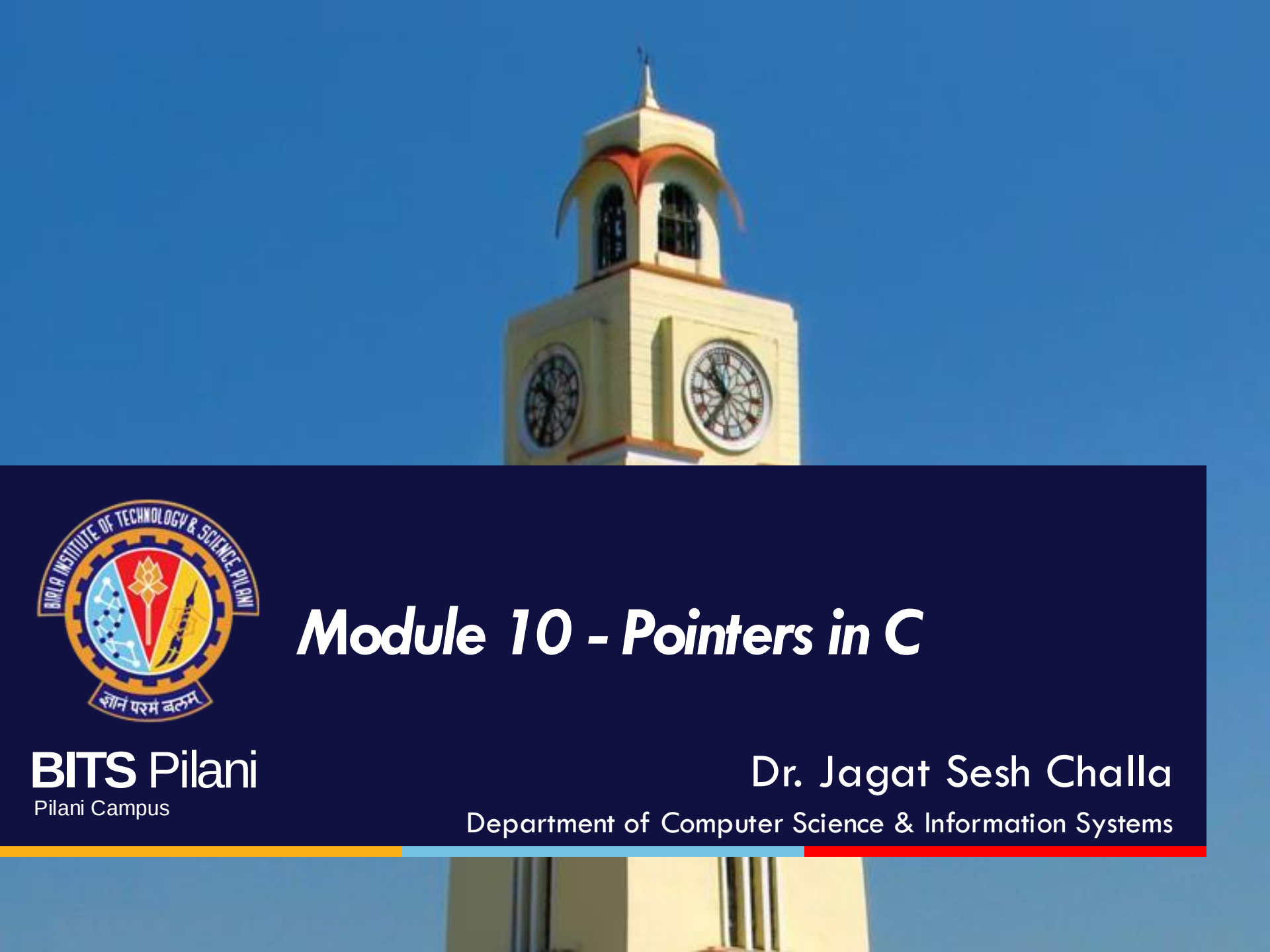


Module 10 - Pointers in C

BITS Pilani
Pilani Campus

Dr. Jagat Sesh Challa
Department of Computer Science & Information Systems

Pointers in C

innovate achieve lead

"Pointers are the beauty of C"

Why?

- ✓ *Allows memory level operations*
  - ✓ *Very few Programming languages allow this: C, C++, Fortran*
- ✓ *Enables "Pass by Reference"*
- ✓ *Enables us to return multiple data items from functions*
  - ✓ *Like arrays, large complex structures*
- ✓ *Enables dynamic memory allocation at run-time*
  - ✓ *You don't need to fix the input size at the time of programming*
- ✓ *Many More…*

Module Overview

- Pointers in C
- Pointer Arithmetic
- Arrays and Pointers
- Structures and Pointers
- Pass by reference

innovate achieve lead

BITS Pilani

Pilani Campus

Pointers in C

Addresses and Pointers

Consider a variable declaration in the following program:

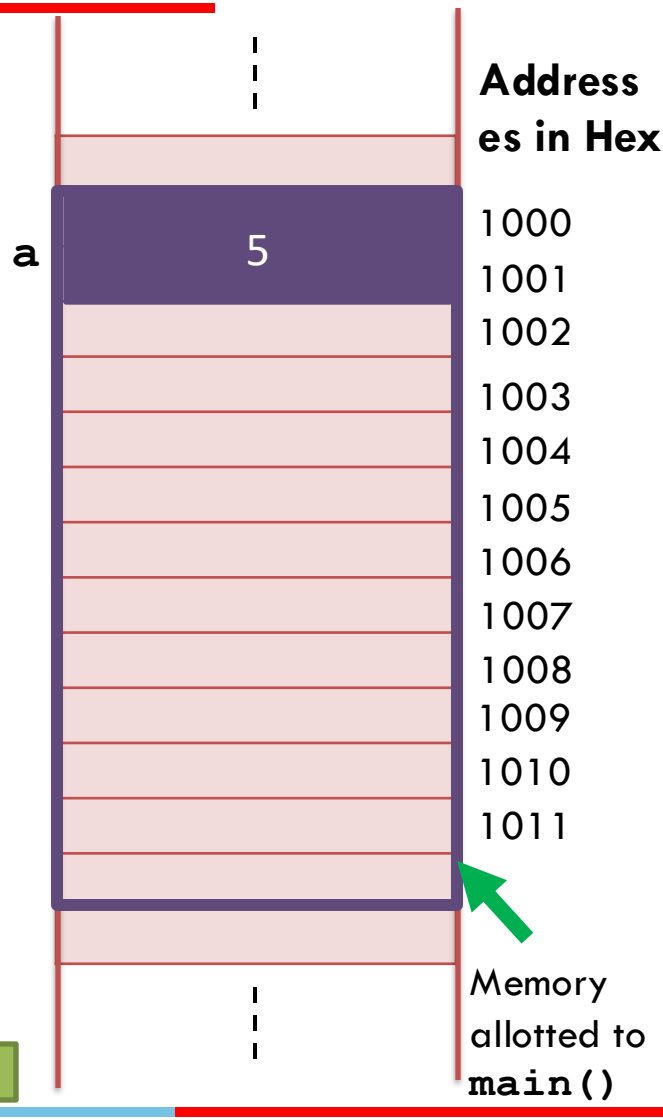
```
int main(){
        int a = 5;
}
```

Say, the variable "a" occupies 2 bytes starting at memory location whose address is 1000 (**in hexa-decimal**). (Assuming 16-bit addresses)

This address can be accessed by "&a"

```
printf("Value of a: %d",a);          5
printf("Address of a: %p",&a);       1000
```

%p used to print addresses



Addresses and Pointers

- The address of this variable **a**, can be stored in a variable called a ***pointer variable***

```
int * ptr = &a;
```

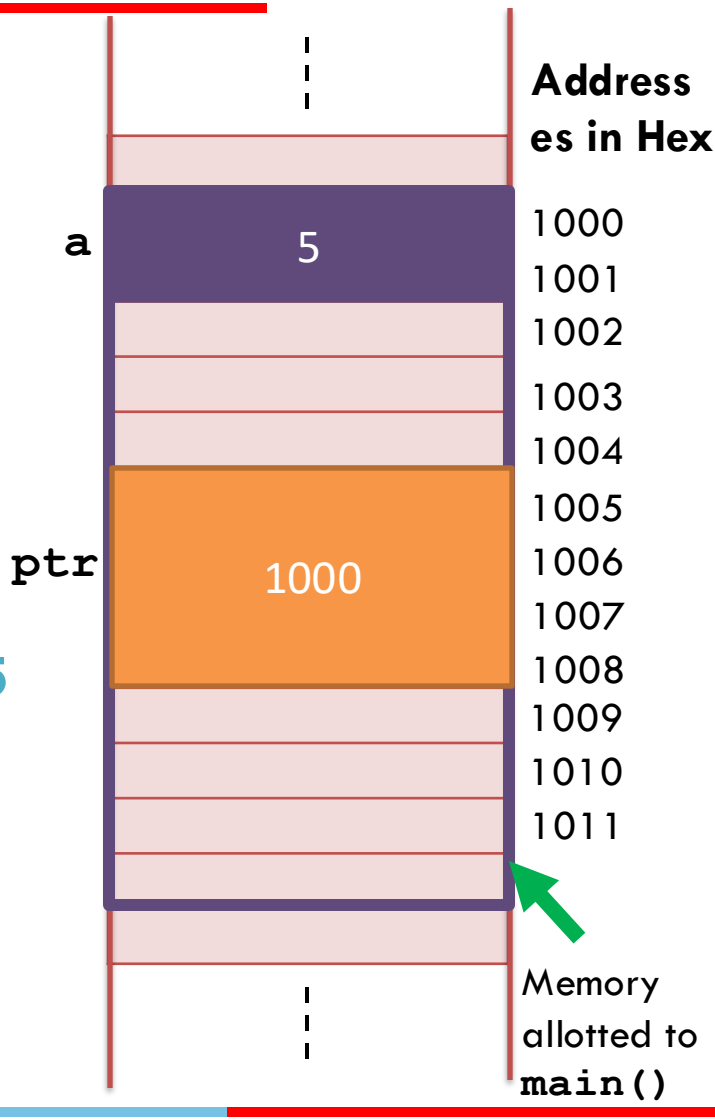
- **ptr** is a pointer variable of integer type. It is capable of storing the address of an integer variable.

- We can access the value stored in the variable **a** by ***ptr**

```
printf("Value of a: %d",*ptr);
```

5

- ***ptr** translates to ***value at** **ptr**. (**de-referencing**)
- *Pointer variable of any type typically occupies 4 bytes* (or 8 bytes) in memory depending upon the compiler.

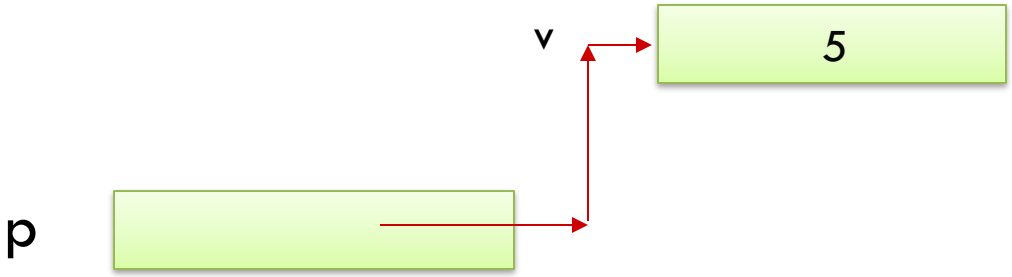


Simplifying with an example

Given declarations

```
int v;
int *p;
```

`p = &v;` is a valid assignment statement.



What is the effect of the following?

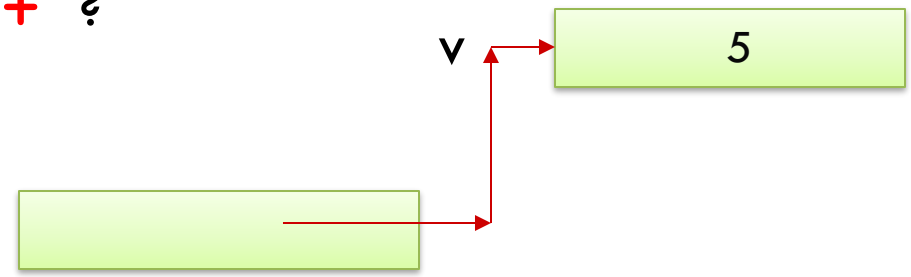
```
v = 5;
```

Simplifying with an example…

Now, what is the effect of `(*p)++` ?

`(*p)++` is the same as

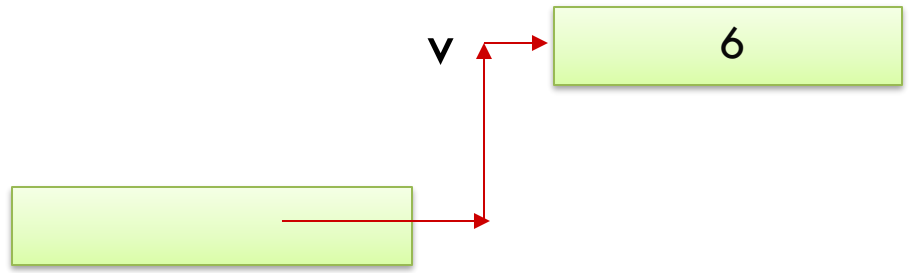
```
*p = *p + 1;
```



`*p` (i.e., contents of p) is 5;

And it is changed to 6;

So `v` is also 6



Example with float

```
float u,v;        // floating-point variable declaration
float * pv;       // pointer variable declaration
……
pv = &v;          // assign v's address to pv
```

u and **v** are floating point variables

pv is a pointer variable which points to a floating-point quantity

Another Example

```
int main(){
    int v = 3, *pv;

    pv = &v;

    printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);

    v=v+1;

    printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);
}
```

Note: %p prints contents of a pointer variable which is the address

64-bit addresses

Output:

```
v=3, pv=0x7ffc57c45a78, *pv=3
v=4, pv=0x7ffc57c45a78, *pv=4
```

innovate achieve lead

BITS Pilani

Pilani Campus

Pointer Arithmetic

Size of a Pointer

	Size
Borland C / Turbo C	2 bytes
32 – bit architecture	4 bytes
64 – bit architecture	8 bytes

Modern Intel and AMD Processors are typically 64-bit architectures

Pointer Arithmetic

- **Incrementing a pointer**
  - NewPtr = CurrentPtr + N bytes
  - Where *N* is size of pointer data type.

Example:

```
int a = 5;
int * p = &a; // assume &a = 1000 (assume 16-bit addresses)
int * q = p + 1;
printf("printing ptr: %p", p);          1000
printf("printing ptr: %p", q);          1002
```

Incrementing **ptr** will *increase its value by 2* as int is of 2 bytes (assume)

What will be printed by the above print statements?

Pointer Arithmetic

innovate achieve lead

- **Adding K to a pointer**
  - NewPtr = CurrentPtr + K * N bytes.
  - Where K is a constant integer
  - N is size of pointer data type.

We will see its application when we study arrays with pointers!

Example:

```
int a = 5;
int * p = &a; // assume &a = 1000 (assume 16-bit addresses)
int * q = p + 4;
printf("printing ptr: %p", p);
printf("printing ptr: %p", q);
```

1000

1008

ptr will *increase its value by 8* as int is of 2 bytes and 4*2 = 8

What will be printed by the above print statements?

Pointer Arithmetic

- What does `*p++` and `*p+q` do?
  - Need Precedence and Associativity Rules to decide
- Rule 1: Unary operators have higher precedence than binary operators
  - So, `*p+q` is the same as `(*p) + q`
- Rule 2: All Unary operators have the same precedence.
  - So, we still need associativity to decide `*p++`
- Rule 3: All unary operators have right associativity.
  - So, `*p++` is the same as `* (p++)`
- What if you want to increment the contents?
  - Use `(*p)++`

Operator Precedence

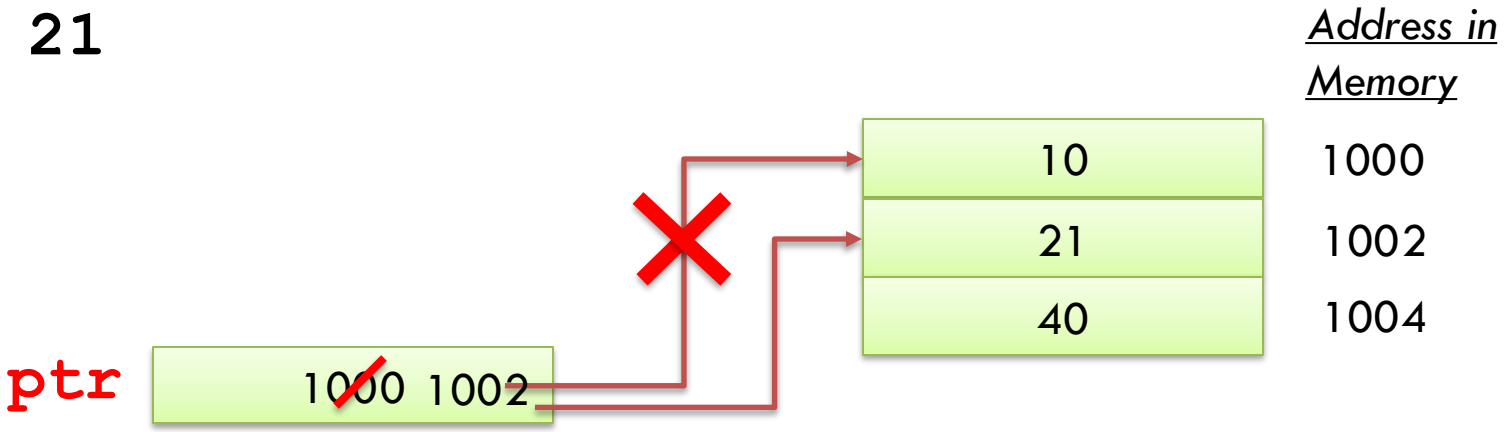
innovate achieve lead

```
()   []   ->   .
++   --   +   -   !   ~              (unary)
   (type)   *   &   sizeof
*   /   %
+   -
<<   >>
<   <=   >   >=
==   !=
&
^
|
&&
||
?:
=   +=   -=   *=   /=   %=
  &=   ^=   |=   <<=   >>=
,
```

Now, let us see various cases of incrementing a pointer variable, based on this precedence

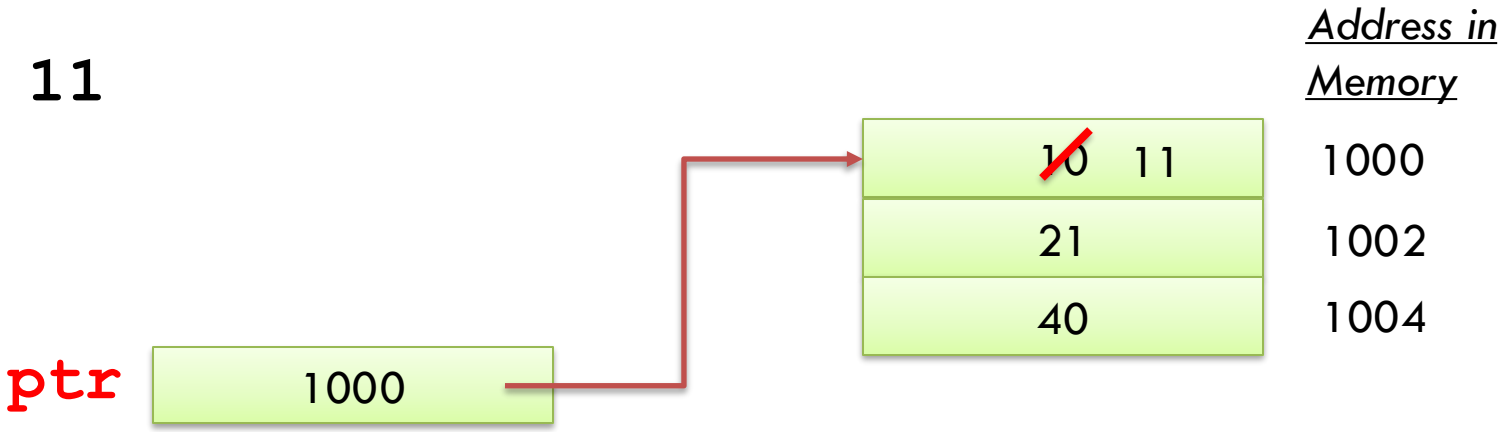
Incrementing a pointer variable: Case 1

```
int c1 = *++ptr;
// c1 = *(++ptr);
// increment ptr and dereference its (now incremented) value

// c1 = 21
```



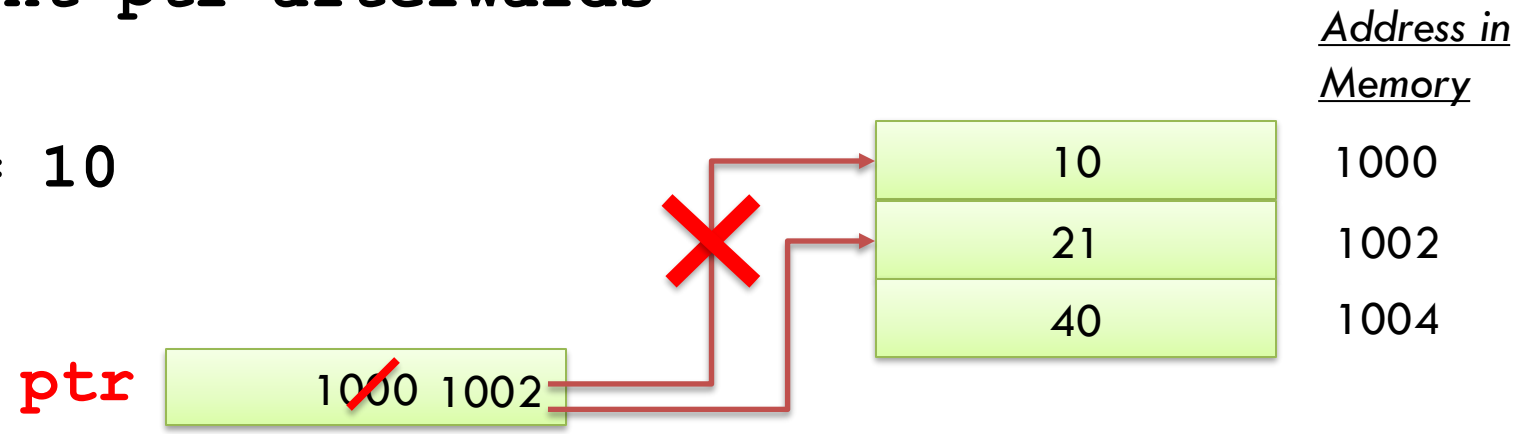
Incrementing a pointer variable: Case 2

```
int c2 = ++*ptr;
// c2 = ++(*ptr);
// dereference ptr and increment the
   dereferenced value

// c2 = 11
```



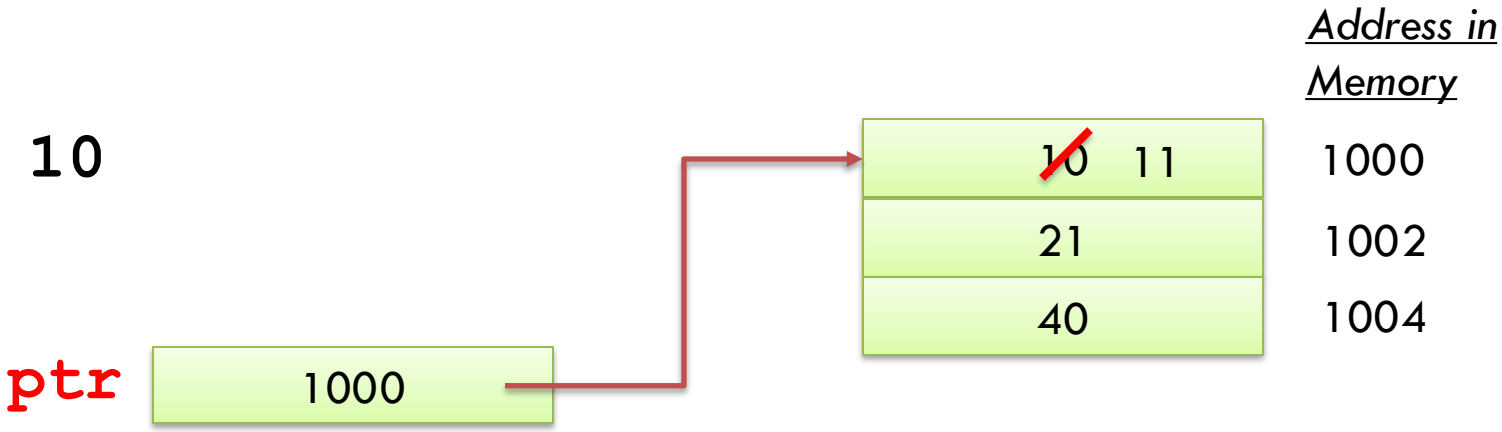
Incrementing a pointer variable: Case 3


```
int c3 = *ptr++; // or int c3 = *(ptr++);
// both of the above has same meaning
// c3 = *ptr; ptr = ptr+1;
// dereference current ptr value and
increment ptr afterwards

// c3 = 10
```



Incrementing a pointer variable: Case 4


```
int c4 = (*ptr)++;
// c4 = *ptr; *ptr = *ptr + 1;
// dereference current ptr value and increment
   the dereferenced value - now we need
   parentheses

// c4 = 10
```



Example

```
int main() {
   int v =3, *pv;

   pv = &v;

   printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);

   *pv++;

   printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);
}
Output:
v=3, pv=0x7ffeb75a749c, *pv=3
v=3, pv=0x7ffeb75a74a0, *pv=-1218808672
```

Note the difference of 4 bytes in the addresses. In this example, size of int is assumed to be 4 bytes.

garbage

Example – Variation 1

```
int main() {
   int v =3, *pv;

   pv = &v;

   printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);

   (*pv)++;

   printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);
}
Output:
v=3, pv=0x7ffde009df1c, *pv=3
v=4, pv=0x7ffde009df1c, *pv=4
```

No change to the address stored in the pointer variable. The value stored is incremented.

Example – Variation 2

```
int main() {
   int v =3, *pv;

   pv = &v;

   printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);

   ++*pv;

   printf("v=%d, pv=%p, *pv=%d\n", v, pv, *pv);
}
Output:
v=3, pv=0x7fffbb21f63c, *pv=3
v=4, pv=0x7fffbb21f63c, *pv=4
```

No change to the address stored in the pointer variable. The value stored is incremented.

innovate achieve lead

BITS Pilani

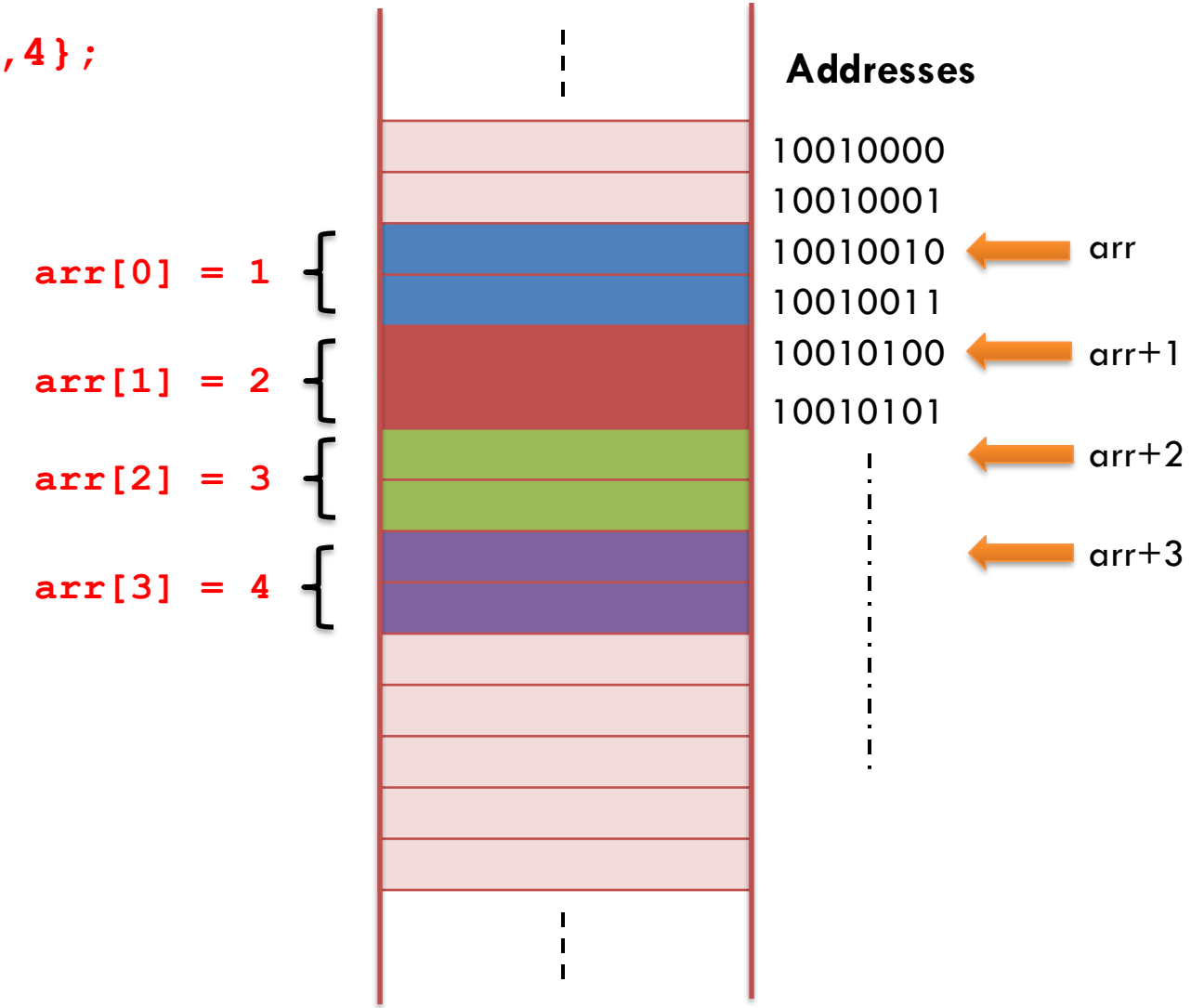
Pilani Campus

Arrays and Pointers

Arrays and Pointers

```
int arr[4] = {1,2,3,4};
```

- **arr** stores the address of the first element of the array
- **arr** is actually a pointer variable
- **arr+1** gives the address of the second element
- difference between **arr** and **arr+1** is actually 2 bytes
- **arr+2** gives the address of the third element and so on…



Arrays and Pointers

innovate achieve lead

Accessing elements of arrays

`arr[0]` is same as `*arr`

`arr[1]` is same as `*(arr+1)`

`arr[2]` is same as `*(arr+2)`

…

Address of first element:

`arr` or `&arr[0]`

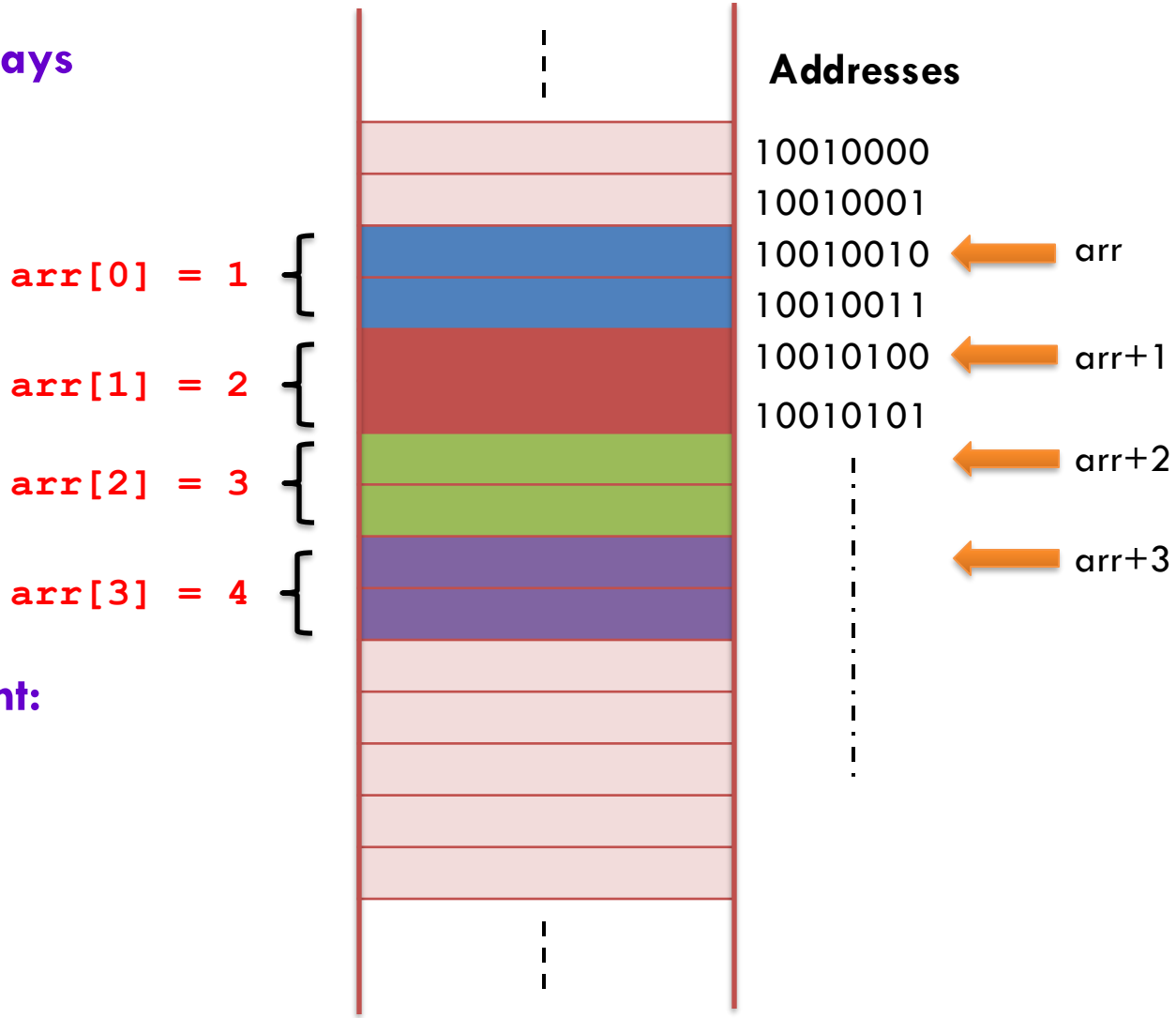
Address of second element:

`arr+1` or `&arr[1]`

Address of third element:

`arr+2` or `&arr[2]`

and so on…

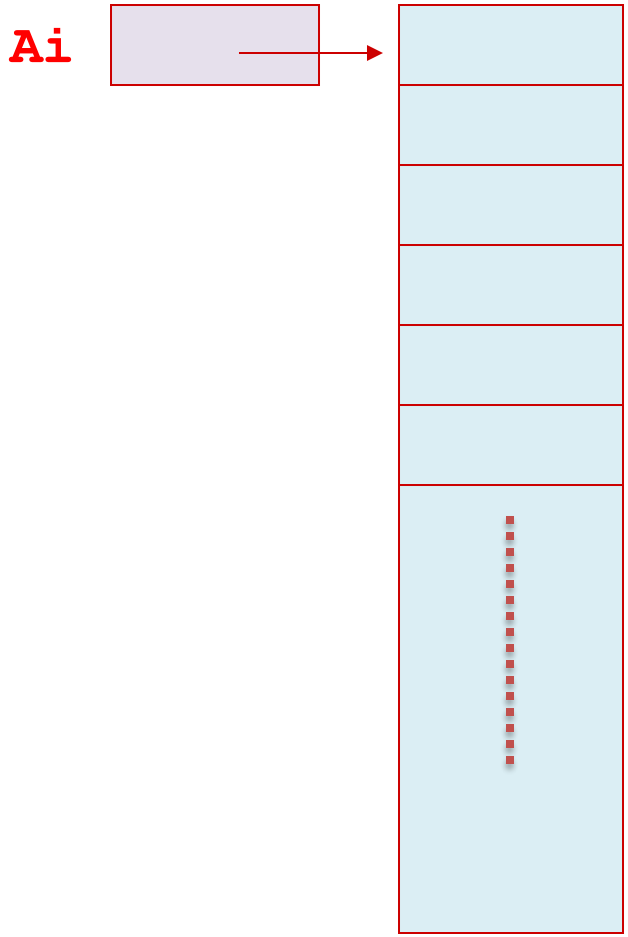


Simplifying with an example

Given

```
int Ai[100];   // array of 100 ints
```

Ai is the starting address of the array.



So, `Ai[5]` is same as `*(Ai+5)`

Observe that `Ai+5` is "address arithmetic":

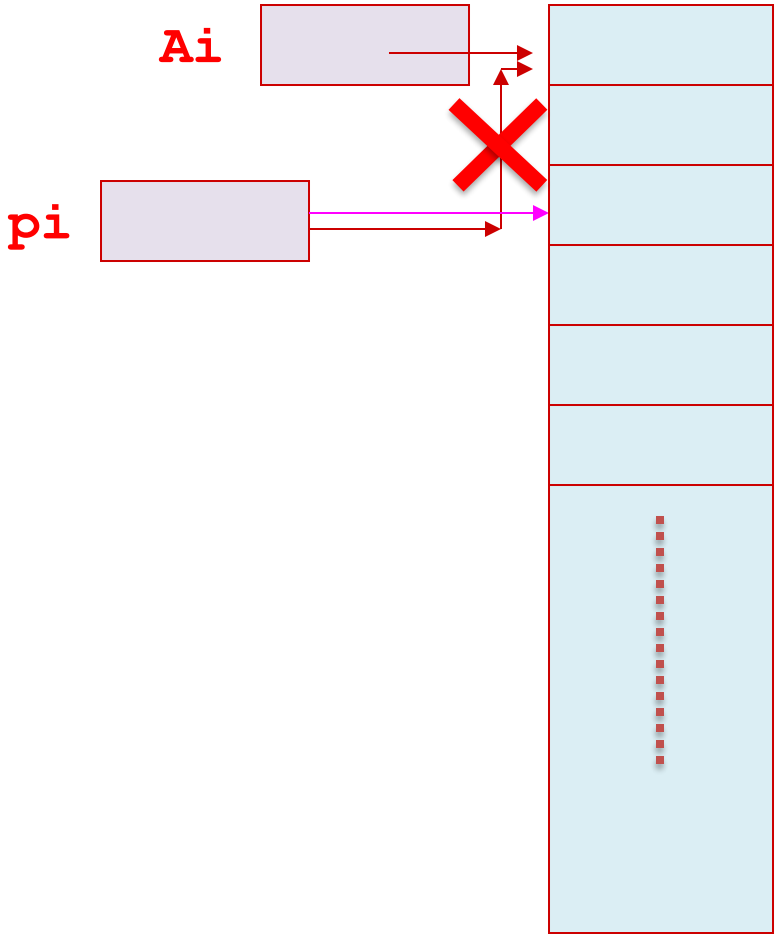
address `Ai` is added to int `5`

to obtain an address.

Simplifying with an example

Given

```
int Ai[100];
int *pi;
```



the following are valid:

```
pi = Ai
```

```
pi = Ai +2
```

`pi - Ai` which will evaluate to `2`

Example

```
int main(){
int *ptr, i, iA[5]={5,10,15,20,25};
for(i=0;i<5;i++)
   printf("iA[%d]:address=%p
   data=%d", i, &iA[i], iA[i]);

// Accessing the Arrays using
// pointer
ptr = iA;
for(i=0;i<5;i++){
   printf("\npointer address = %p
   data = %d ", ptr, *ptr);
   ptr++;
}
return 0;
}
```

This program assumes sizeof(int) to be 4 bytes

```
Output:
iA[0]:address=0x7ffd2adfbf10 data=5
iA[1]:address=0x7ffd2adfbf14 data=10
iA[2]:address=0x7ffd2adfbf18 data=15
iA[3]:address=0x7ffd2adfbf1c data=20
iA[4]:address=0x7ffd2adfbf20 data=25
pointer address = 0x7ffd2adfbf10 data = 5
pointer address = 0x7ffd2adfbf14 data = 10
pointer address = 0x7ffd2adfbf18 data = 15
pointer address = 0x7ffd2adfbf1c data = 20
pointer address = 0x7ffd2adfbf20 data = 25
```

Example: Array of Pointers

innovate achieve lead

```
int main(){
    int *ptr[3], i, iA[]={5,10,15}, iB[]={1,2,3}, iC[]={2,4,6};
    ptr[0]=iA;
    ptr[1]=iB;
    ptr[2]=iC;
    for(i=0;i<3;i++) {
         printf("iA[%d]:addr=%p data=%d ",i,ptr[0]+i,*(ptr[0]+i));
         printf("iB[%d]:addr=%p data=%d ",i,ptr[1]+i,*(ptr[1]+i));
         printf("iC[%d]:addr=%p data=%d ",i,ptr[2]+i,*(ptr[2]+i));
    }
    return 0;
}
```

This program assumes sizeof(int) to be 4 bytes

```
Output:
iA[0]:addr=0x7ffe7213707c data=5
iB[0]:addr=0x7ffe72137088 data=1
iC[0]:addr=0x7ffe72137094 data=2
iA[1]:addr=0x7ffe72137080 data=10
iB[1]:addr=0x7ffe7213708c data=2
iC[1]:addr=0x7ffe72137098 data=4
iA[2]:addr=0x7ffe72137084 data=15
iB[2]:addr=0x7ffe72137090 data=3
iC[2]:addr=0x7ffe7213709c data=6
```

Another example

```
int main(){
   int line[]={10,20,30,40,50};

   line[2]=*(line + 1);
   *(line+1) = line[4];

   int *ptr;      ptr = &line[5];      ptr--;
   *ptr = line[3];
   *line=*ptr;

   for(int i =0;i<5;i++)
       printf("%d ", *(line+i));
   return 0;
}
```

Output:

40 50 20 40 40

innovate achieve lead

BITS Pilani

Pilani Campus

Pointers to Structures

Pointers to Structure

innovate achieve lead

Consider the following structure definitions:

```
struct stud {
   int roll;
   char dept_code[25];
   float cgpa;
} class, *ptr;
```

```
struct book {
   char Name[20];
   float price;
   char ISBN[30];
};
struct book b, *br;
```

```
ptr = &class   ✓
br = &b;  ✓
```

```
ptr = &b  ✗
br = &class;  ✗
```

Accessing members in pointers to Structures

- Once `ptr` points to a structure variable, the members can be accessed through dot(`.`) or arrow operators:

```
(*ptr).roll,  (*ptr).dept_code,   (*ptr).cgpa
```

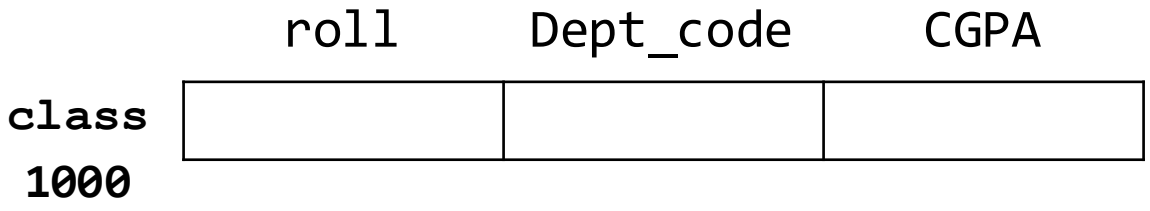
OR

```
ptr->roll,    ptr->dept_code,     ptr->cgpa
```

Syntactically, `(*p).a` is equivalent to `p->a`

Illustration

```
struct stud {
   int roll;
   char dept_code[25];
   float cgpa;
} class, *ptr;
```




```
ptr = &class  // ptr = 1000 (assumes 16-bit addresses)
```

Caveats

- When using structure pointers, we should take care of operator precedence.
- Member operator "`.`" has higher precedence than "`*`"
- `ptr->roll` and `(*ptr).roll` mean the same thing. `*ptr.roll` will lead to error
- The operator "`->`" has the highest priority among operators.
- `++ptr->roll` will increment *roll*, not *ptr*
- `(++ptr)->roll` will do the intended thing.

Pointers to Array of Structures

innovate achieve lead

```
struct stud {
   int roll;
   char dept_code[25];
   float cgpa;
} class[3], *ptr;
```

	roll	Dept_code	CGPA
1000 Class[0]			
1036 Class[1]			
1072 Class[2]			

```
ptr = class; // ptr = 1000, ptr+1 = 1036, ptr+2 = 1072
```

The assignment `ptr = class` assigns the address of `class[0]` to `ptr`

Example

```
struct stud {
   int roll;
   char dept_code[25];
   float cgpa;
};
int main()
{
   int i=0;
   struct stud sArr[3];
   struct stud * ptr;
   ptr = sArr;
   for(i=0;i<3;i++)
   {
      scanf("%d",&(sArr[i].roll));
      scanf("%s",sArr[i].dept_code);
      scanf("%f",&(sArr[i].cgpa));
   }
   int avgCGPA = 0;
   for(i=0;i<3;i++)
   {
      avgCGPA += (ptr+i)->cgpa;
   }
   avgCGPA /=3;
   printf("AvgCGPA is %d",avgCGPA);
   return 0;
}
```

innovate achieve lead

BITS Pilani

Pilani Campus

Call/Pass by reference
Variable, Arrays and Structures

Swapping two variables using a function: Attempt 1 – Pass by value

innovate achieve lead

First Attempt: Pass by value

```
void swap(int x, int y){
  int temp = x;
  x = y;
  y = temp;
}
```

```
int main(){
  int a = 10, b = 20;
  printf("Before Swapping %d %d\n", a, b);
  swap(a, b);
  printf("After Swapping %d %d\n", a, b);
  return 0;
}
```

```
Output:
10 20
10 20
```

- The values of **a** and **b** get copied into **x** and **y**
- The swapping of **x** and **y** doesn't get reflected back in **a** and **b** when **swap()** function returns
- Also, we can't return **x** and **y** to **main()** function as C supports return of a single variable.

Swapping two variables using a function: Attempt 2 – Pass by reference

innovate achieve lead

Second Attempt: Pass by reference

```
void swap(int *x, int *y) {
   int temp = *x;
   *x = *y;
   *y = temp;
}
```

```
int main() {
   int a = 10, b = 20;
   printf("Before Swapping %d %d\n", a, b);
   swap(&a, &b);
   printf("After Swapping %d %d\n", a, b);
   return 0;
}
```

- The addresses of **a** and **b** get copied into **x** and **y**
- The swapping of ***x** and ***y** gets reflected back in **a** and **b** when **swap()** function returns

Passing Arrays into functions

innovate achieve lead

Passing arrays into functions is by default call by reference.

```
void sort(int a[]) {
  int temp, i , j, sorted = 0;
  for(i = 0; i < SIZE-i-1; i++){
    for(j = 0; j<SIZE-1-i; j++){
      if(a[j] > a[j + 1])
      {
          temp = a[j];
          a[j] = a[j + 1];
          a[j + 1] = temp;
      }
    }
  }
}
```

When arrays are passed as parameters, you pass the address of the first location which the array variable name

```
int main() {
   int arr[8] = {2,5,9,7,1,5,4,6};
   int SIZE = 8;
   printf("Array before sort: \n");
   for (i = 0; i < SIZE; i++)
        printf("%d ", arr[i]);
   printf("\n");
   sort(arr);
   printf("Array after sort: \n");
   for (i = 0; i < SIZE; i++)
        printf("%d ", arr[i]);
   printf("\n");
   return 0;
}
```

sort() function implements bubble sort which is one of the sorting algorithms. Don't worry about it.

Taking arrays as input parameter

```
void sort(int a[]) {
  int temp, i , j, sorted = 0;
  for(i = 0; i < SIZE-i-1; i++){
    for(j = 0; j<SIZE-1-i; j++){
      if(a[j] > a[j + 1])
      {
          temp = a[j];
          a[j] = a[j + 1];
          a[j + 1] = temp;
      }
    }
  }
}
```

```
void sort(int * a) {
  int temp, i , j, sorted = 0;
  for(i = 0; i < SIZE-i-1; i++){
    for(j = 0; j<SIZE-1-i; j++){
      if(a[j] > a[j + 1])
      {
          temp = a[j];
          a[j] = a[j + 1];
          a[j + 1] = temp;
      }
    }
  }
}
```

Both are equivalent…

Example: Computing length of a string

- Applications of Pointer arithmetic:

```
int strlen(char *str) {
    char *p;
    for (p=str; *p != '\0'; p++);
    return p-str;
}
```

- Observe the similarities and differences with arrays.

```
int strlen(char str[]) {
    int j;
    for (j=0; str[j] != '\0'; j++);
    return j;
}
```

Character Arrays and Pointers: Example 2


```
Boolean isPalindrome(char *str) {

   char *fore, *rear;
   fore = str; rear = str + strlen(str) -1;

   for (; fore < rear; fore++, rear--)
        if (*fore != *rear) return FALSE;

   return TRUE;
}
```

Pass by reference using structures

innovate achieve lead

```
typedef struct{
   int a;
   float b;
} ST;

void swap(ST * p1, ST * p2){
    ST temp;
    temp.a = (*p1).a;
    temp.b = (*p1).b;
    (*p1).a = (*p2).a;
    (*p1).b = (*p2).b;
    (*p2).a = temp.a;
    (*p2).b = temp.b;
}
```

```
int main() {

ST s1, s2;
s1.a=10; s1.b=10.555;
s2.a=3; s2.b=3.555;

printf("s1.a:%d, s1.b:%f\n",s1.a,s1.b);
printf("s2.a:%d, s2.b:%f\n",s2.a,s2.b);

swap(&s1, &s2);

printf("s1.a: %d, s1.b:%f\n",s1.a,s1.b);
printf("s2.a: %d, s2.b:%f\n",s2.a,s2.b);

}
```

Be careful of the precedence of "*"and "." , with the latter having higher precedence. You must use () if you want the operation to be correct

Pass by reference using structures (equivalent)

```
typedef struct{
   int a;
   float b;
} ST;


void swap(ST * p1, ST * p2){
    ST temp;
    temp.a = p1->a;
    temp.b = p1->b;
    p1->a = p2->a;
    p1->b = p2->b;
    p2->a = temp.a;
    p2->b = temp.b;
}
```

```
int main() {

ST s1, s2;
s1.a=10; s1.b=10.555;
s2.a=3; s2.b=3.555;

printf("s1.a:%d, s1.b:%f\n",s1.a,s1.b);
printf("s2.a:%d, s2.b:%f\n",s2.a,s2.b);

swap(&s1, &s2);

printf("s1.a: %d, s1.b:%f\n",s1.a,s1.b);
printf("s2.a: %d, s2.b:%f\n",s2.a,s2.b);

}
```

`(*p1).a` is equivalent to `p1->a`

Pass by reference using structures (another equivalent)

innovate achieve lead

```
typedef struct{
   int a;
   float b;
} ST;

void swap(ST * p1, ST * p2){
    ST temp;
    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

```
int main() {

ST s1, s2;
s1.a=10; s1.b=10.555;
s2.a=3; s2.b=3.555;

printf("s1.a:%d, s1.b:%f\n",s1.a,s1.b);
printf("s2.a:%d, s2.b:%f\n",s2.a,s2.b);

swap(&s1, &s2);

printf("s1.a: %d, s1.b:%f\n",s1.a,s1.b);
printf("s2.a: %d, s2.b:%f\n",s2.a,s2.b);

}
```

innovate achieve lead

BITS Pilani

Pilani Campus

Pointers Revisited

Null Pointer

- Initialize a pointer variable to NULL when that pointer variable isn't assigned any valid memory address yet.
  - `int *p = NULL;`
- Check for a **NULL** pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer related code.
- **NULL** pointer is also useful in implementation of linked lists.
  - *Next to next module.*

Generic or Void Pointer

Do we need different types of pointers to store the addresses of the variable of different types?

No!

```
void *void_ptr;    int x = 5; float y = 2.5f;

void_ptr = &x;
printf("\n x=%d",*(int *)void_ptr);

void_ptr = &y;
printf("\n y=%f",*(float *)void_ptr);
```

Pointer to a Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers.

Eg. `int **ptr;`

Example

```
int main () {
   int  var, *ptr, **pptr;

   var = 3000;
   ptr = &var;
   pptr = &ptr;

   printf("Value of var = %d, *ptr = %d, **pptr = %d\n",
   var,*ptr,**pptr );

   return 0;
}
Output:
Value of var = 3000, *ptr = 3000, **pptr = 3000
```

"pointer to a pointer" is useful in creating 2-D arrays with dynamic memory allocation (next module)

Review Question

```
int main(){
    int (*x1)[3];
    int y[2][3]={{1,2,3},{4,5,6}};
    x1 = y;
    for (int i = 0; i<2; i++)
      for (int j = 0; j<3; j++)
        printf("\n The X1 is %d and Y is %d",*(*(x1+i)+j), y[i][j]);
        // printf("\n The X1 is %d and Y is %d", x1[i][j], y[i][j]);
        // would also work
    return 0;
}
```

Output:

```
The X1 is 1 and Y is 1
The X1 is 2 and Y is 2
The X1 is 3 and Y is 3
The X1 is 4 and Y is 4
The X1 is 5 and Y is 5
The X1 is 6 and Y is 6
```

Review Question

```
int main()
{
    int arr[3][4] = {{1, 2, 3, 4},{5, 6, 7, 8},{9, 10, 11, 12}};
    int i = 1,j = 2;

    printf("\n Data at *(arr+i)+j =  %d",*(*(arr+i)+j));
    printf("\n Data at *(arr+i+j) =  %d",*(*(arr+i+j)));

    return 0;
}
```

Output:

```
 Data at *(arr+i)+j =  7
 Data at *(arr+i+j) =  1970957920
```

Review Question

```
int main()
{
    int x[] = {10,12,14};
    int *y, **z;

    y = x;
    z = &y;

    printf("x = %d, y = %d, z = %d\n", x[0], *(y+1), *(*z+2));
    printf("x = %d, y = %d, z = %d\n", x[0], *y+1, **z+2);

    return 0;
}
```

Output:

```
x = 10, y = 12, z = 14
x = 10, y = 11, z = 12
```

innovate achieve lead

BITS Pilani

Pilani Campus

Thank you

Q & A