CS1101: Foundations of Programming

Pointers

Dept. of Computer Science & Engineering
Indian Institute of Technology Patna

Basic concept

- In memory, every data item occupies one or more contiguous bytes (eg. `char` - 1 byte, `int` - 4 bytes, etc.)
- Whenever a variable is declared, the system allocates required amount of memory to hold the value of the variable
  - Every byte of the memory has unique address. For multi-byte data, this is usually specified by the address of the first byte
- In C, many manipulations can be done with addresses

Accessing address of a variable

- Address of a variable can be accessed using the & (address-of) operator
- The operator & immediately preceding a variable returns the address of the variable
- & operator can be used only with a simple variable or an array element, example
  `&x`
  `&a[5]`
- Following are illegal usage
  `&123` — address of a constant is not defined
  `&(x+y)` — address of an expression is not defined

Example

- **Consider** `int var=100;`
  - **This statement reserves a memory location for integer variable** `var` **and put the value 100 in that location**
  - **Suppose that the address of that location chosen is 792**
  - **During execution of the program the system always associates the name with the address 792**
  - **The value 100 can be accessed by using either the name** `var` **or by looking at whatever is written in the address (**`&var`**)**
  - `var` **refers to 100**
  - `&var` **refers to the memory address 792**

780		
784		
788		
792	100	**var**
796		
800		

Pointer declaration

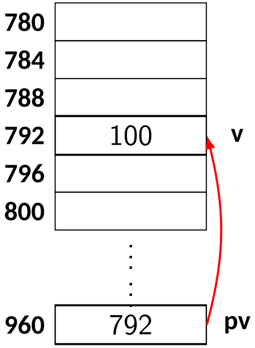
- A pointer is just a C variable whose value is the address of another variable
- A pointer variable must be declared first before using it
- **Syntax:** <data_type> *<pointer_name>;
- **Example** int *pvar;
  - The * tells that the variable pvar is a pointer
  - pvar will be used to point a variable of type int
- Just after declaration, pvar may not point to any valid location, usually it will have some garbage values.
- One can initialized to NULL value int *pvar=NULL;
- Pointers are variables and are stored in the memory. They too have their own addresses like &pvar

Example

- Consider

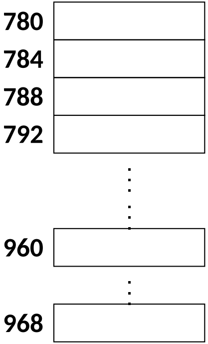
```
int v=100;
int *pv;
pv = &v;
```

- As the memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory
- A variable that hold memory address are usually called pointers
- As pointers are also variables, their values are also stored in some memory locations
- Once the `pv` is assigned a valid memory location, the * operator can be used to access the value at that address



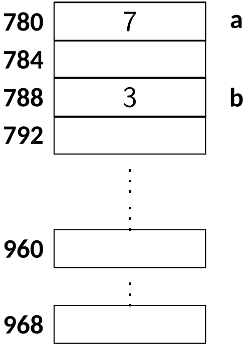
Example

```
int a=7, b=3;
int *x, *y;
x = &a;
y = &b;
*x = 20;
*y = *x + 3;
y=x;
```



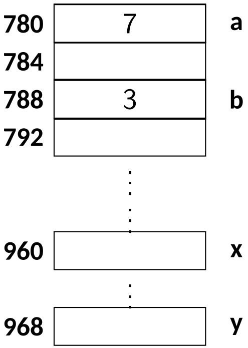
Example

```
int a=7, b=3;
int *x, *y;
x = &a;
y = &b;
*x = 20;
*y = *x + 3;
y=x;
```



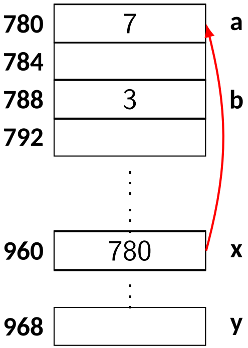
Example

```
int a=7, b=3;
int *x, *y;
x = &a;
y = &b;
*x = 20;
*y = *x + 3;
y=x;
```



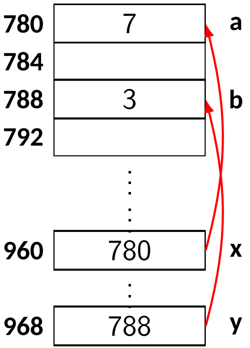
Example

```
int a=7, b=3;
int *x, *y;
x = &a;
y = &b;
*x = 20;
*y = *x + 3;
y=x;
```



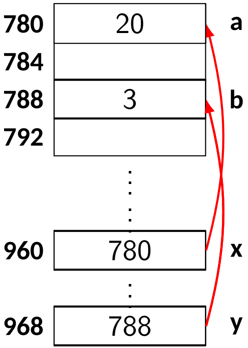
Example

```
int a=7, b=3;
int *x, *y;
x = &a;
y = &b;
*x = 20;
*y = *x + 3;
y=x;
```



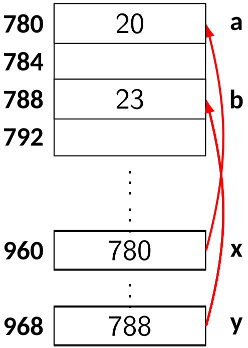
Example

```
int a=7, b=3;
int *x, *y;
x = &a;
y = &b;
*x = 20;
*y = *x + 3;
y=x;
```



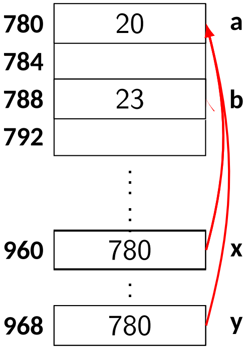
Example

```
int a=7, b=3;
int *x, *y;
x = &a;
y = &b;
*x = 20;
*y = *x + 3;
y=x;
```



Example

```
int a=7, b=3;
int *x, *y;
x = &a;
y = &b;
*x = 20;
*y = *x + 3;
y=x;
```



Note

- **Pointers have types, e.g.** `int *pi; float *pf;`
- **Pointer variables should always point to a data item of the same type**

```
float f; int *pi;
pi = &f; // Avoid such things, compiler can alert
```

- **However, type casting can be used under certain scenario but with care**

```
pi = (int *)&f;
```

Pointers and arrays

- **When an array is declared:**
  - **The array has a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations**
  - **The base address is the location of the first element of the array**
  - **The compiler defined the array name as a constant pointer to the first element**

Example

Consider the following

```
int x[5]={1, 2, 3, 4, 5};
int *p;
```

Assuming base address of x is 1000 and an integer requires 4 bytes

Element	Value	Address
x[0]	1	1000
x[1]	2	1004
x[2]	3	1008
x[3]	4	1012
x[4]	5	1016

Both `x` **and** `&x[0]` **will have the value 1000**

`p=x;` **and** `p=&x[0];` **are equivalent.**

Example

`int x[5]={1, 2, 3, 4, 5};`

`int *p;`

Suppose we assign `p = &x[0];`

Now we access successive values of x by using `p++` **or** `p--` **to move from one element to another**

Relationship between `p` **and** `x`

p	= &x[0]	= 1000
p+1	= &x[1]	= 1004
p+2	= &x[2]	= 1008
p+3	= &x[3]	= 1012
p+4	= &x[4]	= 1016

`(p+i)` **gives the address of** `x[i]`

`(p+i)` **is the same as** `&x[i]`

`*(p+i)` **gives the value of** `x[i]`

For any array `A`**, we have** `A+i = &A[i]` **and** `*(A+i) = A[i]`**.**

Printing pointers

```
int main(){
   int a[5]={1,2,3,4,5}, i, *p;
   for(i=0; i<4; i++)
     printf("&a[%d] = %p, a[%d] = %d\n",i, a+i, i, *(a+i));
   p = a;
   printf("p = %p, &p = %p\n", p, &p);
   return 0;
}
```

```
&a[0] = 0x7fff0ea756f0, a[0] = 1
&a[1] = 0x7fff0ea756f4, a[1] = 2
&a[2] = 0x7fff0ea756f8, a[2] = 3
&a[3] = 0x7fff0ea756fc, a[3] = 4
p = 0x7fff0ea756f0, &p = 0x7fff0ea756e8
```

Printing pointers

```
int main(){
   int a[5]={1,2,3,4,5}, i, *p;
   for(i=0; i<4; i++)
     printf("&a[%d] = %p, a[%d] = %d\n",i, a+i, i, *(a+i));
   p = a;
   printf("p = %p, &p = %p\n", p, &p);
   return 0;
}
```

```
&a[0] = 0x7fff0ea756f0, a[0] = 1
&a[1] = 0x7fff0ea756f4, a[1] = 2
&a[2] = 0x7fff0ea756f8, a[2] = 3
&a[3] = 0x7fff0ea756fc, a[3] = 4
p = 0x7fff0ea756f0, &p = 0x7fff0ea756e8
```

What is `&a`?
It is NOT an `int` pointer

Pointer to an array vs pointer to a pointer

```
int main(){
   int a[5]={1,2,3,4,5}, *p;
   printf("a    = %p\n", a);
   printf("a+1  = %p\n", a+1);
   printf("&a   = %p\n", &a);
   printf("&a+1 = %p\n", &a+1);
   p = a;
   printf("p    = %p\n", p);
   printf("p+1  = %p\n", p+1);
   printf("&p   = %p\n", &p);
   printf("&p+1 = %p\n", &p+1);
   return 0;
}
```

```
a    = 0x7ffea1ad6a50
a+1  = 0x7ffea1ad6a54
&a   = 0x7ffea1ad6a50
&a+1 = 0x7ffea1ad6a64
p    = 0x7ffea1ad6a50
p+1  = 0x7ffea1ad6a54
&p   = 0x7ffea1ad6a48
&p+1 = 0x7ffea1ad6a50
```

Pointers in expressions

- **Pointers variable can be used in expressions**
- **If `p` is a `double` pointer, then `*p` points to a `double` number (similarly for other pointers)**
- **Let `p1` and `p2` are two pointer variables, then we can write following expressions**

```
val = (*p1) + (*p2);
x = (*p1) * (*p2);
*p1 = *p1 + 10;
y = *p1 / *p2 - 10;
```

Arithmetic on pointers

- **Certain arithmetic operations are legal in C**
  - **Add an integer to pointer**
  - **Subtract an integer to pointer**
  - **Subtract one pointer from another**
    - **If `p1` and `p2` are two pointers to the same array then `p2-p1` gives the number of elements between `p1` and `p2`**

Arithmetic on pointers

- Certain arithmetic operations are legal in C
  - Add an integer to pointer
  - Subtract an integer to pointer
  - Subtract one pointer from another
    - If `p1` and `p2` are two pointers to the same array then `p2-p1` gives the number of elements between `p1` and `p2`
- Some operations are NOT allowed
  - Add two pointers `p1 = p1 + p2;`
  - Multiply or divide a pointer in an expression

```
p1 = p2 * 5;
p1 = 10 * p2 - 10;
```

Scale factor

- **An integer value can be added to or subtracted from a pointer variable**

```
int a[5]={1,2,3,4,5}, *p;
p = &x[1];
printf("%d",*p); // will print 2
p++;             // increase p by the number of bytes for int
printf("%d",*p); // will print 3
p=p+2;           // increase p by 2*sizeof(int)
printf("%d",*p); // will print 5
```

Scale factor

```
int a[5]={1,2,3,4,5}, *pa;
char c[5]={'m','n','p','q','r'}, *pc;
pa = a;       // pa points to 1st element of a
pc = c;       // pc points to 1st element of c
pa = pa + 1; // pa points to 2nd element of a
pc = pc + 1; // pc points to 2nd element of c
```

When a pointer variable is incremented by 1, it does not necessarily increment by 1 byte. It increases by the size of the data type to which it points to.

Thus, pointers have type. They are not just a single address data type.

Pointer type & Scale factor

- Scale factor depends on the data type
  - `char` – 1
  - `int` – 4
  - `float` – 4
  - `double` – 8
- If `p` is `int` pointer, then `p--` will decrease the value of `p` by 4
- If `p` is `double` pointer, then `p++` will increase the value of `p` by 8
- The exact scale factor can vary from machine to another
- Exact size on a given machine can be found using `sizeof` operator

```
printf("Size of int: %d\n",sizeof(int));
printf("Size of double: %d\n",sizeof(double));
printf("Size of char: %d\n",sizeof(char));
```

Example: Scale factor

```
⋮
char c[10], *pc;
int i[10], *pi;
float f[20], *pf;
double d[20], *pd;
pc = c; printf("pc=%p pc+1=%p\n",pc, pc+1);
pi = i; printf("pi=%p pi+1=%p\n",pi, pi+1);
pf = f; printf("pf=%p pf+1=%p\n",pf, pf+1);
pd = d; printf("pd=%p pd+1=%p\n",pd, pd+1);
⋮
```

Example: Scale factor

```
⋮
char c[10], *pc;
int i[10], *pi;
float f[20], *pf;
double d[20], *pd;
pc = c; printf("pc=%p pc+1=%p\n",pc, pc+1);
pi = i; printf("pi=%p pi+1=%p\n",pi, pi+1);
pf = f; printf("pf=%p pf+1=%p\n",pf, pf+1);
pd = d; printf("pd=%p pd+1=%p\n",pd, pd+1);
⋮
```

```
pc=0x7ffe0960637e pc+1=0x7ffe0960637f
pi=0x7ffe09606250 pi+1=0x7ffe09606254
pf=0x7ffe09606280 pf+1=0x7ffe09606284
pd=0x7ffe096062d0 pd+1=0x7ffe096062d8
```

Pointers and functions

- In C, arguments to function are passed by value
  - The data are copied to function. Any modifications made in the called function are not visible in the calling function
- Pointers can be passed to function
  - This allows the data item within the calling function to be accessed using the address and modified

Example

```
void swap1(int a, int b){
   int t;
   t = b; b = a; a = t;
}
void swap2(int *a, int *b){
   int t;
   t = *b; *b = *a; *a = t;
}
int main(){
   int a=10, b=20;
   swap1(a,b); printf("a=%d b=%d\n",a,b);
   swap2(&a,&b); printf("a=%d b=%d\n",a,b);
   return 0;
}
```

Example

```
void swap1(int a, int b){
   int t;
   t = b; b = a; a = t;
}
void swap2(int *a, int *b){
   int t;
   t = *b; *b = *a; *a = t;
}
int main(){
   int a=10, b=20;
   swap1(a,b); printf("a=%d b=%d\n",a,b);
   swap2(&a,&b); printf("a=%d b=%d\n",a,b);
   return 0;
}
```

Output:

a=10 b=20

a=20 b=10

Useful application of pointers

- In C, a function can return only one value
- Consider a situation when a function computes two values and need to be returned to the calling function
- Possible option using pointers
  - Declare variables in the calling function and pass these addresses as arguments to the function
  - Called function can store necessary values to those memory addresses which will be reflected in calling function

Example: returning multiple values

```c
void func(int a, int b, int *px, int *py){
   *px = a * b;
   *py = a + b;
}
int main(){
   int a=10, b=20, p, s;
   func(a,b,&p,&s);
   printf("p=%d s=%d\n",p,s);
   return 0;
}
```

Example: returning multiple values

```
void func(int a, int b, int *px, int *py){
   *px = a * b;
   *py = a + b;
}
int main(){
   int a=10, b=20, p, s;
   func(a,b,&p,&s);
   printf("p=%d s=%d\n",p,s);
   return 0;
}
```

Output:

p=200 s=30

Pointers / arrays in function prototypes

- The following statements have the same meaning

```
func(int a[],…);
func(int a[100],…);
func(int *a,…);
```

- In all three cases `a` is an `int` pointer. It does not matter whether the actual parameter is the name of an `int` array or an `int` pointer. Inside the function `a` is copy of the address passed
- If the parameter passed is a pointer to an individual item use pointer notation in the function prototype
- If the parameter passed is an array, you can use any one of the two conventions in the function prototype. The array notation may be preferred

Function can return a pointer

- **Find the first upper-case letter in a string**

```
char *fupper(char s[]){
   while (*s)
     { if((*s >='A') && (*s <= 'Z')) return s; else s++; }
   return NULL;
}
int main(){
   char s[100], *p;  scanf("%s",s);  p = fupper(s);
   if(p) printf("Found %c\n",*p);
   else printf("Not found\n");
   return 0;
}
```

Function can return a pointer

- **Find the first upper-case letter in a string**

```
char *fupper(char s[]){
   while (*s)
     { if((*s >='A') && (*s <= 'Z')) return s; else s++; }
   return NULL;
}
int main(){
   char s[100], *p;  scanf("%s",s);  p = fupper(s);
   if(p) printf("Found %c\n",*p);
   else printf("Not found\n");
   return 0;
}
```

A function should not return a pointer to a local variable. After the function returns, the local variable no longer exists.

Issues with fixed size array

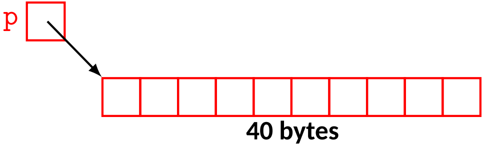
- Amount of data cannot be predicted beforehand so we used to have large size array and utilize a small portion of it
- Number of elements can keep on changing during program execution
- Dynamic memory allocation:
  - Know how much memory is needed after the program is run then dynamically allocate the required amount of memory
  - User can provide the desired size
- C provides functions to allocate memory dynamically — `malloc`, `calloc`, `realloc`

Memory allocation functions

- `malloc` — **Allocates requested number of bytes and returns a pointer to the first byte of the allocated space**
- `calloc` — **Allocates space for an array of elements, initializes them to zero and then returns a pointer to the first byte of the memory**
- `free` — **Frees previously allocated memory**
- `realloc` — **Modifies the size of previously allocated space**

Allocating a block of memory

- **A block of memory can be allocated using function** `malloc`
  - **Input / argument — number of bytes to be reserved**
  - **Returns a pointer of type** `void*` **which can be casted to any pointer**
- **Prototype:** `void *malloc(size_t size);` **(**`size_t` **is a special unsigned integer type)**
- **Function call:** `int *p = (int *) malloc(10 * sizeof(int));`
- **It reserves $10 \times 4 = 40$ bytes of memory,** `p` **will be casted to** `int *` **and point to newly allocated space**
- `char *pc = (char *) malloc(10);`



Note

- `malloc` **always allocated contiguous chunk of memory**
  - **The allocation can fail if sufficient space is not available. In that case `malloc` returns NULL**
- **Example:**

```
if((p=(int*)malloc(100*sizeof(int)))==NULL){
  printf("Cannot allocate memory\n");
  exit(1);
}
```

- **You can use `exit` (status) instead of `return` status. To use `exit`, you need to include `#include <stdlib.h>`**

Example

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int *x, n, i;
    printf("Enter the size\n"); scanf("%d",&n);
    if(n <= 0) { printf("invalid\n"); exit(1); }
    x = (int *) malloc(n * sizeof(int));
    if(x == NULL) { printf("Insufficient memory\n"); exit(1); }
    for(i=0; i < n; i++){ scanf("%d",x+i); printf("%d",x[i]); }
    return 0;
}
```

`malloc` **can be used to allocate a single variable also. Once it is allocated, it can be accessed with pointer or array notations,** `*(p+i)` **or** `p[i]`

Releasing the allocated memory: free

- An allocated memory can be returned to the system for future use by the `free` function: `free(ptr);` — here `ptr` is pointer to a memory allocated using `malloc` or `calloc` or `realloc`
- No size can be mentioned in `free`. The system remembers while allocating the memory using `malloc` or similar functions. The entire block of memory is freed that was allocated with `malloc` like call
- `ptr` must be the starting address of an allocated block. A pointer to the interior of a block cannot be passed to free
- Dynamically allocated memory stays until explicitly freed or the program terminates
- You cannot free an array `x[]` defined like `int x[20];`

Example: free

```
int main(){
   int i, n *x, s=0;
   printf("Enter no. of students\n");
   scanf("%d",&n);
   x=(int *)malloc(n*sizeof(int));
   printf("Enter marks for students\n");
   for(i=0; i<n; i++) scanf("%d",x+i);
   for(i=0; i<n; i++) s += *(x+i);
   printf("Avg: %f\n",((float )s)/n);
   free(x);
   return 0;
}
```

Altering the size: `realloc`

- Sometime we need to alter the size of some previously allocated memory space
- One can use `realloc`
- If original allocation is made as `ptr=malloc(size);` then reallocation can be done as `ptr=realloc(ptr, new_size);`
- The new memory may or may not begin at the same place as the old one. If it does not find enough space, it is allocated in an entirely new region and moves the content to new place
- The old data remains intact
- If it is unable to allocate sufficient memory, NULL will be returned

Example: realloc

```
int main(){
  int *A = (int *)malloc(10*sizeof(int )), size = 10, n = 0, x;
  printf("Keep on entering +ve integers. Enter 0 or a -ve integer to stop.\n");
  while (1) {
    printf("Next integer: ");  scanf("%d", &x);
    if(x <= 0) break;
    ++n;
    if(n > size) {
      size += 10;  A = (int *) realloc(A, size * sizeof(int ));
    }
    A[n-1] = x;
  }
  A = (int *) realloc(A, n * sizeof(int)); size = n;
  // Process the integers read from the user
  ...
  free(A);  return 0;
}
```