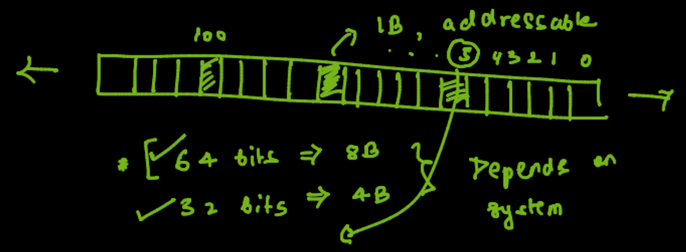
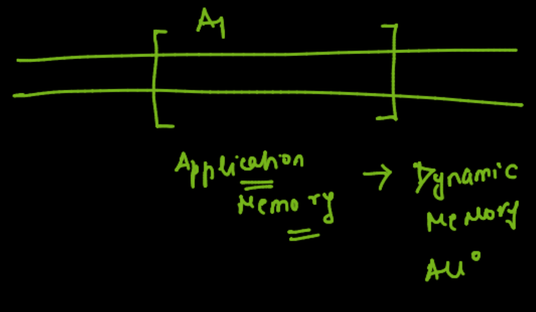




1 GB ⇒ 2^{30} B

64 bits ⇒ 2^{64} unique

2^{64} B ×

2^{10} B → 1 KB

2^{20} B → 1 MB

2^{30} B → 1 GB

⋮

= int x = 10; ⇒ 4B

= char ch = 'A' ⇒ 1B

↳ Address of (&)

&x ← addr of x

&ch ← addr of ch

↳ Hexadecimal

= 0 to 9, (A) to (F) → 64 bits

10, 11, 12, 13, 14, 15

A B C D E F

16 = 2^4

F → (1111) 8421 = 15

A → (1010)$_{10}$

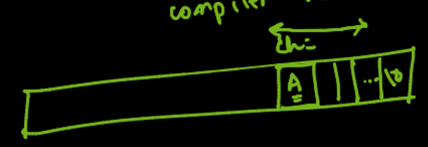
$(7)_{16}$ → (0111)

16 × 4 = 64

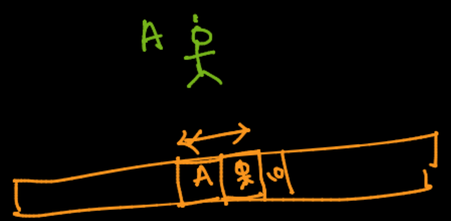
64 bits ⇒ 16 bits are reserved

4 bits ⇒ 12

↳ cout << &ch ;



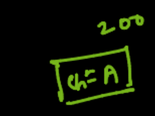
cout * ch



cout << &ch; ←
=

Pointers

↳ Variable that stores address of another variable



&x ← int x=10 ;

&ch ← char ch='A' ;

⇛ int * xptr ; // declare a pointer to an integer

⇛ xptr = &x ; // assignment

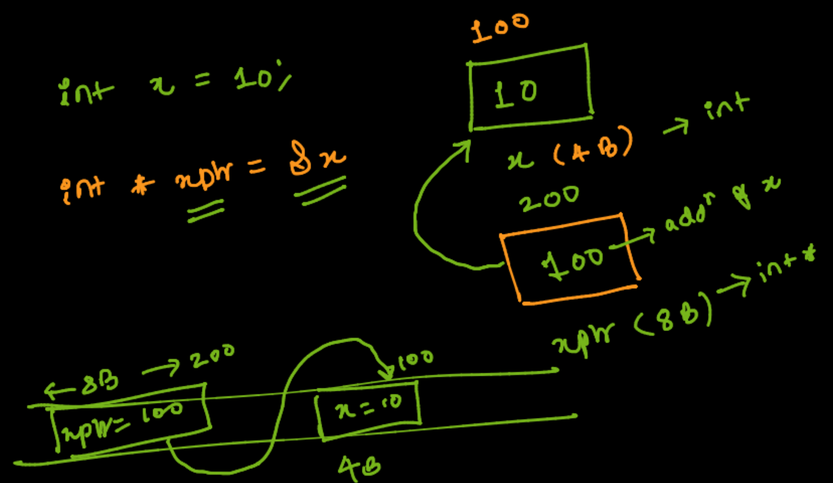
char * chptr ; // declaration

chptr = &ch ; // assignment

✓ int x ;
✓ x=10

] int x=10; ✓

[int * xptr = &x ;
char * chptr = &ch ;]



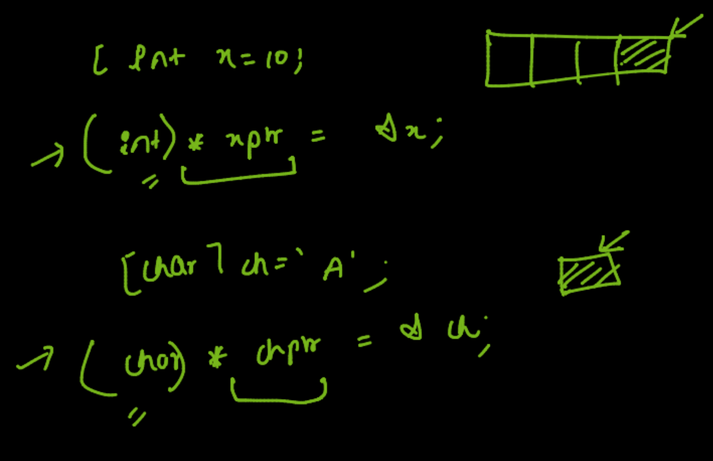
2-min talk

Why are pointers strongly typed ?

when they just store addr of a byte.

int x = 516;

512 + 4 = 516



(char *) (char)

int x = 10;

int * xptr = &x; → type-casting

char * chptr = (char *) &x; ✓

it is possible, but don't do it

double *
float *

cout << (int *) &ch;

[void *]

generic pointer type

```
int x = 10;
int *xptr = &x;
```


```
cout << x;      // 10
cout << &x;     // 100
cout << xptr;   // 100
```

dereferencing the pointer

```
int x = 10;
int *xptr = &x;
```

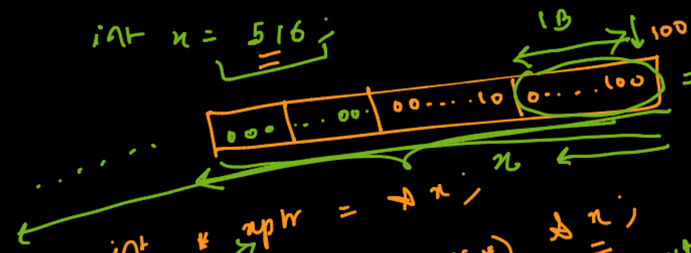
```
cout << *(xptr) << endl;
```

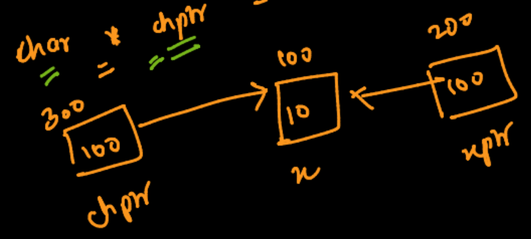
[value at]

value at 100 => 10

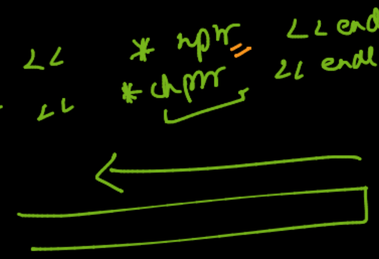
```
char ch = 'A';
char * chptr = &ch;
cout << (int*) chptr << endl; // 200
cout << *chptr << endl;       // 'A'
```


```
int x = 516;
int * xptr = &x;
char * chptr = (char *) &x;
double* dptr = (double*) &x;
```






```
cout << *xptr << endl;   => 516
cout << *chptr << endl;  => 4
```

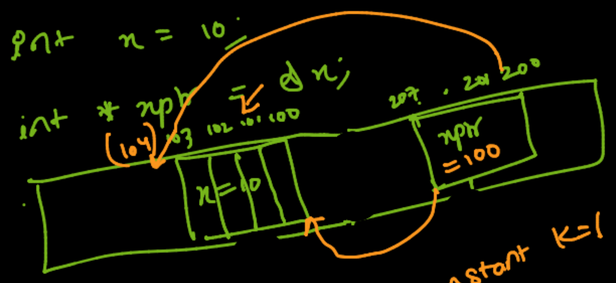


(int*) chptr => 100

void*

int x = 10;

int * xptr = &x;



xptr + (K) → constant K=1

⇒ xptr + 1 ?

xptr + K ⇒ K * size(datatype) (pointer)

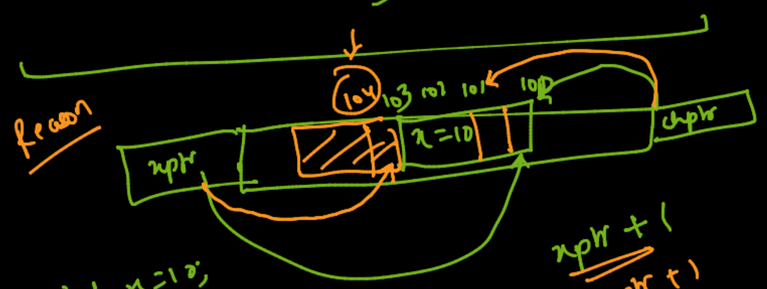
xptr = xptr + 1

xptr = 100 + 1(4)

= 104

xptr = 100 + 4(4)

= 116



int x=10;

int * xptr = &x;

char * chptr = (char *) &x;

xptr + (chptr + 1)

*(xptr + 1)

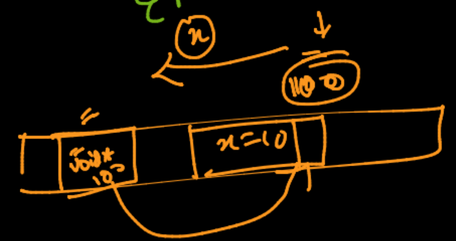
Void * generic pointer

1. int x= 10;

Void * gptr = &x;

2. When you try to deref. a gen. pointer?

Error.



3.) pointer arithmetic won't work with generic pointer.

gptr + 1 ⇒ gptr + 1 * (size(void*)) error

Null Pointer

⇒ int n = 10;

int *xptr ; // declaration

cout << xptr << endl; // garbage

cout << *xptr << endl; // garbage or seg fault

Appl^n memory → or Outside → Segmentation

[int * xptr = NULL ; ✓ or 0 ✓ or nullptr ✓]

⇒ *xptr ⇓ error

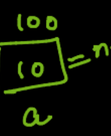
[NULL ✓ or nullptr ✓ or 0 ✓]

Pointers and Function

1. Call by value
2. Call by ref. (used ref. variables)

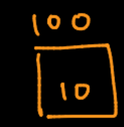
⇒

```
f( int n ) {
  n++;
}

main( ) {
  int a = 10;
  f(a);
}
```



= not changed

```
f( int &n ) {
  n++;
}

main( ) {
  a = 10;
  f(a);
}
```



```
f( int *aptr ) {
   *aptr ← dereferencing
}

main ( ) {
   int a = 10;
   f( &a );
}
```


```
void f ( int * aptr ) {
   cout << *aptr << endl; //10
   *aptr++;
}

main( ) {
   → int a=10;
   f( &a );
}
```

↳ Reference Variable → stripped down version of pointer

↳ Pointer

```
f( int &x ) {
   x++;
}

main ( ) {
   a=10; f(a)
}
```

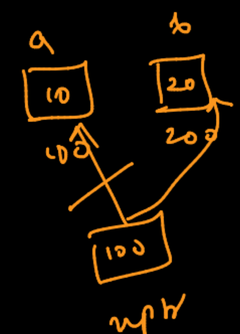
```
int &x = a;
x++;
```

```
f( int *x ) {
   *x++;
}

main() {
   a=10 ; f(&a)
}
```

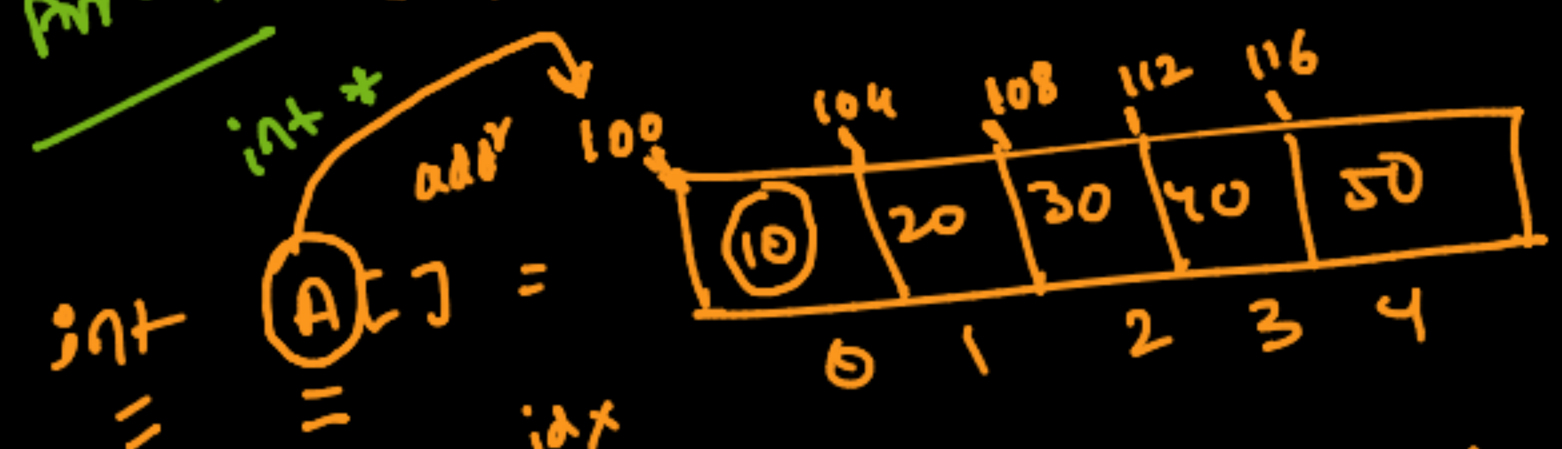
```
int *x = &a;
*xptr++
```

1. Reference variables initial with valid addr
2. it cannot be initialized with NULL
3. it cannot be reinitialized

```
int a=10;
int *aptr = &a;
   aptr = &b;
int b=20;
```



Pointers and Arrays (thought of) *



100 &A[0] or A
104 &A[1] or A+1
(100 + i(int)) &A[i] or A+i

A[0] | *A
A[1] | *(A+1)
A[i] | *(A+i)

In C++, name of the array can be thought of as a pointer to the first element of the array.

first element → 0th index

```
f( int A[ ],int n) {
  sum=0;
  for(i=0; i<n; i++)
    sum += A[i];
  cout << sum;
}
main( ) {
  int A[ ] = {10,20,30,40,50};
  f(A);
}
```

↳ passed by ref. by default

```
f(int *A) {   ↳ &A[0]
}
main( ) {
  int A[ ] = {1, 2, 3, 4, 5};
  f(A);   ↳ &A[0]
}
```

*A, *(A+1), *(A+2)....