Pointers
When a variable is defined the compiler (linker/loader actually) allocates a real memory address for the variable.

- `int x;` will allocate 4 bytes in the main memory, which will be used to store an integer value.

When a value is assigned to a variable, the value is actually placed to the memory that was allocated.

- `x = 3;` will store integer 3 in the 4 bytes of memory.
Pointers

- When the value of a variable is used, the contents in the memory are used.
  - \( y = x \); will read the contents in the 4 bytes of memory, and then assign it to variable \( y \).
- \&x can get the address of \( x \). (referencing operator &)
- The address can be passed to a function:
  - `scanf("%d", &x);`
- The address can also be stored in a variable ......
Pointers

- To declare a pointer variable
  
  `type * pointername;`

- For example:
  
  - `int * p1;`  
    
    `p1` is a variable that tends to point to an integer, (or `p1` is a int pointer)
  
  - `char * p2;`
  
  - `unsigned int * p3;`

- `p1 = &x;     /* Store the address in p1 */`

- `scanf("%d", p1); /* i.e. scanf("%d", &x); */`

- `p2 = &x;     /* Will get warning message */`
Initializing Pointers

- Like other variables, always initialize pointers before using them!!
- For example:

```c
int main()
{
    int x;
    int *p;
    scanf("%d",p);    /*
     */
    p = &x;
    scanf("%d",p);   /* Correct */
}
```
Using Pointers

- You can use pointers to access the values of other variables, i.e. the contents of the memory for other variables.

- To do this, use the * operator (dereferencing operator).
  - Depending on different context, * has different meanings.

- For example:

```c
int n, m=3, *p;
p= &m;
n= *p;
printf("%d\n", n);
printf("%d\n", *p);
```
int m = 3, n = 100, *p;
p = &m;
printf("m is %d\n", *p);
m++;
printf("now m is %d\n", *p);
p = &n;
printf("n is %d\n", *p);
*p = 500; /* *p is at the left of "=" */
printf("now n is %d\n", n);
Pointers as Function Parameters

- Sometimes, you want a function to assign a value to a variable.
  - e.g. `scanf()`

- Eg. you want a function that computes the minimum AND maximum numbers in 2 integers.

- Method 1, use two global variables.
  - In the function, assign the minimum and maximum numbers to the two global variables.
  - When the function returns, the calling function can read the minimum and maximum numbers from the two global variables.

- This is bad because the function is not reusable.
Pointers as Function Parameters

Instead, we use the following function:

```c
void min_max(int a, int b, int *min, int *max){
    if(a > b){
        *max = a;
        *min = b;
    } else{
        *max = b;
        *min = a;
    }
}
```

```c
int main()
{
    int x,y;
    int small, big;
    printf("Two integers: ");
    scanf("%d %d", &x, &y);

    min_max(x,y,&small,&big);
    printf("%d <= %d", small, big);
    return 0;
}
```
When a pointer variable points to an array element, there is a notion of adding or subtracting an integer to/from the pointer.

```
int a[10], *p;
p = &a[2];
*p = 10;
*(p+1) = 10;
printf("%d", *(p+3));
```

```
int a[10], *p;
a[2] = 10;
a[3] = 10;
printf("%d", a[5]);
```
More examples:

```c
int a[10], *p, *q;
p = &a[2];
q = p + 3;  /* q points to a[5] now */
p = q - 1;  /* p points to a[4] now */
p++;        /* p points to a[5] now */
p--;        /* p points to a[4] now */
*p = 123;    /* a[4] = 123 */
q = p;       /* q points to a[4] now */
scanf("%d", q) /* scanf("%d", &a[4]) */
```
If two pointers point to elements of a same array, then there are notions of subtraction and comparisons between the two pointers.

```c
int a[10], *p, *q, i;
p = &a[2];
q = &a[5];
i = q - p;     /* i is 3*/
i = p - q;     /* i is -3 */
p < q;         /* true */
p == q;        /* false */
p != q;        /* true */
```
Pointers and Arrays

◆ Recall that the value of an array name is also an address.
◆ In fact, pointers and array names can be used interchangeably in many (but not all) cases.
  – E.g. int n, *p;  p = &n;
  – n = 1;  *p = 1;  p[0] = 1;
◆ The major differences are:
  – Array names come with valid spaces where they “point” to. And you cannot “point” the names to other places.
  – Pointers do not point to valid space when they are created. You have to point them to some valid space (initialization).
Using Pointers to Access Array Elements

```c
int a[10], *p;
p = &a[2];
p[0] = 10;
p[1] = 10;
printf("%d", p[3]);
```

A pointer can be used just like an array.
An Array Name is Like a Constant Pointer

- Array name is like a constant pointer which points to the first element of the array.
  ```c
  int a[10], *p, *q;
p = a;      /* p = &a[0] */
q = a + 3;   /* q = &a[0] + 3 */
a ++;        /* illegal !!! */
  ```

- Therefore, you can “pass an array” to a function. Actually, the address of the first element is passed.
  ```c
  int a[ ] = { 5, 7, 8, 2, 3 };
  sum( a, 5 );  /* Equal to sum(&a[0],5) */
  ...........
  ```
/* Sum – sum up the ints in the given array */
int sum(int *ary, int size)
{
    int i, s;
    for(i = 0, s=0; i<size;i++){
        s+= ary[i];
    }
    return s;
}

/* In another function */
int a[1000],x;

......
x=
sum(&a[100],50);
/* This sums up a[100], a[101], ..., a[149] */
Allocating Memory for a Pointer (1)

- The following program is wrong!

```c
#include <stdio.h>
int main()
{
    int *p;
    scanf("%d",p);
    return 0;
}
```

- This one is correct:

```c
#include <stdio.h>
int main()
{
    int *p;
    int a;
    p = &a;
    scanf("%d",p);
    return 0;
}
```
Allocating Memory for a Pointer (2)

- There is another way to allocate memory so the pointer can point to something:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p;
    p = (int *) malloc( sizeof(int) );  /* Allocate 4 bytes */
    scanf("%d", p);
    printf("%d", *p);
    free(p);  /* This returns the memory to the system. */
                /* Important !!! */
}
```
Allocating Memory for a Pointer (3)

- Prototypes of `malloc()` and `free()` are defined in `stdlib.h`

```c
void * malloc(size_t number_of_bytes);
void free(void * p);
```

- You can use `malloc` and `free` to dynamically allocate and release the memory;

```c
int * p;
p = (int *) malloc(1000 * sizeof(int));
for(i=0; i<1000; i++)
    p[i] = i;
p[1000] = 3;    /* Wrong! */
free(p);
p[0] = 5;    /* Wrong! */
```
#include <stdio.h>
#include <stdlib.h>
/* Print out all prime numbers which are less than m */
void print_prime(int m)
{
    int i, j;
    int * ary = (int *) malloc(m * sizeof(int));
    if (ary == NULL) exit(-1);
    for (i = 0; i < m; i++)
        ary[i] = 1;
    /* Assume all integers between 0 and m - 1 are prime */
    ary[0] = ary[1] = 0;
    /* Note that in fact 0 and 1 are not prime */
    for (i = 3; i < m; i++)
    {
        for (j = 2; j < i; j++)
        {
            if (ary[i] && i % j == 0)
            {
                ary[i] = 0;
                break;
            }
        }
    }
    for (i = 0; i < m; i++)
    {
        if (ary[i]) printf("%d ", i);
        free(ary);
        printf("\n");
    }
}
int main()
{
    int m;
    printf("m = ");
    scanf("%d", &m);
    printf("\n");
    print_prime(m);
    return 0;
}