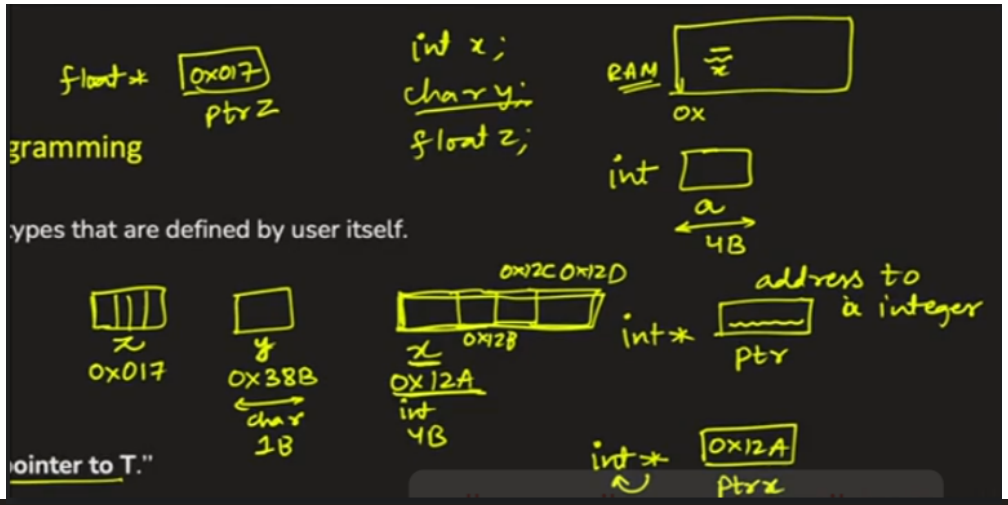
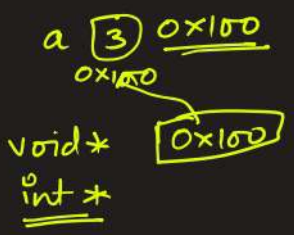


C Programming Lecture 8

Sunday, 16 June 2024 6:02 PM



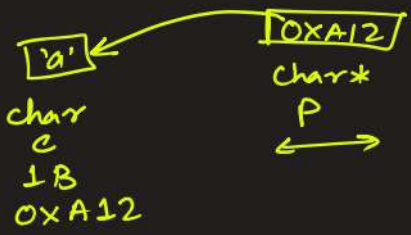
Pointers & Arrays in C programming

User-Defined data types: Data types that are defined by user itself.

- Arrays ✓
- Pointers ✓

chac

Pointers in C

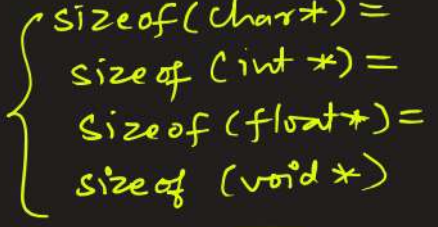


For any type **T**, **T*** is the type "**pointer to T**."

That is, a variable of type **T*** can hold the address of an object of type **T**.

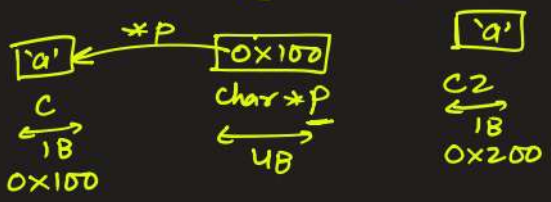
For example:

```
char c = 'a';
char* p = &c; // p holds the address of c; & is the address-of operator
```



Derefrencing or Indrection

Referring to the object pointed to by the pointer




```
char c = 'a';
char* p = &c; ✓
char c2 = *p; // c2 == 'a'; * is the dereference operator
```

The object pointed to by p is c, and the value stored in c is 'a', so the value of *p assigned to c2 is 'a'

In a 32-bit system, the pointer size is typically 4 bytes, and in a 64-bit system, it is usually 8 bytes

void*

- In low-level code, we occasionally need to store or pass along an address of a memory location without actually knowing what type of object is stored there
- A **void*** is used for that, read as 'pointer to an object of unknown type."
- A pointer to any type of object can be assigned to a variable of type void*
- A void* can be assigned to another void*

```
int* pi;
void* pv = pi; // allowed
```

```
#include<stdio.h>
int main(){
    int a = 3;
    void* p = &a;
    printf("%d", *(int*)p);
    return 0;
}
```

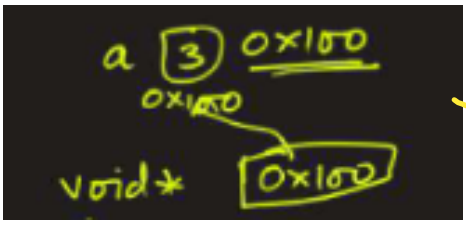
```
#include <stdio.h>

int main() {
    int a = 3;
    int* p = &a;
    int* q = p;
    printf("%d", *q);
    return 0;
}
```

```
#include <stdio.h>

int main() {
    int a = 3;
    int* p = &a;
    int* q = p;
    printf("%u", q);
    printf("%u", p);
    return 0;
}
```

p & q pointing to same address



when we use void pointer compiler doesn't know how many bytes to read

```
#include <stdio.h>

int main() {
    int a = 3;
    void* p = &a;
    printf("%d", *p);
    return 0;
}
```

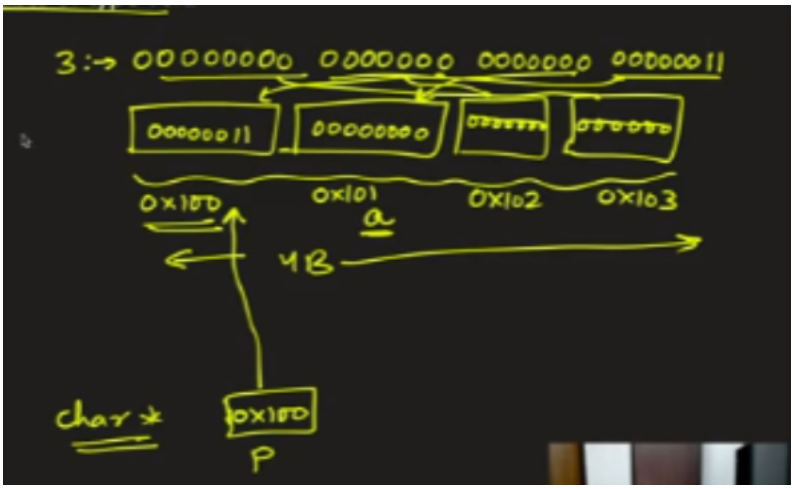
not work

```
#include <stdio.h>

int main() {
    int a = 3;
    void* p = &a;
    printf("%d", *(int*)p);
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    int a = 3;
    void* p = &a;
    printf("%d", *(char*)p);
    return 0;
}
```



Constant pointers

A constant pointer in C cannot change the address of the variable to which it is pointing, i.e., the address will remain constant. Therefore, we can say that if a constant pointer is pointing to some variable, then it cannot point to any other variable.

```
<type of pointer> *const <name of pointer>;
```

```c
#include<stdio.h>
int main() {
    int a = 1;
    int b = 2;
    int* const ptr;
    ptr = &a; // Not allowed
    ptr = &b; // Not allowed
    printf("Value of ptr is :%d",*ptr);
    return 0;
}
```

int * const ptr ✓ — const ptr to integer

(const int)* ptr ✓ — Pointer to const integer

int a; — 3 0x10 — a

int * const ptr — 0x10 — ptr

Pointer to a constant

A pointer to constant is a pointer through which the value of the variable that the pointer points cannot be changed. The address of these pointers can be changed, but the value of the variable that the pointer points cannot be changed.

```
const <type of pointer>* <name of pointer>
```

```c
#include<stdio.h>
int main(){
    int a = 1;
    int b = 2;
    const int* ptr;
    ptr = &a; // allowed
    ptr = &b; // allowed
    *ptr = 3; // not allowed
    b = 3; // allowed
    printf("Value of ptr is :%d", *ptr);
    return0;
}
```

```c
#include <stdio.h>

int main() {
    const int a = 3, b = 5
    const int* const ptr = &a;
    ptr = &b; // Not allowed
    *ptr = 9; // Not allowed

    return 0;
}
```

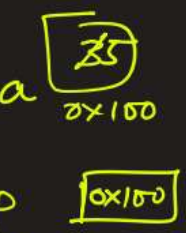
best practise
const int a = 3
const int* const ptr = &a

```
int a = 3;
int * p = &a;
*p = 5;
printf("%d", a); ← 5
printf("%d", *p); ← 5
```

```
int a = 3;
const int * p = &a;
a = 5; ✓ allowed
*p = 7; ✗ NOT allowed.
```

```
const int a = 3;
Const int * p = &a;
*p = 5; ✗ Not allowed
printf("%d", *p) ← 3
```

```
const int a = 3;
int * p = &a
a = 5; ✗ NOT allowed
*p = 5; ✓ allowed
printf("%d", a) ← 5
printf("%d", *p) ← 5
```

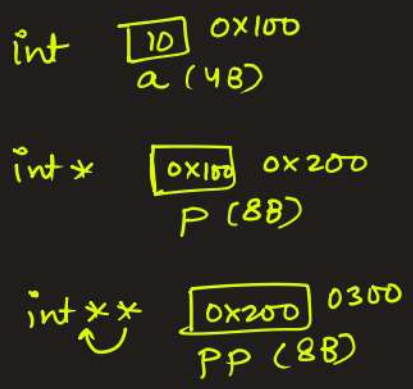


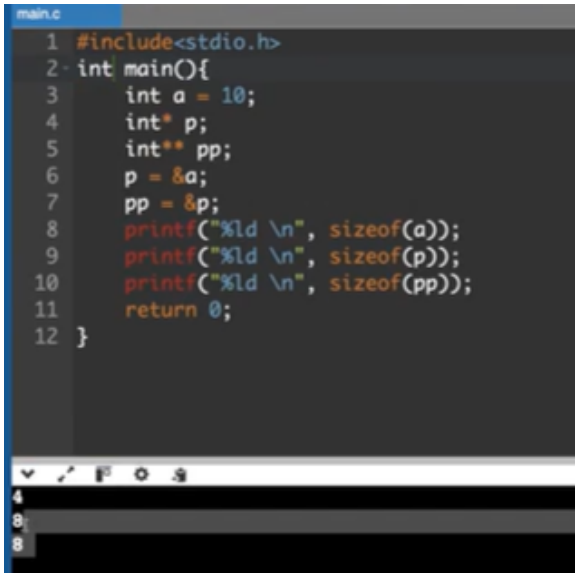
security issue bec. the pointer change the value which declare const
eg: cont int a = 3; after pointer p go to address and change the value to 5
now a = 5; *p = 5

Pointer to a pointer

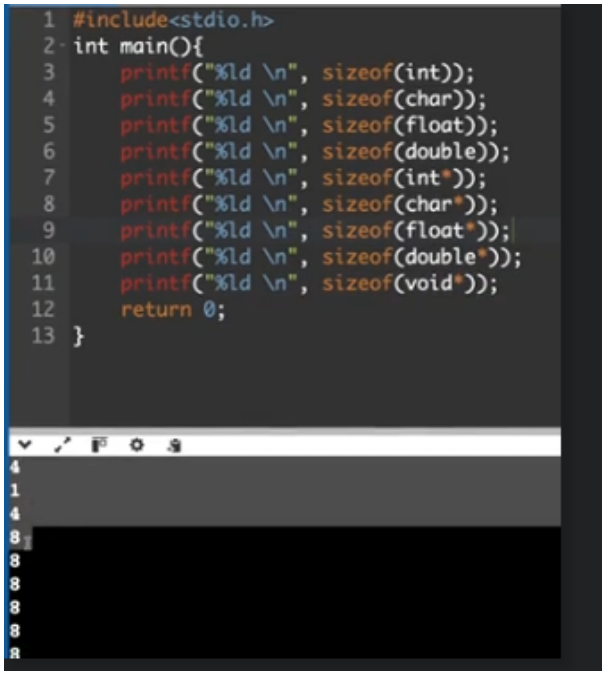
In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer.

```c
#include<stdio.h>
void main(){
    int a = 10;
    int* p;
    int** pp;
    p = &a;
    pp = &p;
    printf("address of a: %u\n",p);
    printf("address of p: %u\n",pp);
    printf("value stored at p: %d\n",*p);
    printf("value stored at pp: %d\n",**pp);
    return 0;
}
```





pointer 8 bytes bec.
online compiler 64 bit



```c
#include<stdio.h>
int main(){
    int a = 3;
    int* p = &a;
    printf("%ld \n", p);

    p++;
    printf("%ld \n", p);
    return 0;
}
```

```
|              |      int *
|              long int
|             %ls
140736321797052
140736321797056
```

Pointer arithmetic

We can perform arithmetic operations on the pointers like addition, subtraction, etc.
However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer.
In pointer-from-pointer subtraction, the result will be an integer value.

int ** p;
p++;

Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

4B int* p;
p = p+1;

p = p + number

char * p;
p = p+1;

```
new_address= current_address + (number * size_of(data type))
new_address= current_address - (number * size_of(data type))
```

p = p - number

```c
#include<stdio.h>
int main(){
    int a = 3; // 4B
    char b = 'c'; // 1B
    double c = 3.141; // 8B

    int* p1 = &a;
    char* p2 = &b;
    double* p3 = &c;

    printf("Address in p1: %ld \n", p1);
    printf("Address in p2: %ld \n", p2);
    printf("Address in p3: %ld \n", p3);

    p1++;
    p2++;
    p3++;

    printf("Address in p1: %ld \n", p1); // p1+4
    printf("Address in p2: %ld \n", p2); // p2+1
    printf("Address in p3: %ld \n", p3); // p3+8

    return 0;
}
```

```
|                                long int
Address in p1: 140726298833060
Address in p2: 140726298833059
Address in p3: 140726298833064
Address in p1: 140726298833064
Address in p2: 140726298833060
Address in p3: 140726298833072
```

```c
#include<stdio.h>
int main(){
    int a = 3; // 4B
    char b = 'c'; // 1B
    double c = 3.141; // 8B

    int* p1 = &a; // 8B
    char* p2 = &b; // 8B
    double* p3 = &c; // 8B
    char** p4 = &p2; // 8B

    printf("Address in p1: %ld \n", p1);
    printf("Address in p2: %ld \n", p2);
    printf("Address in p3: %ld \n", p3);
    printf("Address in p4: %ld \n", p4);

    p1 = p1-10;
    p2++;
    --p3;
    p4 = p4+100;

    printf("Address in p1: %ld \n", p1); // p1-40
    printf("Address in p2: %ld \n", p2); // p2+1
    printf("Address in p3: %ld \n", p3); // p3-8
    printf("Address in p4: %ld \n", p4); // p4+800

    return 0;
}
```

all pointer in 64 bit takes
8 bytes, int, char, double