Physical Memory And Addresses

- In a typical modern computer, a physical memory can be divided into two main deferent types:

1. Registers.
   - considered to be at the top level of Memory-Hierarchy.
   - provide the fastest way to access data.
   - directly accessed by the CPU
   - belongs to the CPU and typically located near (inside/behind/ above) it.
   - perform specialized functions e.g. stores next machine code instruction to be executed.
   - typically addressed by mechanisms other than main memory.
   - registers may be numbered or have arbitrary names depending on the processor design.
   - are limited on size (16, 32, 64… bits depending on the computer hardware).

2. Memory - primarily is of three types:

✓ Cache Memory
   - acts as a buffer between the CPU and main memory.
   - used to hold those parts of data and program which are most frequently used by CPU.
   - faster than main memory
   - required less access time than main memory.
   - has limited capacity
   - very expensive.

✓ Main Memory (primary memory)
   - holds data and instructions on which computer is currently working.
   - data is lost once power is switched off.
   - faster than secondary memory.

✓ Secondary Memory (external memory)
   - used for data storage (not lost once power is switched off)
   - e.g. CD-ROM, DVD HDs

Pointers

- a pointer is a special variable whose value is an address of another variable.

- C allows declaration of pointers to any variable type.

- a pointer must be associated to a particular type.

- a pointer must be initialized before using it.
- ✓ If a pointer is declared but not initialized – then it will contain garbage value of a location that might be in use by other application.

- pointers considered to be a powerful programming tool:
- ✓ programming becomes much easier.
- ✓ a pointer produces more efficient code.
- ✓ passing data location to a function much easier than copying every element of the data.
- ✓ support powerful use of dynamic memory allocation.

- pointers are explicitly used in C (arrays, functions and structures).

- in C there is a very close connection between pointers and arrays - Array name is the address of the first element in the array.

- miss understanding of pointers properly means losing all the power and flexibility C provides.

& is used to get location in memory
* is used to get value of a given location

pa = &a → bff8c0b8
*pa = *(&a) = *(bff8c0b8) → 3

```
#include <stdio.h>
int main()
{
   int a = 2;
   int *pa;
   pa = &a;
   *pa = 3;
   printf("%d %p %d %d %p\n", a, &a, *(&a) ,*pa, pa);
   return 1;
}
```

'pa' contains address of 'a'

'a' contains 3

value in memory

location in memory

output
3 bff8c0b8 3 3 bff8c0b8

How to correctly print an address (pointer)

NOTE

To print an address , some times I use %p without casting to void* or even using %x instead, which may generate a warning that you should avoid (this is only to make code more readable in the lectures).

Use “%p” to print an address. Cast to “void *”(if not then warning will be generated)

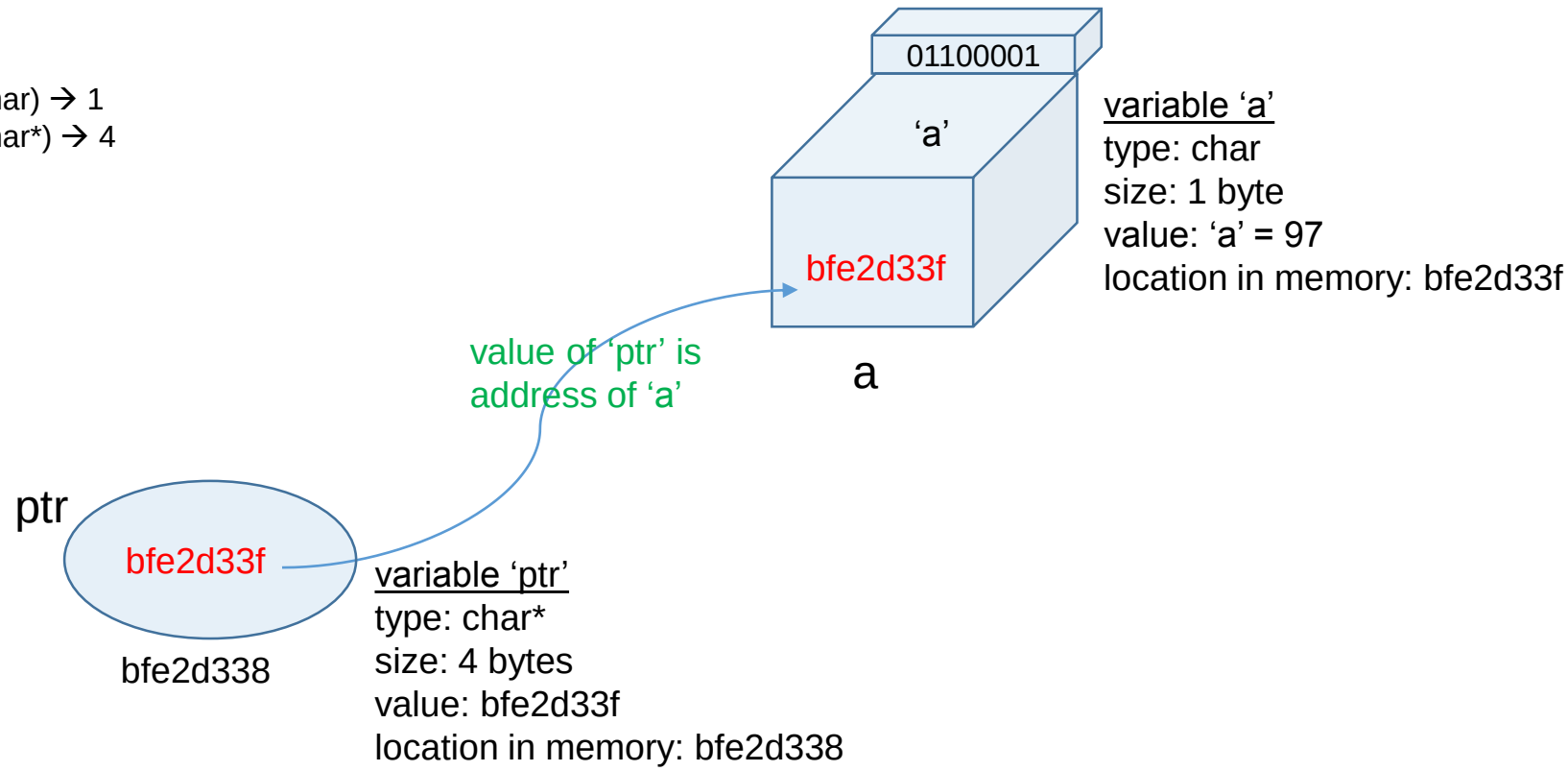
```
#include<stdio.h>
int main()  {
   int a = 3;
   int *pa = &a;

   char c = 'A';
   char *pc = &c;

   double d = 1.5;
   double *pd = &d;

   printf("%p\n", (void*)pa);
   printf("%p\n", (void*)pc);
   printf("%p\n", (void*)pd);

return 1;
}
```

Output:
0xbfc9fccc
0xbfc9fcdf
0xbfc9fcc0

```
char a = 'a';
char *ptr = &a;
printf("%p\n", ptr);
printf("%c\n",*ptr);
```

output
0xbfe2d33f
a

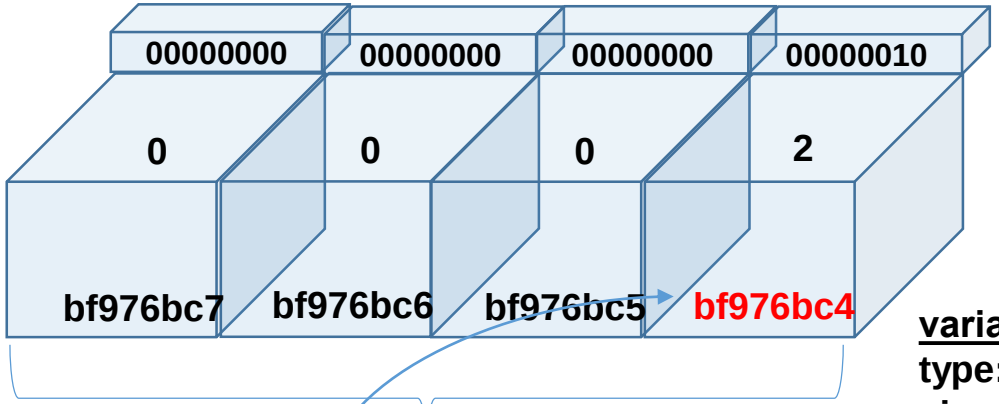
***
sizeof(char) → 1
sizeof(char*) → 4



```
int a = 2;
int *ptr = &a;

printf("%d %p\n", a, &a);
printf("%p %p\n", ptr, &ptr);
```

output
2 bf976bc4
bf976bc4 bf976bc8

***
sizeof(int) → 4
sizeof(int*) → 4



variable 'a'
type: int
size: 4 bytes
value: 2
location in memory: bf976bc4

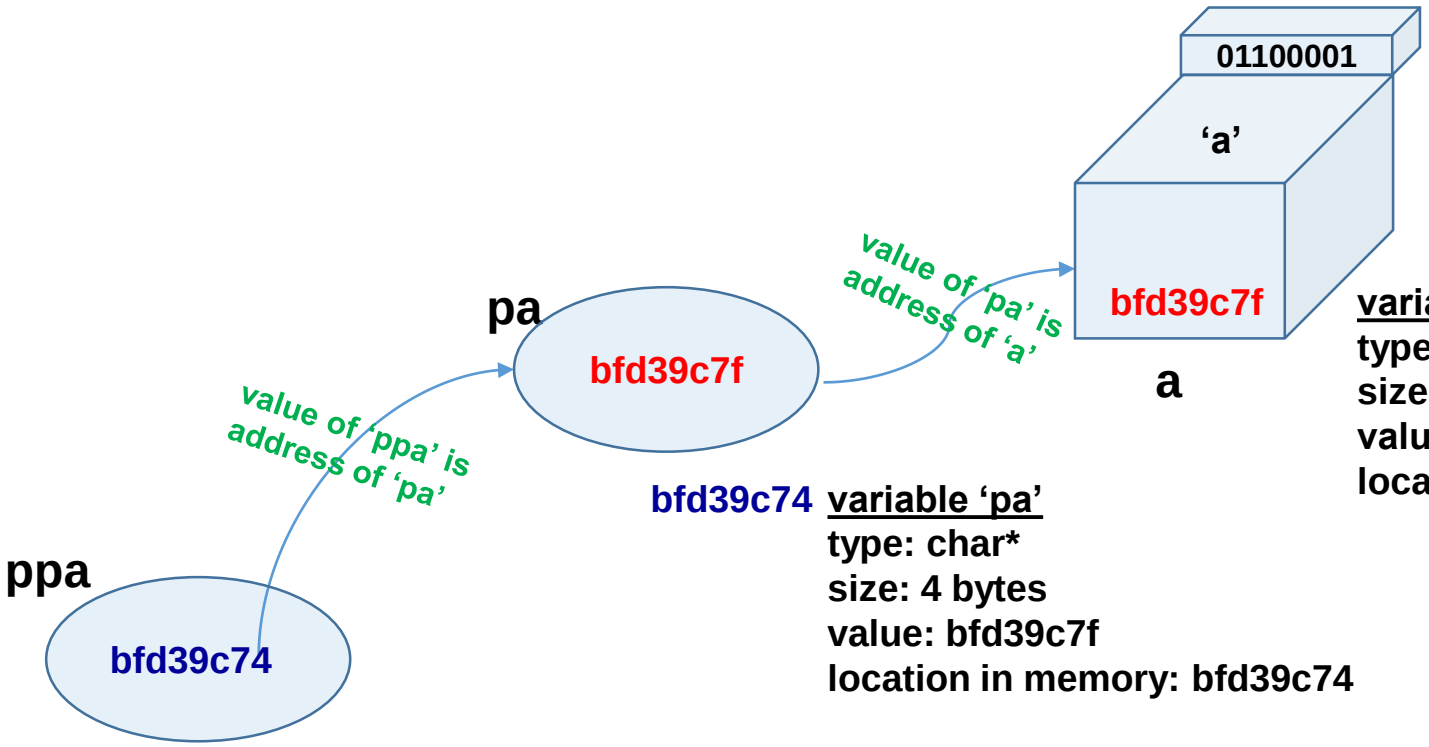
value of 'ptr' is address of 'a'

ptr
bf976bc4
bf976bc8

variable 'ptr'
type: int*
size: 4 bytes
value: bf976bc4
location in memory: bf976bc8

Pointer to Pointer

```
char a = 'a';
char *pa = &a;
char **ppa = &pa;
```



variable 'a'
type: char
size: 1 byte
value: 'a' = 97
location in memory: bfd39c7f

bfd39c74 **variable 'pa'**
type: char*
size: 4 bytes
value: bfd39c7f
location in memory: bfd39c74

bfd39c78 **variable 'ppa'**
type: char**
size: 4 bytes
value: bf976bc4
location in memory: bfd39c78

Pointer to Pointer

```
#include <stdio.h>
int b =3;
void f1(int* pa) {
    pa = &b;
}
void f2(int** ppa) {
    *ppa = &b;
}
int main()  {
    int a=2, *pa, **ppa;
    pa = &a;
    ppa = &pa;
    printf("%d %d %d %p %p %p %p %p\n", a,*pa, **ppa, &a, pa, *ppa, &pa, ppa);
    f1(pa);   printf("%d %d %d %p %p %p %p %p\n", a,*pa,**ppa, &a,pa, *ppa, &pa,ppa);
    f2(ppa); printf("%d %d %d %p %p %p %p %p\n", a,*pa,**ppa, &a,pa,*ppa, &pa,ppa);
    return 1;
}
```

a	*pa	**ppa	&a	pa	*ppa	&pa	ppa
2	2	2	bfe1d864	bfe1d864	bfe1d864	bfe1d868	bfe1d868
2	2	2	bfe1d864	bfe1d864	bfe1d864	bfe1d868	bfe1d868
2	3	3	bfe1d864	804a014	804a014	bfe1d868	bfe1d868

Use pointers to update external variable

```
#include<stdio.h>
void sum(int x, int y, int* pc)
{
    *pc = x+y;
}

int main()
{
    int a=1;
    int b=2;
    int c;

    sum(a,b, &c);

    printf("%d\n", c);

    return 1;
}
```

output
3

Returning Pointer To a variable From a Function

version1

```
int* sum(int a, int b)
{
  int c;
  c = a+b;

  return &c;
}
```

warning: function returns address of local variable

version2

```
int c;
int* sum(int a, int b)
{
  c = a+b;

  return &c;
}
```

It is ok to return a pointer to a global variable

version3

```
int* sum(int a, int b)
{
  static int c;
  c = a+b;

  return &c;
}
```

It is ok to return a pointer to a static variable

version4

```
include <stdio.h>
#include <stdlib.h>
int* sum(int a, int b)
{
  int *c;
  c = (int*)malloc(sizeof(int));
  *c = a+b;
  return c;
}
```

It is ok to return a pointer to a dynamically allocated variable

NULL Pointer

- NULL Pointer is a pointer which is pointing to an empty location (to nothing) i.e. a pointer that does not point to any object.
- Pointer which is initialized with NULL value is considered as NULL pointer.
- The NULL pointer is a constant with a value of zero.
- it is considered a good programming practice to initialized a pointer that currently is not in use, with NULL.

```
#include<stdio.h>
int main() {
  printf("%x\n", NULL);
  printf("%p\n", NULL);
  return 1;
}
```

Output:
0
(nil)

```
#include<stdio.h>
int main() {
   int *p = NULL;
   printf("%x\n", p);
   printf("%p\n", (void*)p);
   printf("%x\n", *p);
  return 1;
}
```

Output:
0
(nil)
Segmentation fault (core dumped)

- All bellow declarations are null pointers:

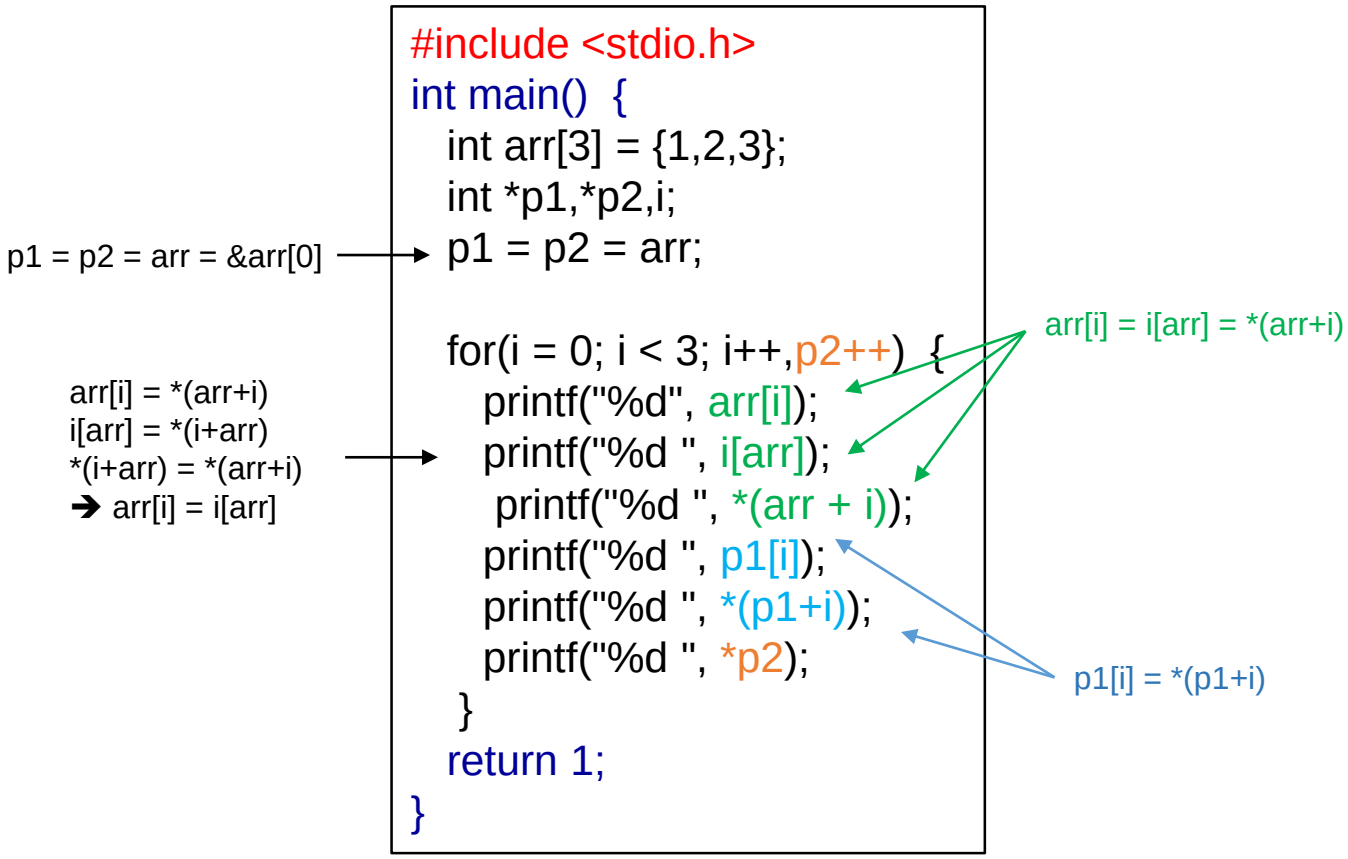
✓ float *ptr = (float *)0; → ptr = 0
✓ char *ptr = (char *)0; → ptr = 0
✓ double *ptr = (double *)0; → ptr = 0
✓ char *ptr = '\0'; → ptr = 0
✓ int *ptr = NULL; → ptr = 0

Pointers and Arrays

name of array is a **constant** variable which it's value is the **address** of the first element of the array

```
#include<stdio.h>
int main()
{
  int arr[3] = {1,2,3};
  int *pArr = arr;

printf("%p %p %p %p %p %p %p\n",arr, &arr, &arr[0], arr+1, &arr[1], arr+2, &arr[2]);
printf("%p %p %p %p %p %p %p\n",pArr,&pArr,&(*(pArr+0)),pArr+1,&(*(pArr+1)),pArr+2 ,
                                                                    &(*(pArr+2)));

return 1;
}
```

```
arr          &arr         &arr[0]       arr+1        &arr[1]       arr+2        &arr[2]
-------------------------------------------------------------------------------------------------
0xbfe56a10 0xbfe56a10  0xbfe56a10   0xbfe56a14   0xbfe56a14   0xbfe56a18   0xbfe56a18

pArr         &pArr        &(*(pArr+0))  pArr+1        &(*(pArr+1))  pArr+2        &(*(pArr+2))
-------------------------------------------------------------------------------------------------
0xbfe56a10  0xbfe56a1c  0xbfe56a10   0xbfe56a14   0xbfe56a14   0xbfe56a18   0xbfe56a18
```



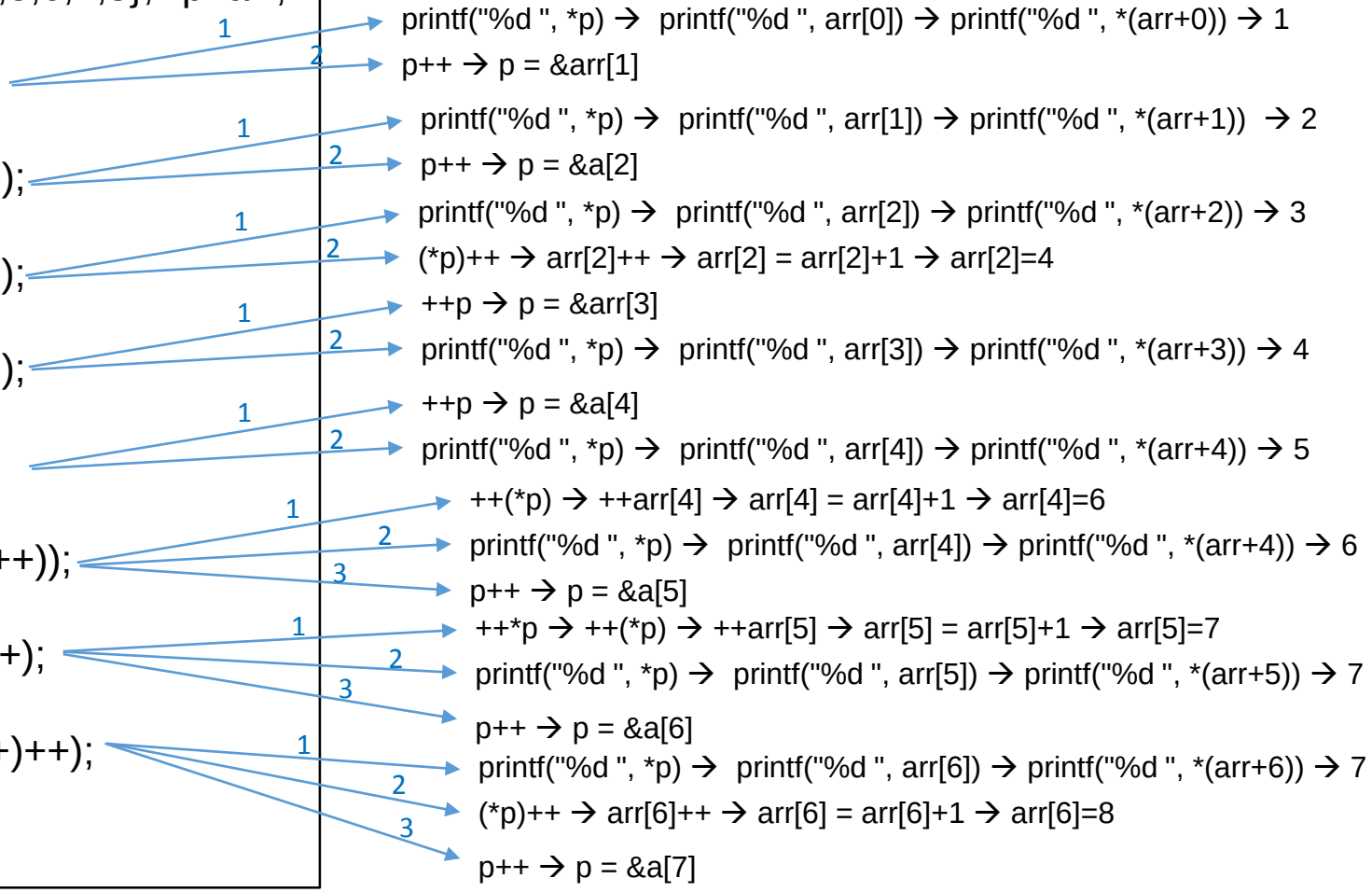
arr[i]	i[arr]	p1[i]	*(arr+i)	*(p1+i)	*p2	arr	p1	p2
1	1	1	1	1	1	bfb50658	bfb50658	bfb50658
2	2	2	2	2	2	bfb50658	bfb50658	bfb5065c
3	3	3	3	3	3	bfb50658	bfb50658	bfb50660

pArr is a pointer to an array of size 3 integers.
➔ each jump of pArr is 3*4 = 12 bytes.

```
#include<stdio.h>
int main()  {
   int arr[3] = {1,2,3};
   int (*pArr)[3] = &arr;

  printf("%d %d %d\n",arr[0],arr[1],arr[2]);
  printf("%d %d %d\n",(*pArr)[0],(*pArr)[1],(*pArr)[2]);
  printf("%d %d %d\n",**pArr,*(*pArr+1),*(*pArr+2));

  printf("%p %p %p %d\n", &arr[0],&arr[1],&arr[2],&arr[3]-&arr[0]);
  printf("%p %p %p %d\n",arr,pArr,pArr+1,*(pArr+1)-arr);

return 1;
}
```

```
1 2 3
1 2 3
1 2 3
0xbffedb90 0xbffedb94 0xbffedb98 2
0xbffedb90 0xbffedb90 0xbffedb9c 3
```

```
#include <stdio.h>
int main()
{
   int arr[8] = {1,2,3,4,5,6,7,8}, *p=arr;

   printf("%d ", *p++);

   printf("%d ", *(p++));

   printf("%d ", (*p)++);

   printf("%d ", *(++p));

   printf("%d ", *++p);

   printf("%d ", ++(*p++));

   printf("%d ", ++*p++);

   printf("%d\n", (*p++)++);
   return 1;
}
```



Output:

1 2 3 4 5 6 7 7

➔arr = {1,2,4,4,6,7,8,8}

Passing Array To a Function

version1

```
int max(int a[], int size)
{
  int i,max=a[0];
  for(i=0; i < size; i++)
  {
     (max < a[i]) ? (max = a[i]) : 1;
  }
  return max;
}
```

version2

```
int max(int *pArr, int size)
{
  int i, max=pArr[0];
  for(i=0; i < size; i++)
  {
     (max < pArr[i]) ? (max = pArr[i]) : 1;
  }
  return max;
}
```

- ✓ Passing an array to a function is by reference only i.e. passing the address of the first element of the array.
- ✓ → int a[] = *PArr

version3

```
int max(int *pArr, int size)
{
  int max=*pArr;
  int* stop = pArr + size;
  while(pArr < stop)
  {
     (max < *pArr) ? (max = *pArr) : 1;
     pArr++;
  }
  return max;
}
```

Returning Array From a Function

version1

```
#define size 3
int[] sum2Arrays(int a1[], int a2[])
{
  int a3[size], i;

  for(i=0;i<size;i++)
    a3[i] = a1[i] + a2[i];

  return a3;
}
```

error

version2

```
#define size 3
int a3[size];
int[] sum2Arrays(int a1[], int a2[])
{
  int i;

  for(i=0;i<size;i++)
    a3[i] = a1[i] + a2[i];

  return a3;
}
```

error

returning an array from a function is not allowed

Returning a Pointer To Array From a Function

version1

```
#define size 3
int* sum2Arrays(int a1[], int a2[])
{
  int a3[size], i;

  for(i=0;i<size;i++)
    a3[i] = a1[i] + a2[i];

  return a3;
}
```

warning: function returns address of local variable

version2

```
#define size 3
int a3[size];
int* sum2Arrays(int a1[], int a2[])
{
  int i;

  for(i=0;i<size;i++)
    a3[i] = a1[i] + a2[i];

  return a3;
}
```

It is ok to return a pointer to a global array

Swap version

version1

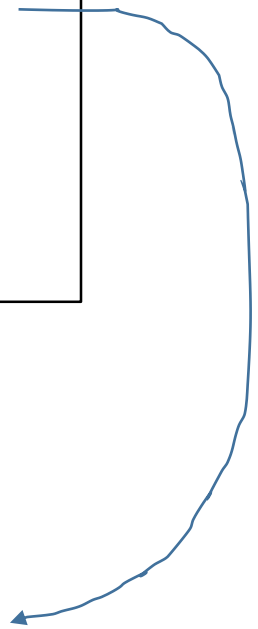
```
void swap(int *pa, int *pb)
{
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
```

```
int a=1, b=2;
swap(&a,&b) ;
```
→ a=2 , b=1

version2

```
void swap(int **ppa, int **ppb)
{
  int* tmp = *ppa;
  *ppa = *ppb;
  *ppb = tmp;
}
```

```
int a=1, b=2, *pa = &a, *pb = &b;
swap(&pa,&pb);
```
→ a=1 , b=2 , pa = &b , pb = &a

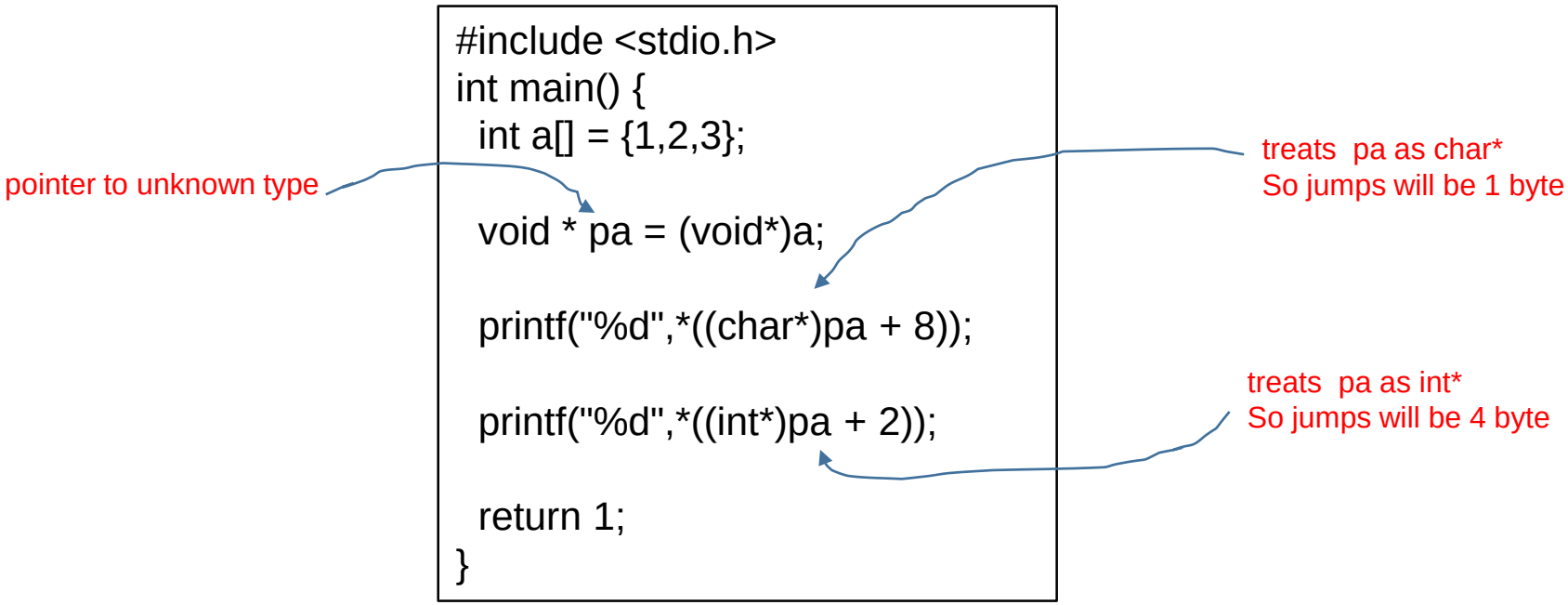


pointer to pointer

Passing a pointer to a function gives the access to a calling variable directly. This is often more efficient than passing a copy of the variable whose copy is placed in the run time stack

generic pointer type
void*

- ✓Generic pointer can point to any data type.
- ✓Uses void* to create generic pointer
- ✓Any pointer can be cast to void * and back again without losing information



warning: wrong type argument to increment

```
int main()
{
  int x = 0x01020304;
  void *p = &x;
  p++;
  printf("%d\n", *((char*)p));

   return 1;
}
```

Output:
3

```
int main()
{
  int x = 0x01020304;
  void *p = &x;
  printf("%p\n", p);
  printf("%p\n", (int*)p);
  printf("%p\n", (char*)p);
  printf("%x\n", *((int *)p));
  printf("%x\n", *((char *)p));
  printf("%x\n", *(((char *)p) +1));
  printf("%x\n", *(((char *)p) +2));
  printf("%x\n", *(((char *)p) +3));

   return 1;
}
```

Output:
0xbf9eaa88
0xbf9eaa88
0xbf9eaa88
1020304
4
3
2
1

Generic Swap version

```
#include <stdio.h>
void swap(void **pa, void **pb)
{
    void *temp;
    temp = *pa;
    *pa = *pb;
    *pb = temp;
}
int main()
{
   int a=2,b=3,*pa=&a,*pb=&b;
   char c1='a',c2='b',*pc1=&c1,*pc2=&c2;
   printf("%d %d %c %c\n", *pa,*pb,*pc1,*pc2);
   swap((void **)&pa,(void **)&pb);
   swap((void**)&pc1,(void**)&pc2);
   printf("%d %d %c %c\n", *pa,*pb,*pc1,*pc2);
 return 1;
}
```

Output:

2 3 a b

3 2 b a

compare s1 and s2 numerically

```
#include <stdio.h>
#include <stdlib.h>
int numcmp(char *s1, char *s2)
{
    double v1, v2;
    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
int main()
{
  printf("%d\n", numcmp("1","2"));
  printf("%d\n", numcmp("123","111"));
 return 1;
}
```

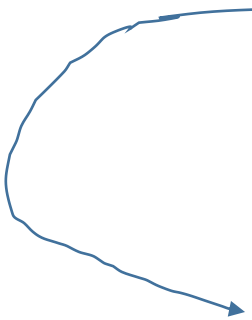
Output:

```
-1
1
```

Pointers and Constant variables

version1

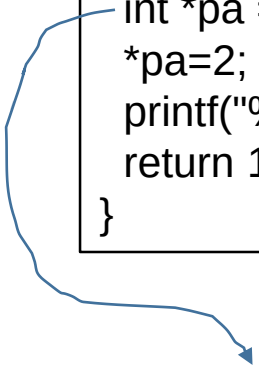
```
#include <stdio.h>
int main()
{
  const int a = 1;
  a=2;
  printf("%d\n",a);
  return 1;
}
```



error: assignment of read-only variable ‘a’

version2

```
#include <stdio.h>
int main()
{
  const int a = 1;
  int *pa = &a;
  *pa=2;
  printf("%d\n",a);
  return 1;
}
```



warning: initialization discards ‘const’ qualifier from pointer target type
→ a=2

Pointers and Array of Constants

version1

```
#include <stdio.h>
int main() {
 const int arr[3] = {1,2,3};
 arr[0] = 4;
 return 1;
}
```

error: assignment of read-only location ‘arr[0]’

version2

```
#include <stdio.h>

int main()
{
    const int arr[3] = {1,2,3};

    int* pArr = arr;

    pArr[0] = 4;
    pArr[1] = 5;
    *(pArr+2) = 6;

    return 1;
}
```

warning: initialization discards ‘const’ qualifier from pointer target type
→ arr[0]=4, arr[0]=5 , arr[0]=6

Pointer to Constant and Constant Pointer

We must initialize const pointer at the time of declaration. Otherwise it will cause error.

version1

```
#include <stdio.h>

int main()
{
   char arr[] = {'a','b','c','\0'};
   char* pArr = arr;
   while( *pArr != '\0' )
   {
      *pArr +=1;
      printf("%c ",*pArr);
      pArr++;
   }
   return 1;
}
```

Output:
b c d

version2

```
#include <stdio.h>

int main()
{
   char arr[] = {'a','b','c','\0'};
   const char* pArr = arr;
   while( *pArr != '\0' )
   {
      *pArr +=1;
      printf("%c ",*pArr);
      pArr++;
   }
   return 1;
}
```

Pointer to constant

error: assignment of read-only location '*pArr'

const char* → the 'const' is on the values 'pArr' points at.

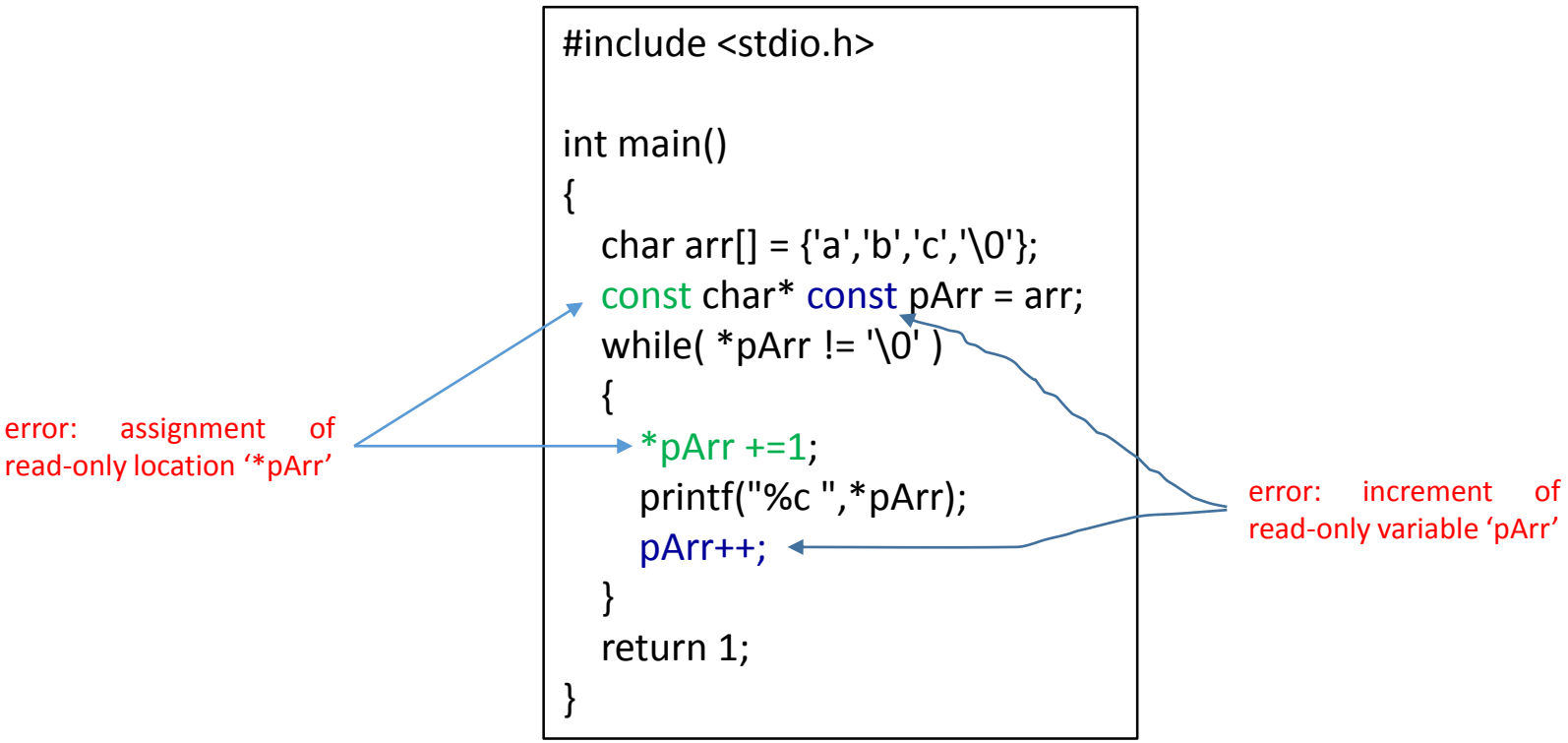
version3

```
#include <stdio.h>

int main()
{
   char arr[] = {'a','b','c','\0'};
   char* const pArr = arr;
   while( *pArr != '\0' )
   {
      *pArr +=1;
      printf("%c ",*pArr);
      pArr++;
   }
   return 1;
}
```

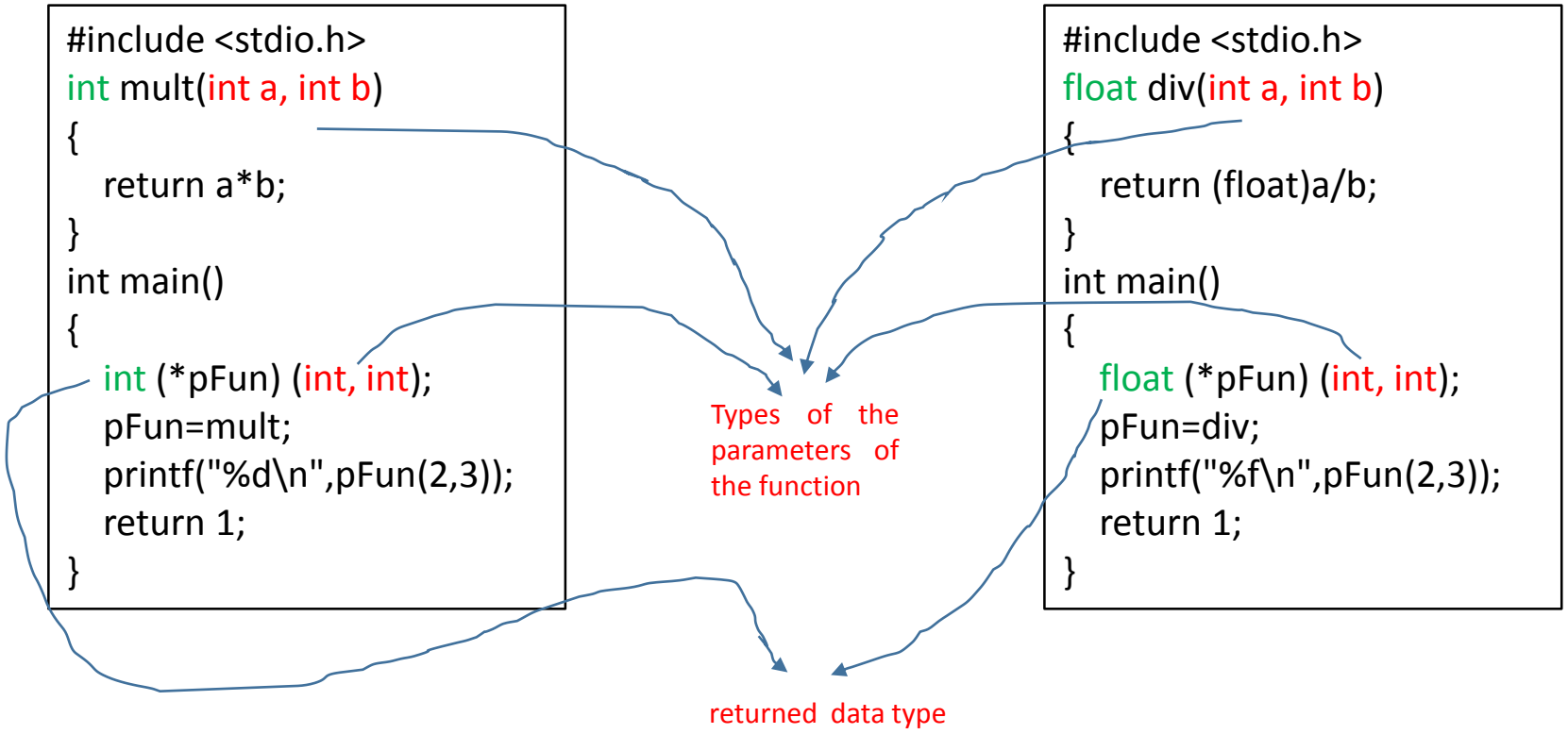
constant pointer

error: increment of read-only variable 'pArr'

char* const → the 'const' is on the pointer 'pArr'


#include <stdio.h>
int main()
{
char arr[] = {'a','b','c','\0'};
const char* const pArr = arr;
while(*pArr != '\0')
{
*pArr +=1;
printf("%c ",*pArr);
pArr++;
}
return 1;
}
error: assignment of read-only location '*pArr'
error: increment of read-only variable 'pArr'

Pointer to a Function



Pointer to a Function

```
#include <stdio.h>
int sum(int a, int b)
{
  return a+b;
}
int mult(int a, int b)
{
  return a*b;
}
int main()
{
  int (*pFun) (int, int);
  pFun=sum; printf("%d\n",pFun(2,3));
  pFun=mult; printf("%d\n",pFun(2,3));
  return 1;
}
```

'pFun' is a pointer to a function which receives 2 integers and return an integer.

Output:
5
6

Dynamic Memory Allocation

Dynamic Memory Allocation	
Function	Description
void *calloc(int n, int size)	allocates an array of **n**-elements each of which size in bytes will be **size**.
void free(void *address)	releases a block of memory block specified by address
void *malloc(int n)	allocates an array of **n**-bytes and leave them uninitialized.
void *realloc(void *address, int newsize)	re-allocates memory extending it up to **newsize**.

Dynamic allocation using malloc

allocate an array of char with 3 elements

Free the allocated memory

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  char *str=NULL;

  str = (char*)malloc(sizeof(char)*3);

  if(str != NULL){
     str[0]='a'; str[1]='b'; str[2]='\0';
  }

  free(str);

  return 1;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *arr;

  arr = (int*)malloc(sizeof(int)*3);

  if(arr != NULL){
     arr[0] =1;   arr[1] =2;   arr[2] =3;
  }

  free(arr);

  return 1;
}
```

In failure malloc will return null

Dynamic allocation using calloc

allocate an array with 3 elements. each element size is 1 byte.

Free the allocated memory

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  char *str;

  str = (char*)calloc(3,1);

  if(str != NULL){
      str[0]='a'; str[1]='b'; str[2]='\0';
  }

  free(str);

  return 1;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *arr;

  arr = (int*)calloc(3,4);

  if(arr != NULL){
    arr[0] =1;   arr[1] =2;   arr[2] =3;
  }

  free(arr);

  return 1;
}
```

In failure calloc will return null

Dynamic allocation using realloc

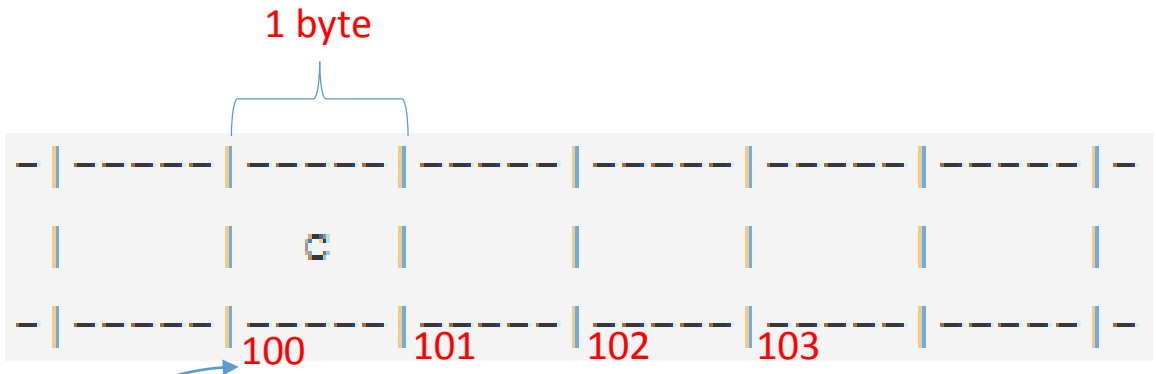
```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int *pArr;

   pArr = (int*) malloc(sizeof(int)*3);
   if(pArr != NULL) {
      pArr[0]=1;   pArr[1]=2;  pArr[2]=3;
   }

   pArr = (int*) realloc(pArr, sizeof(int)*5);
   if(pArr != NULL) {
      int i;
      pArr[3]=4;   pArr[4]=5;
      for(i=0; i<5 ; i++)
         printf("%d ",pArr[i]);
      putchar('\n');
   }

   free(pArr);

   return 1;
}
```

it is not guaranteed that the same address will be allocated for both blocks.

Output:
1 2 3 4 5

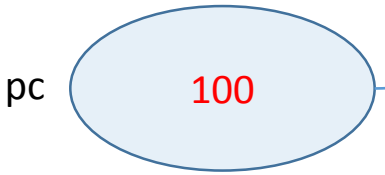
In failure realloc will return null

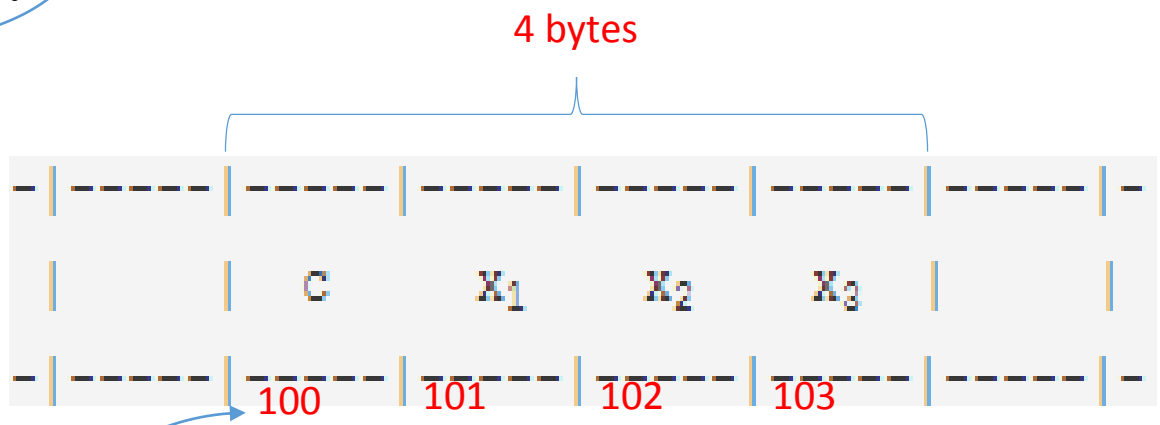
Pointer Casting




```
char c=5;
char* pc = &c;
```

printf("%d",*pc) → 5

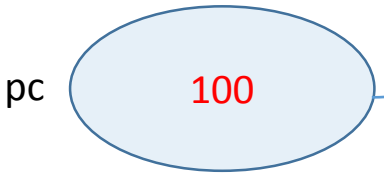





```
char c=5;
int* pc = (int*)&c;
```

printf("%d",*pc) → ?

What are the values of x1, x2 and x3 ?



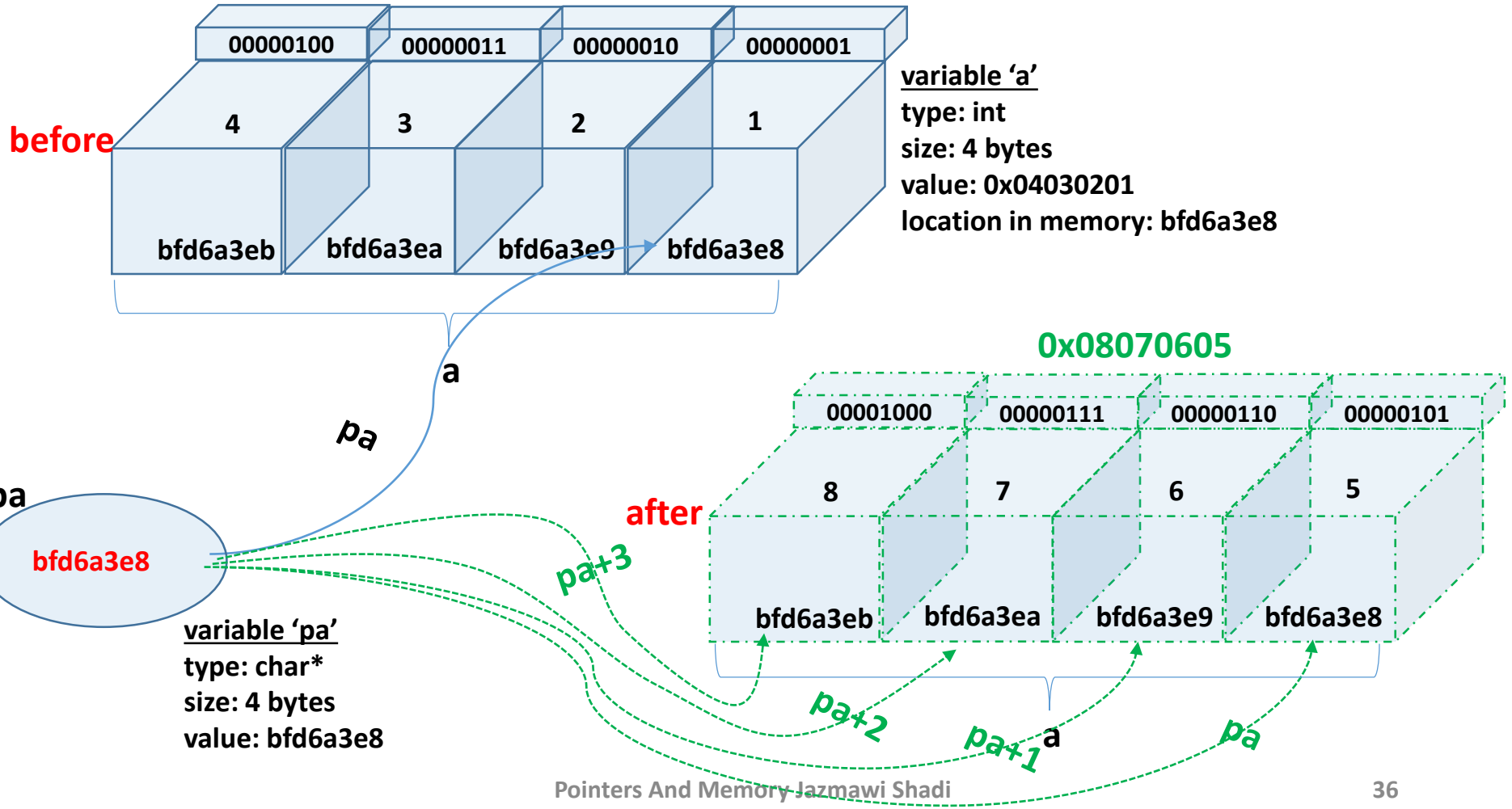
Little-Endian→int val =c * 2^0 + x_1 * 2^8 + x_2 * 2^{16} + x_3 * 2^{24}

Big-Endian→int val =c * 2^{24} + x_1 * 2^{16} + x_2 * 2^8 + x_3 * 2^0

Pointer Casting

```
int a = 0x04030201;
char *pa = (char*)&a;
*pa = 5;
*(pa+1) = 6;
*(pa+2) = 7;
*(pa+3) = 8;
```

***
sizeof(int) → 4
sizeof(char) → 1



variable 'a'
type: int
size: 4 bytes
value: 0x04030201
location in memory: bfd6a3e8

variable 'pa'
type: char*
size: 4 bytes
value: bfd6a3e8

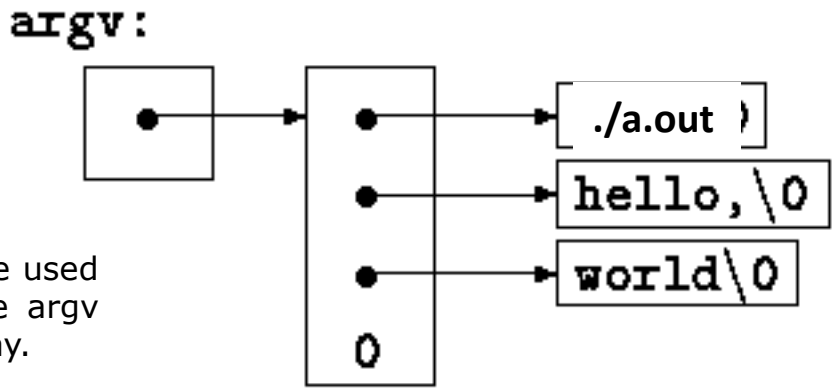
Command Line Arguments

argc: the number of command-line arguments the program was invoked with.

argv parameter is an array of string pointers

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

`argv[0]` is the name by which the program was invoked, so `argc` is at least 1



- ✓ each argv element can be used just like a string, or use argv as a two dimensional array.
- ✓ argv[argc] is a null pointer.

Command Line Arguments

```
student@ubuntu:~/Desktop/proj2$ ./a.out "i" "am" "student"
./a.out
i
am
student
student@ubuntu:~/Desktop/proj2$
```

version1

```
#include <stdio.h>
int main(int argc, char* argv[])
{
   printf("number of arguments:%d\n", argc);
   printf("Arg1-File name:%s\n", argv[0]);
   printf("Arg2:%s\n", argv[1]);
   printf("Arg3:%s\n", argv[2]);
   printf("Arg4:%s\n", argv[3]);

   return 1;
}
```

version2

```
#include <stdio.h>
int main(int argc, char** argv)
{
   printf("number of arguments:%d\n", argc);
   printf("Arg1-File name:%s\n", *argv);
   printf("Arg2:%s\n", *(argv+1));
   printf("Arg3:%s\n", *(argv+2));
   printf("Arg4:%s\n", *(argv+3));

   return 1;
}
```

Output:

number of arguments: 4 ← argc
Arg1-File name: ./a.out ← argv[0]
Arg2: i ← argv[1]
Arg3: am ← argv[2]
Arg4: student. ← argv[3]

```
student@ubuntu:~/Desktop/proj2$ ./a.out "i" "am" "student"
./a.out
i
am
student
student@ubuntu:~/Desktop/proj2$
```

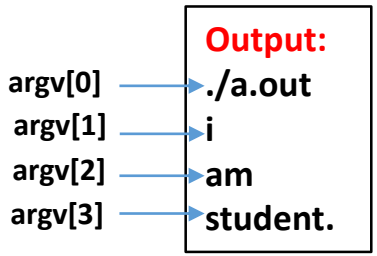
version1

```
#include <stdio.h>
int main(int argc, char* argv[])
{
   int i;
   for(i=0; i<argc; i++)
   {
      printf("%s\n",argv[i]);
   }

   return 1;
}
```

The first version treats argv as an array of character pointers:

version2

```
#include <stdio.h>
int main(int argc, char** argv)
{
   while(argc--)
   {
      printf("%s\n",*argv++);
   }

   return 1;
}
```



```
student@ubuntu: ~/Desktop/proj2
student@ubuntu:~/Desktop/proj2$ ./a.out "2" "+" "3"
2 + 3 = 5
student@ubuntu:~/Desktop/proj2$ ./a.out "2" "-" "3"
2 - 3 = -1
student@ubuntu:~/Desktop/proj2$ ./a.out "2" "*" "3"
2 * 3 = 6
student@ubuntu:~/Desktop/proj2$ ./a.out "2" "/" "3"
2 / 3 = 0
student@ubuntu:~/Desktop/proj2$
```

array of pointers to functions

argv[2][0]

```
#include <stdio.h>
#include <stdlib.h>
int sum (char* a, char* b) {
   return atoi(a) + atoi(b);
}
int sub (char* a, char* b) {
   return atoi(a) - atoi(b);
}
int mult (char* a, char* b){ return atoi(a) * atoi(b); }
int division(char* a, char* b){ return atoi(a) / atoi(b); }
int main(int argc, char* argv[]) {
   int (* pf[]) (char*,char*) = {sum, sub, mult, division};
   char* x = argv[1];
   char* y = argv[3];
   switch(*argv[2]) {
     case '+': printf("%s %s %s = %d\n", x, argv[2], y, pf[0](x,y));
               break;
     case '-': printf("%s %s %s = %d\n", x, argv[2], y, pf[1](x,y));
               break;
     case '*': printf("%s %s %s = %d\n", x, argv[2], y, pf[2](x,y));
               break;
     case '/': printf("%s %s %s = %d\n", x, argv[2], y, pf[3](x,y));
               break;
   }
   return 1;
}
```


END