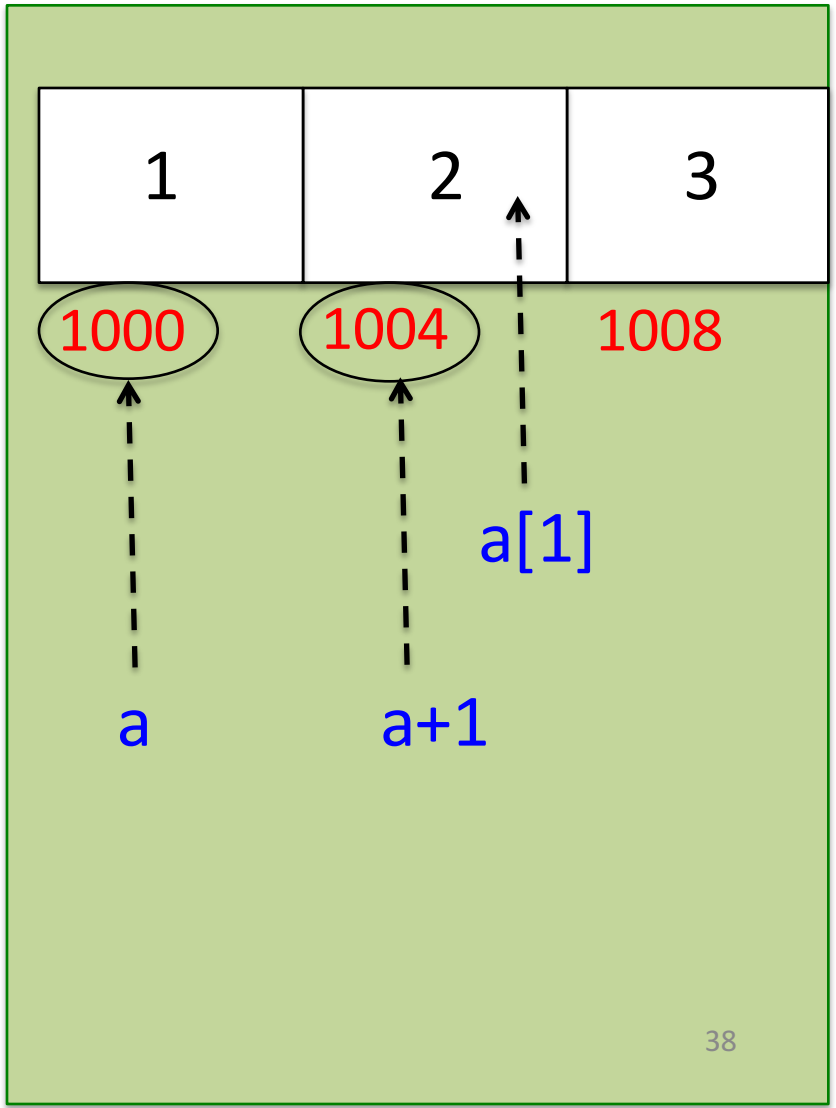


Pointer arithmetic (2)

- int a[3] = {1, 2, 3};

- printf("%d\n", a[1]);
- // This prints 2.

- printf("%d", ***(a+1)**);
- // This also prints 2.



Pointer arithmetic (2)

- int a[3] = {1, 2, 3};

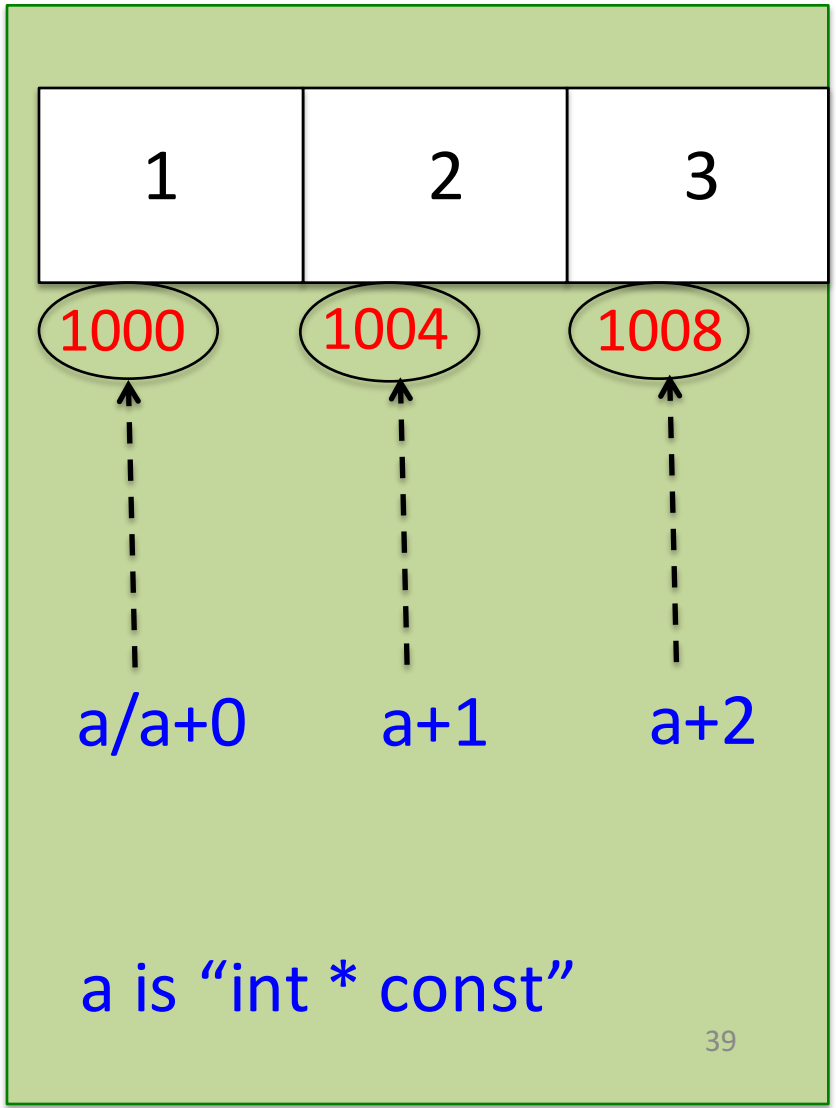
- printf("%d\n", a[1]);
- printf("%d", *(a+1));

- Arrays defined statically (i.e. at compile time) are constant pointers.

- a[1] ➔ *(a+1)

- Here the a+1 implies
  a + 1 * sizeof(int)

- This is called pointer arithmetic.



PART 3

1) How do we dereference pointer to a struct or a class variable ?

2) When do we use '->' and when do we use the '.' operator?

Pointer to a class instance

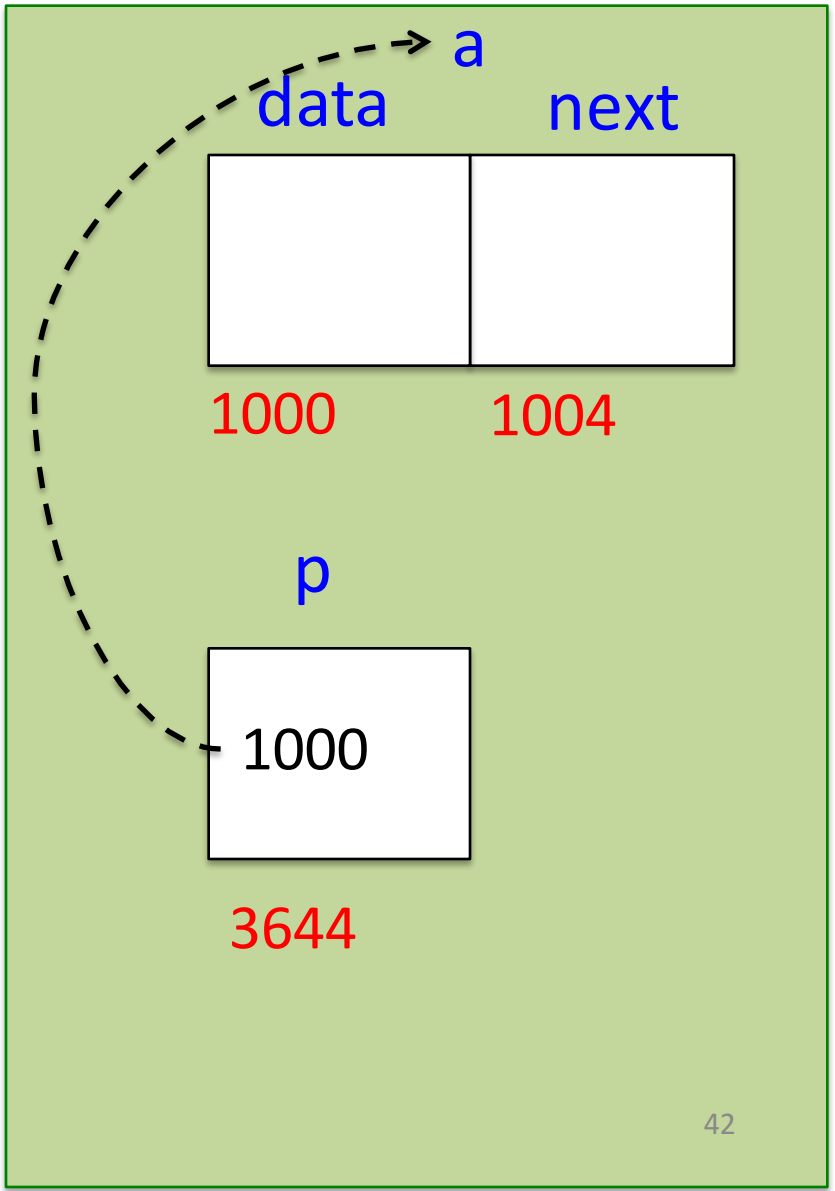
```
class node {
 public:
  int data;
  node* next;
};
```

```
void main () {
  node a;
  node* p = &a;
 }
```

Pointer to a class instance (2)

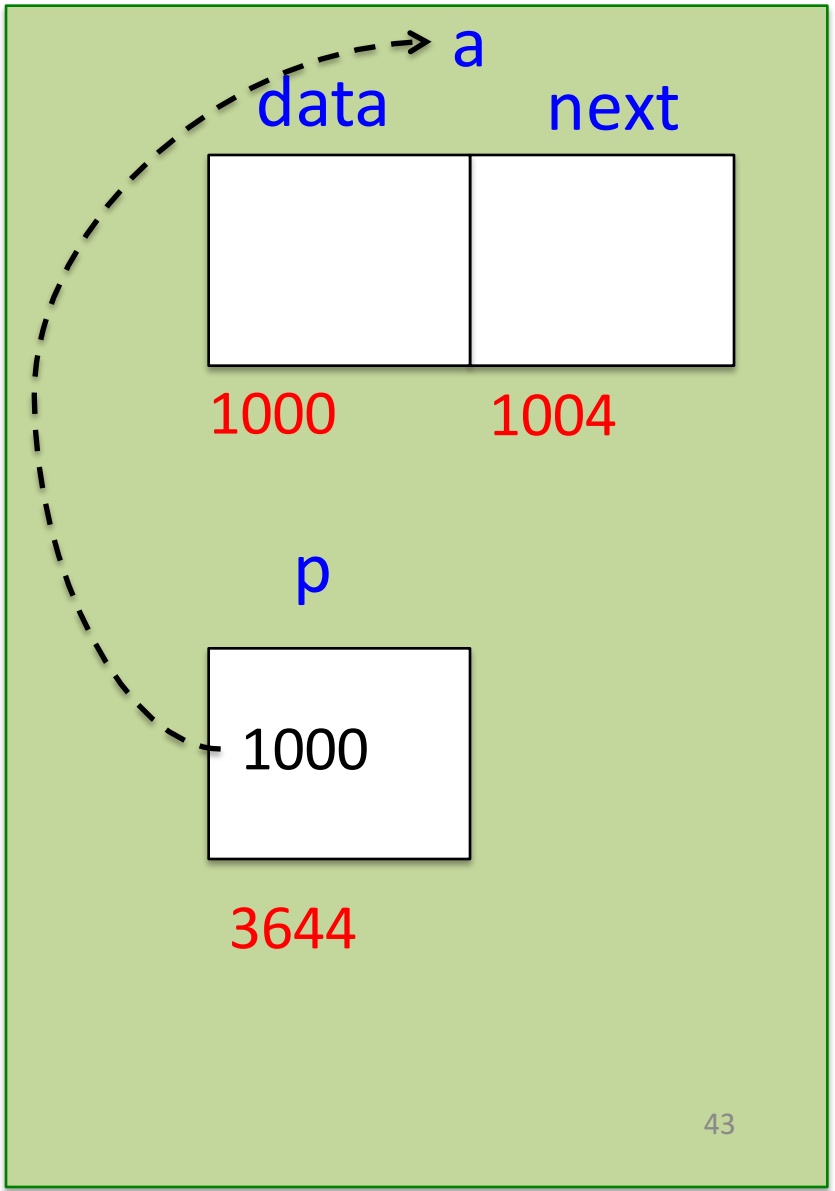
- class node {
  public:
   int data;
   node* next;
};

- void main () {
   node a;
   node* p = &a;
  }



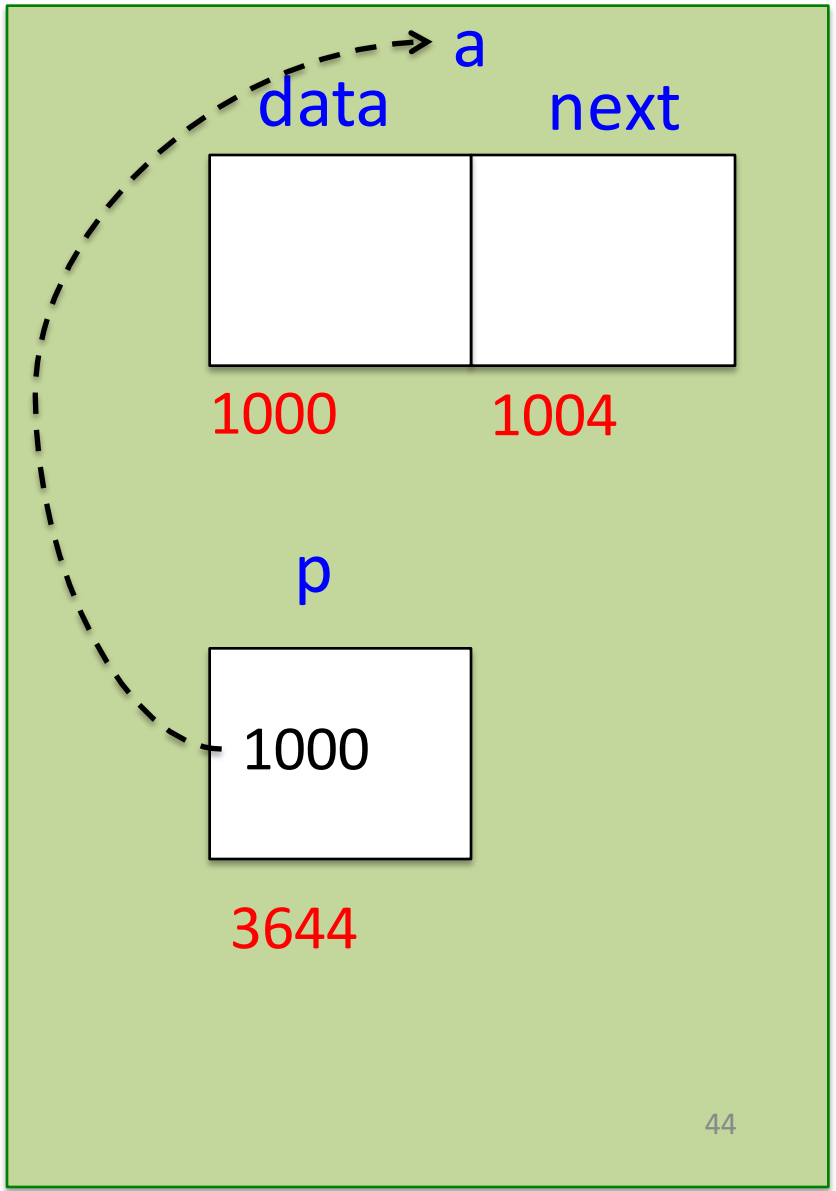
Pointer to a class (3): '.' and '->'

- Given a, we can access data or next variables using the '.' operator.
  - printf("%d\n", **a.data**);
- To access these variables using pointer p, we can use the '->' operator.
  - printf("%d\n", **p->data**);



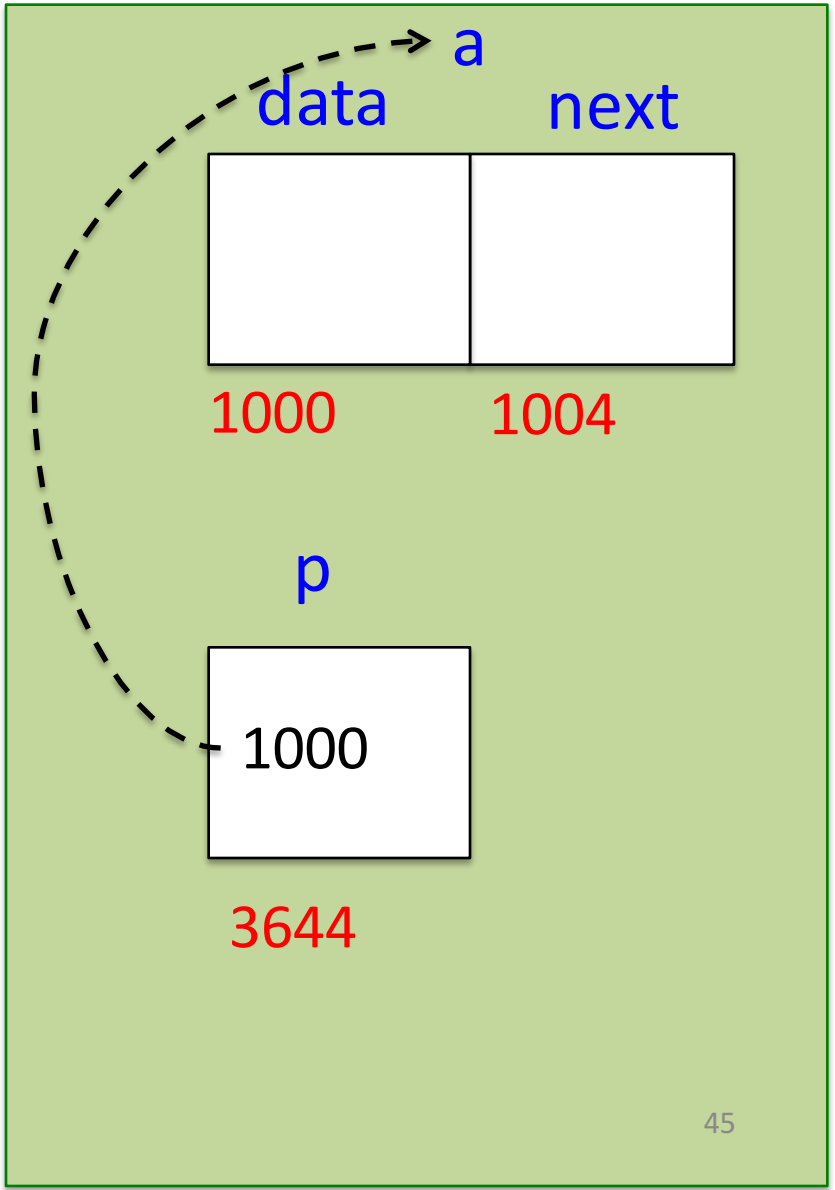
Pointer to a class(4): '.' and '->'

- '.' operator expects a class instance on its left side and a class member on its right side.
  - **(*cls_inst*).(cls_*mem*)**
- '->' operator expects address of a class instance on its left side and a class member on its right side.
  - **(*cls_inst_ptr*)->(*cls_mem*)**



Exercise 3

- Using a and '->' can you access the variable data?
- Using p and '.' can you access the variable data?
- Why or why not?
  - If you can access then how can this be done?



References

[1] Kernighan, B. W. and D. M. Ritchie. *The C Programming language*. Prentice Hall PTR, (1988).

[2] Kanetkar Y. P. *Let us C*. Jones & Bartlett Learning, (1999).

[3] Stroustrup B. *The C++ Programming Language*. Addison-Wesley Professional. (1985).

[4] Lafore, R. *Object-Oriented Programming in C++*. Waite Group. (1998).

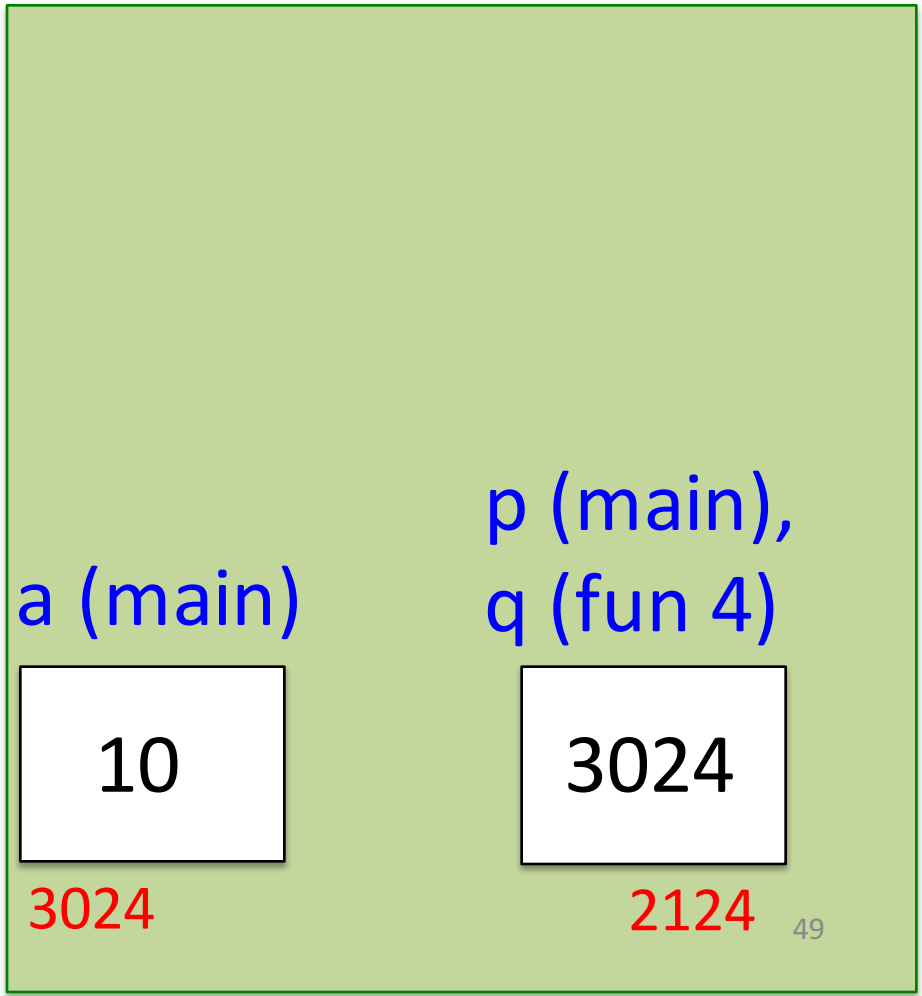
Thank you ☺

- Slides can be found on my website:
  - https://sites.google.com/site/virkaryogesh66/home/tutorials

Solutions: Exercise 1

```
void fun4(int*& q) {
q = new int[3];
}
```

- ```
  void main () {
  int a = 10;
  int* p = &a;
  fun4(p);
  }
  ```


  ```

Solutions: Exercise 1

- void fun4(int*& q) {

  q = new int[3];

  }

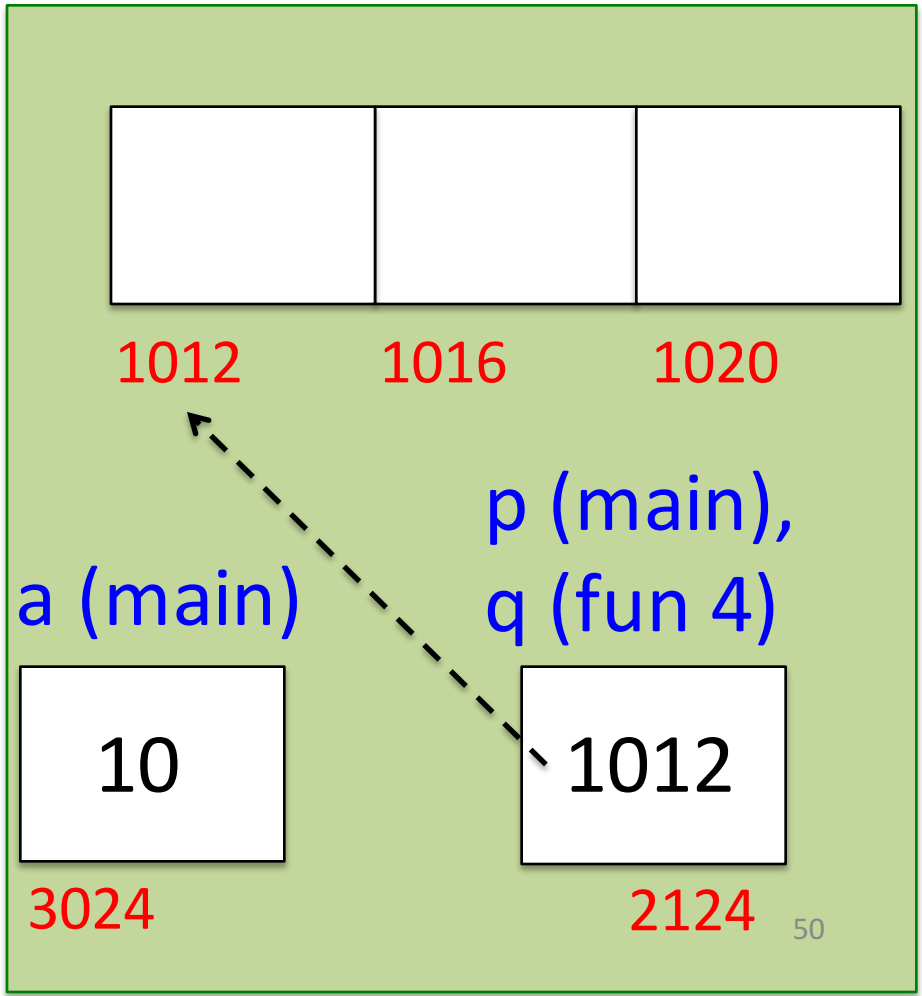
- void main () {

  int a = 10;

  int* p = &a;

  fun4(p);

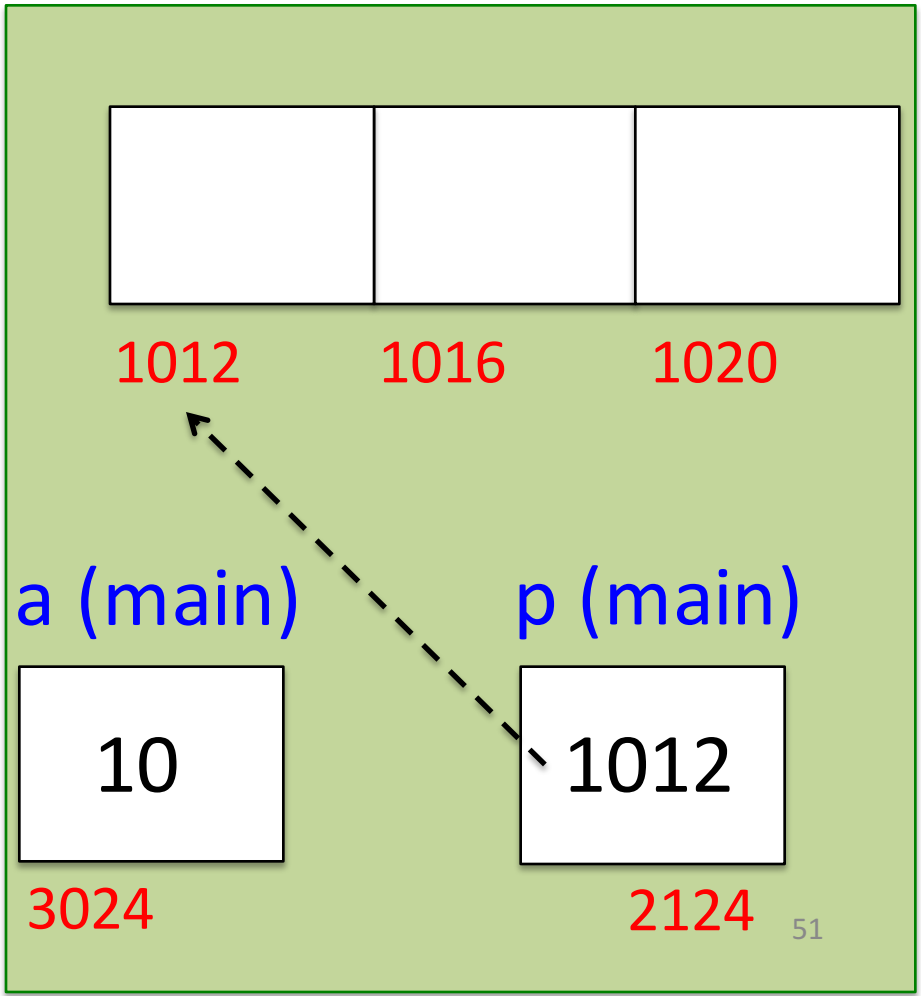
  }



Solutions: Exercise 1

- void fun4(int*& q) {
  q = new int[3];
  }

- void main () {
  int a = 10;
  int* p = &a;
  fun4(p);
  }



Solution: Exercise 2 (Only 4 is correct)

- Given:
  - int a = 10;
  - int* p;
  - p = &a; // initializing pointer after declaration

1. First:
   - int** dp;
   - *dp = a;
   - // cannot dereference uninitialized pointer

2. Second:
   - int** dp = &a;
   - //double pointer can only hold address of a single pointer.

3. Third:
   - int** dp = &p;
   - int***tp = &(&p);
   - // &(&p) is trying to get an address of a constant literal, i.e. &p. This is meaningless.

4. Fourth:
   - int** dp = &p;
   - int*** tp = &dp;

Solution: Exercise 3

- Using a and '->' can you access the variable data?
  - Yes.
  - (&a)->data;
- Using p and '.' can you access the variable data?
  - Yes.
  - (*p).data;