Pointers

March 19, 2012

Outline

Pointers

Pointers are variables, which contain the address of some other variables.

Declaration: *datatype* *pointername;
e.g. long * ptra;

The *type* of a pointer depends on the type of the variable it points to. Every pointer points to some data type.

Sizes of basic data types

All data is stored in memory. But different data types occupy different amount of memory.

The `sizeof()` operator in C can be used to determine the number of bytes occupied by each data type.

Sizes of basic data types

All data is stored in memory. But different data types occupy different amount of memory.

The `sizeof()` operator in C can be used to determine the number of bytes occupied by each data type.

For example, on some machine you may have

```
sizeof(int) = 4
sizeof(float) = 4
sizeof(double) = 8
```

Sizes of basic data types

All data is stored in memory. But different data types occupy different amount of memory.

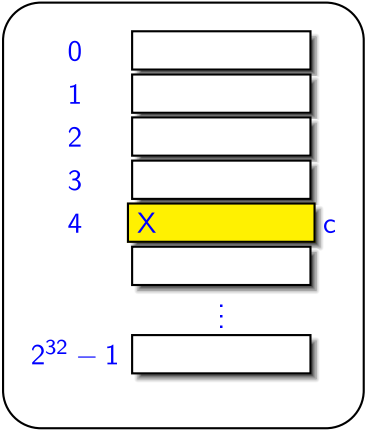
The `sizeof()` operator in C can be used to determine the number of bytes occupied by each data type.

For example, on some machine you may have

```
sizeof(int) = 4
sizeof(float) = 4
sizeof(double) = 8
```

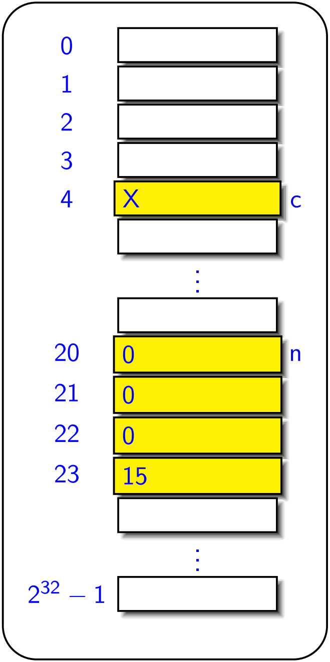
These numbers are NOT the same for all machines. You should use the `sizeof()` operator instead of assuming the value.

A Sample Program

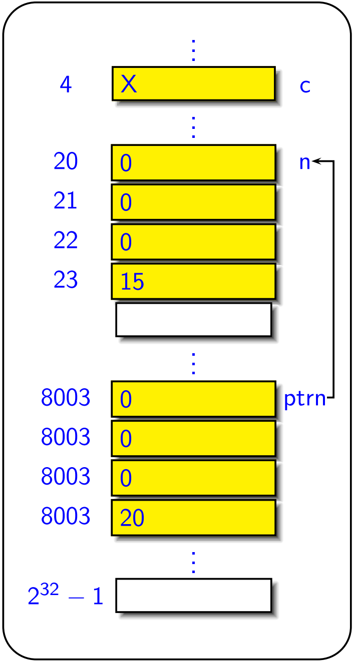


Memory Layout (Bytes)

```c
#include <stdio.h>

int main()
{
    int n;
    char c;
    int *ptrn;

    c='X';
    n=15;
    ptrn=&n;

    return 0;
}
```




```c
#include <stdio.h>

int main()
{
    char c;
    int n;
    int *ptrn;

    c='X';
    n=15;
    ptrn=&n;

    return 0;
}
```




```c
#include <stdio.h>

int main()
{
    int n;
    char c;
    int *ptrn;

    c='X';
    n=15;
    //address of n
    //sizeof(ptrn) = 4
    ptrn=&n;


    return 0;
}
```

sizeof(ptrn) = 4 bytes = 32 bits, since we have 2^{32} byte addresses.

Address Operations

There are two unary operations to consider.

- The * operator: If `ptra` is a pointer variable, then `*ptra` gives you the content of the location pointed to by ptr.

Address Operations

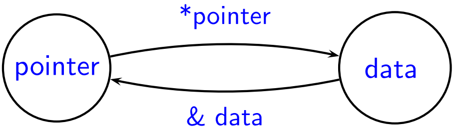
There are two unary operations to consider.

- The * operator: If `ptra` is a pointer variable, then `*ptra` gives you the content of the location pointed to by ptr.
- The & operator: If `v` is a variable, then `&v` is the address of the variable.

Address Operations

There are two unary operations to consider.

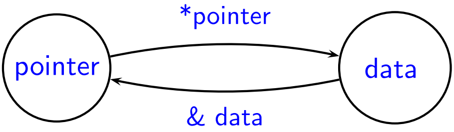
- The * operator: If `ptra` is a pointer variable, then `*ptra` gives you the content of the location pointed to by ptr.
- The & operator: If `v` is a variable, then `&v` is the address of the variable.



Address Operations

There are two unary operations to consider.

- The * operator: If `ptra` is a pointer variable, then `*ptra` gives you the content of the location pointed to by ptr.
- The & operator: If `v` is a variable, then `&v` is the address of the variable.

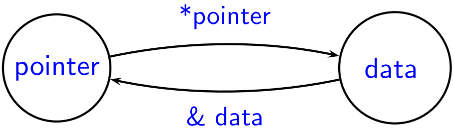


In the previous code, what is *ptrn?

Address Operations

There are two unary operations to consider.

- The * operator: If `ptra` is a pointer variable, then `*ptra` gives you the content of the location pointed to by ptr.
- The & operator: If `v` is a variable, then `&v` is the address of the variable.



In the previous code, what is *ptrn?

Caution: Declaration of a pointer also uses '*'.

Outline

Pointer Arithmetic

Problem: How do we do *relative* addressing? (for example, “next element” in an integer array)

Pointer Arithmetic

Problem: How do we do *relative* addressing? (for example, "next element" in an integer array)

C allows you to perform some arithmetic operations on pointers. (Not every operation is allowed.) Consider

```
<datatype> *ptrn; //datatype can be int, long, etc.
```

Pointer Arithmetic

Problem: How do we do *relative* addressing? (for example, “next element” in an integer array)

C allows you to perform some arithmetic operations on pointers. (Not every operation is allowed.) Consider

```
<datatype> *ptrn; //datatype can be int, long, etc.
```

Pointer Arithmetic

Problem: How do we do *relative* addressing? (for example, “next element” in an integer array)

C allows you to perform some arithmetic operations on pointers. (Not every operation is allowed.) Consider

```
<datatype> *ptrn; //datatype can be int, long, etc.
```

Unary Pointer Arithmetic Operators

Pointer Arithmetic

Problem: How do we do *relative* addressing? (for example, "next element" in an integer array)

C allows you to perform some arithmetic operations on pointers. (Not every operation is allowed.) Consider

```
<datatype> *ptrn; //datatype can be int, long, etc.
```

Unary Pointer Arithmetic Operators

- Operator ++: Adds `sizeof(datatype)` number of bytes to pointer, so that it points to the next entry of the datatype.

Pointer Arithmetic

Problem: How do we do *relative* addressing? (for example, "next element" in an integer array)

C allows you to perform some arithmetic operations on pointers. (Not every operation is allowed.) Consider

```
<datatype> *ptrn; //datatype can be int, long, etc.
```

Unary Pointer Arithmetic Operators

- Operator ++: Adds `sizeof(datatype)` number of bytes to pointer, so that it points to the next entry of the datatype.
- Operator −−: Subtracts `sizeof(datatype)` number of bytes to pointer, so that it points to the next entry of the datatype.

Pointer Arithmetic - Example 1

```c
#include <stdio.h>

int main()
{
    int *ptrn;
    long *ptrlng;

    ptrn++;        //increments by sizeof(int) (4 bytes)
    ptrlng++;      //increments by sizeof(long) (8 bytes)

    return 0;
}
```

Pointer Arithmetic - II

Pointers and integers are not interchangeable. (except for 0.) We will have to treat arithmetic between a pointer and an integer, and arithmetic between two pointers, separately.

Pointer Arithmetic - II

Pointers and integers are not interchangeable. (except for 0.) We will have to treat arithmetic between a pointer and an integer, and arithmetic between two pointers, separately.

Suppose you have a pointer to a long.

```
long *ptrlng;
```

Pointer Arithmetic - II

Pointers and integers are not interchangeable. (except for 0.) We will have to treat arithmetic between a pointer and an integer, and arithmetic between two pointers, separately.

Suppose you have a pointer to a long.

```
long *ptrlng;
```

Binary Operations between a pointer and an integer

Pointer Arithmetic - II

Pointers and integers are not interchangeable. (except for 0.) We will have to treat arithmetic between a pointer and an integer, and arithmetic between two pointers, separately.

Suppose you have a pointer to a long.

```
long *ptrlng;
```

Binary Operations between a pointer and an integer

1. ptrlng+n is valid, if n is an integer. The result is the following byte address
   `ptrlng + n*sizeof(long)`
   and not `ptrlng + n`.
   It advances the pointer by `n` number of `long`s.

Pointer Arithmetic - II

Pointers and integers are not interchangeable. (except for 0.) We will have to treat arithmetic between a pointer and an integer, and arithmetic between two pointers, separately.

Suppose you have a pointer to a long.

```
long *ptrlng;
```

Binary Operations between a pointer and an integer

1. ptrlng+n is valid, if n is an integer. The result is the following byte address
   `ptrlng + n*sizeof(long)`
   and not `ptrlng + n`.
   It advances the pointer by `n` number of `long`s.
2. `ptrlng-n` is similar.

Pointer Arithmetic - III

Consider two pointers ptr1 and ptr2 which point to the same type of data.

```
<datatype> *ptr1, *ptr2;
```

Pointer Arithmetic - III

Consider two pointers ptr1 and ptr2 which point to the same type of data.

```
<datatype> *ptr1, *ptr2;
```

Pointer Arithmetic - III

Consider two pointers ptr1 and ptr2 which point to the same type of data.

```
<datatype> *ptr1, *ptr2;
```

Binary operations between two Pointers

Pointer Arithmetic - III

Consider two pointers ptr1 and ptr2 which point to the same type of data.

```
<datatype> *ptr1, *ptr2;
```

Binary operations between two Pointers

1. Surprise: Adding two pointers together is not allowed!

Pointer Arithmetic - III

Consider two pointers ptr1 and ptr2 which point to the same type of data.

```
<datatype> *ptr1, *ptr2;
```

Binary operations between two Pointers

1. Surprise: Adding two pointers together is not allowed!
2. `ptr1 - ptr 2` is allowed, as long as they are pointing to elements of the same array. The result is

$$\frac{\texttt{ptr1 - ptr2}}{\texttt{sizeof}(\textit{datatype})}$$

In other settings, this operation is undefined (may or may not give the correct answer).

Pointer Arithmetic - III

Consider two pointers ptr1 and ptr2 which point to the same type of data.

```
<datatype> *ptr1, *ptr2;
```

Binary operations between two Pointers

1. Surprise: Adding two pointers together is not allowed!
2. `ptr1 - ptr 2` is allowed, as long as they are pointing to elements of the same array. The result is

$$\frac{\texttt{ptr1 - ptr2}}{\texttt{sizeof}(\textit{datatype})}$$

   In other settings, this operation is undefined (may or may not give the correct answer).

Why all these special cases? These rules for pointer arithmetic are intended to handle addressing inside arrays correctly.

Pointer Arithmetic - IV

If we can subtract a pointer from another, all the relational operations can be supported!

Logical Operations on Pointers

1. `ptr1` $>$ `ptr2` is the same as `ptr1 - ptr2 > 0`,
2. `ptr1` $=$ `ptr2` is the same as `ptr1 - ptr2 = 0`,
3. `ptr1` $<$ `ptr2` is the same as `ptr1 - ptr2 < 0`,
4. and so on.

Outline

Arrays and Pointers

Array names essentially are pointers. Array elements are stored in contiguous (consecutive) locations in memory.

For example, consider `int arr[10];`

1. `arr` is a pointer to the first element of the array.

Arrays and Pointers

Array names essentially are pointers. Array elements are stored in contiguous (consecutive) locations in memory.

For example, consider `int arr[10];`

1. `arr` is a pointer to the first element of the array.
2. That is, `*arr` is the same as `arr[0]`.

Arrays and Pointers

Array names essentially are pointers. Array elements are stored in contiguous (consecutive) locations in memory.

For example, consider `int arr[10];`

1. `arr` is a pointer to the first element of the array.
2. That is, `*arr` is the same as `arr[0]`.
3. `arr+i` is a pointer to `arr[i]`. (`arr+i` is equivalent to `arr+i*sizeof(int)`.)

Arrays and Pointers

Array names essentially are pointers. Array elements are stored in contiguous (consecutive) locations in memory.

For example, consider `int arr[10];`

1. `arr` is a pointer to the first element of the array.
2. That is, `*arr` is the same as `arr[0]`.
3. `arr+i` is a pointer to `arr[i]`. (`arr+i` is equivalent to `arr+i*sizeof(int)`.)
4. `*(arr+i)`, is equal to `arr[i]`.

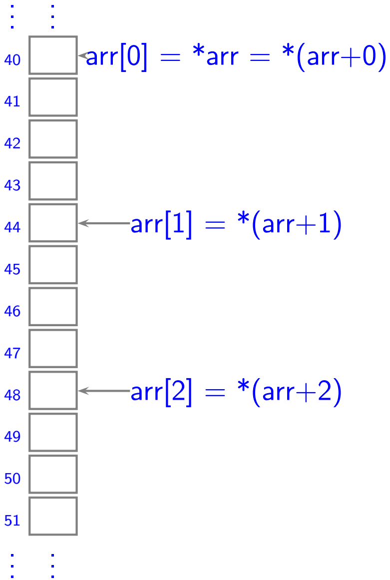
Arrays and Pointers

Array names essentially are pointers. Array elements are stored in contiguous (consecutive) locations in memory.

For example, consider `int arr[10];`

1. `arr` is a pointer to the first element of the array.
2. That is, `*arr` is the same as `arr[0]`.
3. `arr+i` is a pointer to `arr[i]`. (`arr+i` is equivalent to `arr+i*sizeof(int)`.)
4. `*(arr+i)`, is equal to `arr[i]`.
5. Question: What is `&arr[i]` equivalent to?

Arrays and Pointers - Figure




```
int arr[3];
```

Outline

Passing Pointers to Functions

Since pointers are also variables, they can be passed

- As input parameters to functions
- As return values from functions

Passing Pointers - Reason 1

Why do we pass pointer variables to functions?

Recall the swap function which took input integers. This function was unable to swap the variables inside `main()`.

Passing Pointers - Reason 1

Why do we pass pointer variables to functions?

Recall the swap function which took input integers. This function was unable to swap the variables inside `main()`.

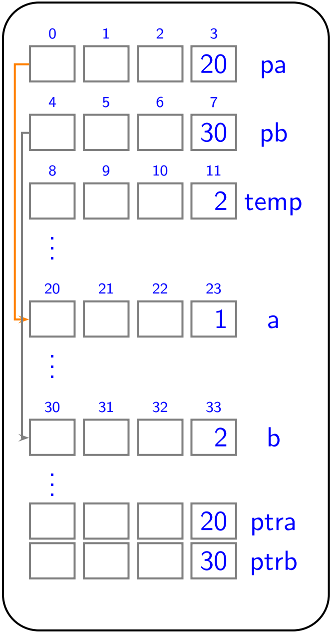
Suppose we want a swap function which is able to swap arguments inside the caller.

Main idea: Pass pointers!!

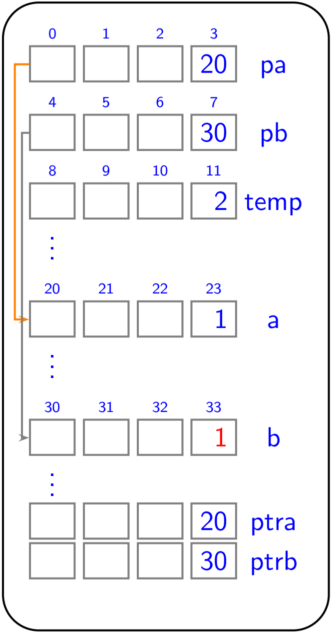
A Swap Program

```
#include <stdio.h>

//Swap the contents of locations pointed to by the
//input pointers
void swap(int *pa, int *pb)
{
    int temp;

    temp = *pb;
    *pb    = *pa;
    *pa    = temp;
    return;
}

int main()
{
   int a = 1, b = 2;
   int *ptra = &a;
   int *ptrb = &b;

   printf(``a=%d b=%d'', a, b);

   swap (ptra, ptrb);     //equivalently, swap(&a, &b);

   //a and b would now be swapped
   printf(``a=%d b=%d'', a, b);
   return 0;
}
```

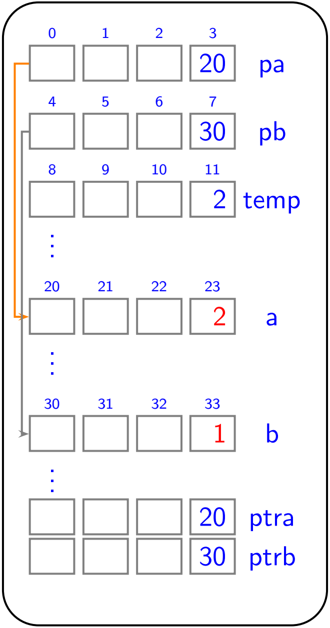
When `swap(pa, pb)` is called, the value of the pointers is copied to the function. The value of the pointers is the address of a and b, respectively.




```
#include <stdio.h>

void swap(int *pa, int *pb)
{
    int temp;
    temp = *pb;
    *pb    = *pa;
    *pa    = temp;
}

int main()
{
   int a = 1, b = 2;
   int *ptra = &a;
   int *ptrb = &b;

   swap (ptra, ptrb);
```




```
#include <stdio.h>

void swap(int *pa, int *pb)
{
     int temp;
     temp=*pb;
     *pb = *pa;
     *pa     = temp;
}

int main()
{
    int a = 1, b = 2;
    int *ptra = &a;
    int *ptrb = &b;

    swap (ptra, ptrb);
```




```c
#include <stdio.h>

void swap(int *pa, int *pb)
{
      int temp;
      temp=*pb;
      *pb = *pa;
      *pa    = temp;
}

int main()
{
    int a = 1, b = 2;
    int *pa = &a;
    int *pb = &b;

    swap (pa, pb);
```

scanf and printf

If we want to modify data in the caller, then we pass address of the variables. We can see this in the difference between `printf` and `scanf`.

scanf and printf

If we want to modify data in the caller, then we pass address of the variables. We can see this in the difference between `printf` and `scanf`.

scanf and printf

If we want to modify data in the caller, then we pass address of the variables. We can see this in the difference between `printf` and `scanf`.

scanf

```
scanf(‘‘%d’’, &n);
```

`scanf` needs to change the content of `n`. This can be done by passing the address of `n`.

scanf and printf

If we want to modify data in the caller, then we pass address of the variables. We can see this in the difference between `printf` and `scanf`.

scanf

```
scanf(``%d'', &n);
```

`scanf` needs to change the content of `n`. This can be done by passing the address of `n`.

scanf and printf

If we want to modify data in the caller, then we pass address of the variables. We can see this in the difference between `printf` and `scanf`.

scanf

```
scanf(``%d'', &n);
```

`scanf` needs to change the content of `n`. This can be done by passing the address of `n`.

printf

```
printf(``%d'',n);
```

`printf` does not need to change the content of `n`.

Passing arrays to functions

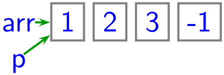
We have already seen that we can pass arrays as input to functions. We also have seen that arrays are essentially pointers.

We can pass pointers, where arrays are expected, and vice versa!

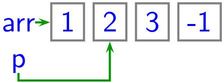
Passing arrays to functions

```
#include <stdio.h>

//Count number of elements in an integer array,
//until the first -1
int num_elts(int *a)
{
        int *p;
        p = a;

        while(*p != -1){
                p++;
        }

        return p-a;
}

int main()
{
        int arr[] = {1, 2, 3, -1};
        printf("%d", num_elts(arr)); //Passing array as pointer
        return 0;
}
```

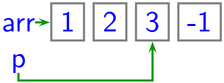
Schematic Diagram of num_elts



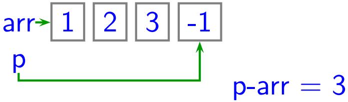
Schematic Diagram of num_elts



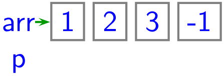
Schematic Diagram of num_elts



Schematic Diagram of num_elts



Schematic Diagram of num_elts



If we changed the call to the following line,

```
num_elts(arr+1);
```

the result is 2, since the num_elts will search in the subarray {2,3,-1}.

Passing Pointers to Functions - Another Reason

Passing a pointer to data, instead of passing the value of the data can be much faster.

This is used to reduce the slowdown due to function calling.

The decision to do this must be taken with care.

Common Mistakes in Pointer Programs

Programming with pointers has to be done with care. Common mistakes include

1. Crossing array boundaries - Suppose an array has 10 elements, and arr is pointing to the first element. If you do *(arr-1), or *(arr+11), you might get unpredictable behaviour.
2. “Dangling Pointers” - pointers that point to data that is not meaningful - for example, using a pointer without initializing it.

Debugging Pointer Programs

If there is an error in a program using pointers, when executing, you will most probably get "Segmentation Fault".

There are several ways to find the error.

1. Go through the code carefully and see if you can locate the bug. (perfect!)

1

[1]Some material in these slides has been taken from course notes by Arnab Bhattacharya.

Debugging Pointer Programs

If there is an error in a program using pointers, when executing, you will most probably get "Segmentation Fault".

There are several ways to find the error.

1. Go through the code carefully and see if you can locate the bug. (perfect!)
2. Use a debugger like `gdb` to debug the code and step through the execution to locate the error. Examine the memory contents when you debug.

1

[1] Some material in these slides has been taken from course notes by Arnab Bhattacharya.

Debugging Pointer Programs

If there is an error in a program using pointers, when executing, you will most probably get "Segmentation Fault".

There are several ways to find the error.

1. Go through the code carefully and see if you can locate the bug. (perfect!)
2. Use a debugger like `gdb` to debug the code and step through the execution to locate the error. Examine the memory contents when you debug.
3. Insert printf statements to pinpoint where the code crashes. (When doing so, make sure to put "\n" at the end of the message - it might not print otherwise!)

[1]

[1] Some material in these slides has been taken from course notes by Arnab Bhattacharya.

Debugging using printf statements - Example

```
void merge_p(int *s, int *t, int *result, int size_s, int size_t)
{
        int *p = s;
        int *q = t;

        printf("Reached Point 0\n");


        while(p-s<size_s && q-t<size_t){
            //...
        }

        printf("Reached Point 1\n");

        if(p-s < size_s){
                while( p-s < size_s) {
                    //...
                }
        }else if(q-t < size_t){
                while( q-t < size_t) {
                    //...
                }
        }

        printf("Reached Point 2\n");

        return;

}
```