Pointer in C language

Pointers

A *pointer* is a reference to another variable (memory location) in a program

- Used to change variables inside a function (reference parameters)
- Used to remember a particular member of a group (such as an array)
- Used in dynamic (on-the-fly) memory allocation (especially of arrays)
- Used in building complex data structures (linked lists, stacks, queues, trees, etc.)

Outline

- Pointers
  - Basics
    - Variable declaration, initialization, NULL pointer
    - & (address) operator, * (indirection) operator
    - Pointer parameters, return values
    - Casting points, void *
  - Arrays and pointers
    - 1D array and simple pointer
    - Passing as parameter
  - Dynamic memory allocation
    - calloc, free, malloc, realloc
    - Dynamic 2D array allocation (and non-square arrays)

Pointer Basics

- Variables are allocated at *addresses* in computer memory (address depends on computer/operating system)
- Name of the variable is a reference to that memory address
- A pointer variable contains a representation of an address of another variable (P is a pointer variable in the following):

```
int V = 101;

int *P = &V;
```

Name	**V**	**P**
Address	*v* (some value)	*p* (some value)
Abstract Representation	101 ←	→
Concrete Representation	4 bytes for int value 101	4 bytes for mem address *v*

Pointer Variable Definition

Basic syntax: *Type* **Name*

Examples:

int *P;   /* P is var that can point to an int var */

float *Q;   /* Q is a float pointer */

char *R;   /* R is a char pointer */

Complex example:

int *AP[5];   /* AP is an array of 5 pointers to ints */

– more on how to read complex declarations later

Address (&) Operator

The address (&) operator can be used in front of any variable object in C -- the result of the operation is the location in memory of the variable

Syntax: &*VariableReference*

Examples:

int V;

int *P;

int A[5];

&V - memory location of integer variable V

&(A[2]) - memory location of array element 2 in array A

&P - memory location of pointer variable P

Pointer Variable Initialization/Assignment

NULL - pointer lit constant to non-existent address
- used to indicate pointer points to nothing

Can initialize/assign pointer vars to NULL or use the address (&) op to get address of a variable
- variable in the address operator must be of the right type for the pointer (an integer pointer points only at integer variables)

Examples:

```
int V;
int *P = &V;
int A[5];
P = &(A[2]);
```

Indirection (*) Operator

A pointer variable contains a memory address

To refer to the *contents* of the variable that the pointer points to, we use indirection operator

Syntax: **PointerVariable*

Example:

```
int V = 101;
int *P = &V;
/* Then *P would refer to the contents of the variable V
   (in this case, the integer 101) */
printf("%d",*P);  /* Prints 101 */
```

Pointer Sample

```
int A = 3;
int B;
int *P = &A;
int *Q = P;
int *R = &B;

printf("Enter value:");
scanf("%d",R);
printf("%d %d\n",A,B);
printf("%d %d %d\n",
  *P,*Q,*R);
```

```
Q = &B;
if (P == Q)
  printf("1\n");
if (Q == R)
  printf("2\n");
if (*P == *Q)
  printf("3\n");
if (*Q == *R)
  printf("4\n");
if (*P == *R)
  printf("5\n");
```

Reference Parameters

To make changes to a variable that exist after a function ends, we pass the address of (a pointer to) the variable to the function (a reference parameter)

Then we use indirection operator inside the function to change the value the parameter points to:

```
void changeVar(float *cvar) {
  *cvar = *cvar + 10.0;
}

float X = 5.0;

changeVar(&X);
printf("%.1f\n",X);
```

Pointer Return Values

A function can also return a pointer value:

```
float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);

  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);

  return theMax;
}

void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA;

  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
  printf("%.1f %.1f\n",*maxA,A[4]);
}
```

Pointers to Pointers

A pointer can also be made to point to a pointer variable (but the pointer must be of a type that allows it to point to a pointer)

Example:

```
int V = 101;
int *P = &V;       /* P points to int V */
int **Q = &P;      /* Q points to int pointer P */

printf("%d %d %d\n",V,*P,**Q); /* prints 101 3 times */
```

Pointer Types

Pointers are generally of the same size (enough bytes to represent all possible memory addresses), but it is inappropriate to assign an address of one type of variable to a different type of pointer

Example:

```
int V = 101;
float *P = &V; /* Generally results in a Warning */
```

Warning rather than error because C will allow you to do this (it is appropriate in certain situations)

Casting Pointers

When assigning a memory address of a variable of one type to a pointer that points to another type it is best to use the cast operator to indicate the cast is intentional (this will remove the warning)

Example:

```
int V = 101;
float *P = (float *) &V; /* Casts int address to float * */
```

Removes warning, but is still a somewhat unsafe thing to do

1D Arrays and Pointers

int A[5] - A is the address where the array starts (first element), it is equivalent to &(A[0])

A is in some sense a pointer to an integer variable

To determine the address of A[x] use formula:

- (address of A + x * bytes to represent int)
- (address of array + element num * bytes for element size)

The + operator when applied to a pointer value uses the formula above:

- A + x is equivalent to &(A[x])
- *(A + x) is equivalent to A[x]

1D Array and Pointers Example

```
float A[6] = {1.0,2.0,1.0,0.5,3.0,2.0};
float *theMin = &(A[0]);
float *walker = &(A[1]);

while (walker < &(A[6])) {
  if (*walker < *theMin)
    theMin = walker;
  walker = walker + 1;
}

printf("%.1f\n",*theMin);
```

1D Array as Parameter

When passing whole array as parameter use syntax *ParamName*[], but can also use **ParamName*

Still treat the parameter as representing array:

```
int totalArray(int *A, int N) {
  int total = 0;
  for (I = 0; I < N; I++)
    total += A[I];
  return total;
}
```

For multi-dimensional arrays we still have to use the *ArrayName*[][*Dim2*][*Dim3*]etc. form

Declarations Examples

`int A`	`A is a int`
`float B [5]`	`B is a 1D array of size 5 of floats`
`int * C`	`C is a pointer to an int`
`char D [6][3]`	`D is a 2D array of size 6,3 of chars`
`int * E [5]`	`E is a 1D array of size 5 of pointers to ints`
`int (* F) [5]`	`F is a pointer to a 1D array of size 5 of ints`
`int G (…)`	`G is a function returning an int`
`char * H (…)`	`H is a function returning a pointer to a char`

Limits of Static Allocation

What if we don't know how much space we will need ahead of time?

Example:

- ask user how many numbers to read in
- read set of numbers in to array (of appropriate size)
- calculate the average (look at all numbers)
- calculate the variance (based on the average)

Problem: how big do we make the array??

- using static allocation, have to make the array as big as the user might specify (might not be big enough)

Dynamic Memory Allocation

Allow the program to allocate some variables (notably arrays), during the program, based on variables in program (dynamically)

Previous example: ask the user how many numbers to read, then allocate array of appropriate size

Idea: user has routines to request some amount of memory, the user then uses this memory, and returns it when they are done. Memory allocated in the *Data Heap*

Memory Management Functions

calloc - routine used to allocate arrays of memory

malloc - routine used to allocate a single block of memory

realloc - routine used to extend the amount of space allocated previously

free - routine used to tell program a piece of memory no longer needed

- note: memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up

Array Allocation with calloc

prototype: void * calloc(size_t num, size_t esize)

- size_t is a special type used to indicate sizes, generally an unsigned int
- num is the number of elements to be allocated in the array
- esize is the size of the elements to be allocated
  - generally use sizeof and type to get correct value
- an amount of memory of size num*esize allocated on heap
- calloc returns the address of the first byte of this memory
- generally we cast the result to the appropriate type
- if not enough memory is available, calloc returns NULL

calloc Example

```
float *nums;
int N;
int I;

printf("Read how many numbers:");
scanf("%d",&N);
nums = (float *) calloc(N, sizeof(float));
/* nums is now an array of floats of size N */
for (I = 0; I < N; I++) {
  printf("Please enter number %d: ",I+1);
  scanf("%f",&(nums[I]));
}
/* Calculate average, etc. */
```

Releasing Memory (free)

- prototype: void free(void *ptr)
  - memory at location pointed to by ptr is released (so we could use it again in the future)
  - program keeps track of each piece of memory allocated by where that memory starts
  - if we free a piece of memory allocated with calloc, the entire array is freed (released)
  - results are problematic if we pass as address to free an address of something that was not allocated dynamically (or has already been freed)

free Example

```
float *nums;
int N;

printf("Read how many numbers:");
scanf("%d",&N);
nums = (float *) calloc(N, sizeof(float));

/* use array nums */

/* when done with nums: */

free(nums);

/* would be an error to say it again - free(nums) */
```

The Importance of free

```
void problem() {
  float *nums;
  int N = 5;

  nums = (float *) calloc(N, sizeof(float));

  /* But no call to free with nums */
} /* problem ends */
```

When function problem called, space for array of size N allocated, when function ends, variable nums goes away, but the space nums points at (the array of size N) does not (allocated on the heap) - furthermore, we have no way to figure out where it is)

Problem called *memory leakage*

Array Allocation with malloc

prototype: void * malloc(size_t esize)

- similar to calloc, except we use it to allocate a single block of the given size esize
- as with calloc, memory is allocated from heap
- NULL returned if not enough memory available
- memory must be released using free once the user is done
- can perform the same function as calloc if we simply multiply the two arguments of calloc together
  - malloc(N * sizeof(float)) is equivalent to
  - calloc(N,sizeof(float))

Increasing Memory Size with realloc

prototype: void * realloc(void * ptr, size_t esize)

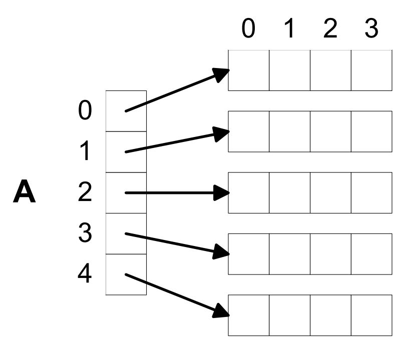
- ptr is a pointer to a piece of memory previously dynamically allocated
- esize is new size to allocate (no effect if esize is smaller than the size of the memory block ptr points to already)
- program allocates memory of size esize,
- then it copies the contents of the memory at ptr to the first part of the new piece of memory,
- finally, the old piece of memory is freed up

realloc Example

```
float *nums;
int I;

nums = (float *) calloc(5, sizeof(float));
/* nums is an array of 5 floating point values */

for (I = 0; I < 5; I++)
  nums[I] = 2.0 * I;
/* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */

nums = (float *) realloc(nums,10 * sizeof(float));
/* An array of 10 floating point values is allocated,
   the first 5 floats from the old nums are copied as
   the first 5 floats of the new nums, then the old
   nums is released */
```

Dynamically Allocating 2D Arrays

- Can not simply dynamically allocate 2D (or higher) array
- Idea - allocate an array of pointers (first dimension), make each pointer point to a 1D array of the appropriate size
- Can treat result as 2D array



Dynamically Allocating 2D Array

```
float **A;  /* A is an array (pointer) of float
               pointers */
int I;

A = (float **) calloc(5,sizeof(float *));
/* A is a 1D array (size 5) of float pointers */

for (I = 0; I < 5; I++)
  A[I] = (float *) calloc(4,sizeof(float));
/* Each element of array points to an array of 4
   float variables */

/* A[I][J] is the Jth entry in the array that the
   Ith member of A points to */
```

Non-Square 2D Arrays

No need to allocate square 2D arrays:

```
float **A;
int I;

A = (float **) calloc(5,
         sizeof(float *));

for (I = 0; I < 5; I++)
  A[I] = (float **)
           calloc(I+1,
            sizeof(float));
```