

ECE264: Advanced C Programming

Summer 2019

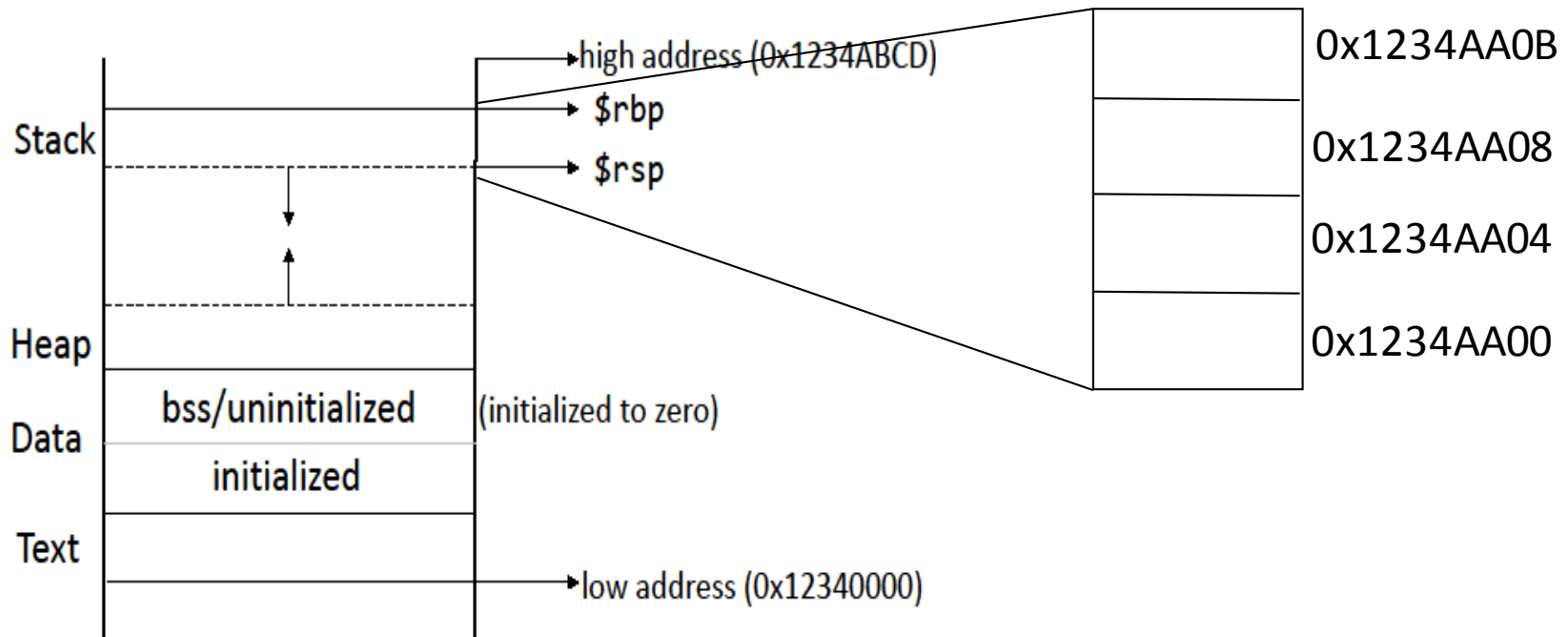
Week 2: Addresses, Pointers, Pointer Arithmetic

Addresses

- Humans are not good at remembering numerical addresses.
 - What are the GPS coordinates (latitude and longitude) of your residence?
- Addresses in computer programs are just numbers.

- Addresses in computer programs identify memory locations.
- Computer programs think and live in terms of memory locations.

Program Memory Layout - Revisited



- Every memory location is a box holding data
- Each box has an address

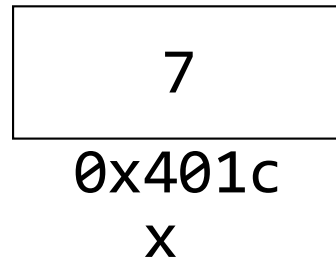
Addresses Contd..

- A program navigates by visiting one address after another.
- We (humans) choose convenient ways to identify addresses so that we can give directions to a program
 - Variables

Handles to Addresses

- What is a variable?
 - Its just a handle to an address / program memory location

- `int x = 7;`



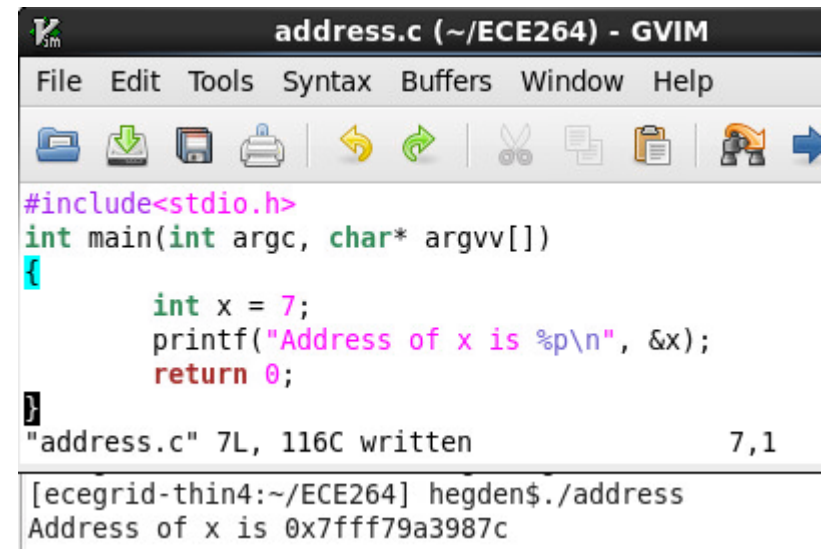
- Read x => Read the content at address 0x401C
- Write x=> Write at address 0x401C

Visualizing Addresses

- The *address of* (&) operator fetches a variable's address in C.
- &x would return the address 0x401C.
- Format specifier 'p' :

```
printf("%p\n",&x)
```

prints the Hexadecimal
address of x



```
address.c (~/ECE264) - GVIM
File Edit Tools Syntax Buffers Window Help

#include<stdio.h>
int main(int argc, char* argv[])
{
    int x = 7;
    printf("Address of x is %p\n", &x);
    return 0;
}

"address.c" 7L, 116C written 7,1

[eccegrid-thin4:~/ECE264] hegden$ ./address
Address of x is 0x7fff79a3987c
```

Pointers

- Pointer is a data type that *holds an address*.

`<type>* <pointer_name>;`

We read it as “**pointer to** `<type>`”

- Example:

- `int* p;`

is a variable named `p` whose type is pointer to `int`
OR `p` is an integer pointer

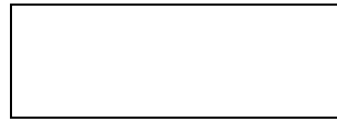
Note that the variable declared is `p`, *not* `*p`

- A pointer always stores an address
- `<type>` of the pointer tells us what kind of data is stored at that address
- Example:
 - `int* p;`

declares a pointer variable `p` holding an address, which identifies a memory location capable of storing an integer.

- `int* p;`

Remember `p` is a variable and all variables are just names identifying addresses.

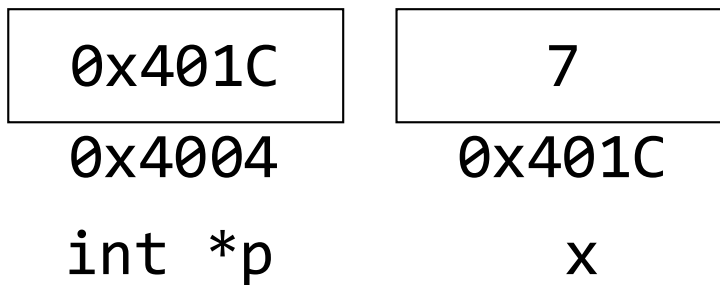


`0x4004`
`int *p`

Initializing Pointers

- `int* p=&x;`

//p holds the address of a memory location that stores an integer.



- We say *p points to x*

- Cannot assign arbitrary addresses to pointers.
- Example:
`int* p=5;`
- Operating system determines addresses available to each program.

The NULL address

- NULL is a special address

- Example

```
int* p=NULL; //p points to nowhere
```

- Useful when it is not yet known where p points to.
- Uninitialized pointers store garbage addresses

Using Pointers

- The *dereference* operator (*)
 - Lets us access the memory location at the address stored in the pointer

```
int x=7;  
int *p = &x; //p now points to x  
*p = 10; //this is the same as x=10  
int y=*p; //this is the same as y=x
```

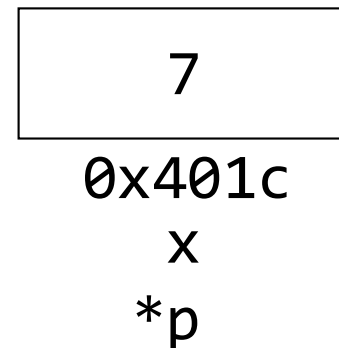
The expression `*p` is equivalent to `x`

- Pointers as alternate names to memory locations

```
int x=7;  
int *p = &x; //p now points to x  
*p = 10; //this is the same as x=10  
int y=*p; //this is the same as y=x
```

The expression `*p` is equivalent to `x`

`x` is the name for an address
`*p` is the name for an address



- Pointers as “dynamic” names to memory locations

```
int x=7;
```



0x401c

x

//x always names the location 0x401C

```
int *p = &x; // *p is now another name for x
```

```
int y = *p //like saying y=x
```

```
p = &y; // *p is now another name for y
```

```
*p=8; //like saying y=8
```


The swap function

```
int a = 8;  
int b = 10;
```

```
void swap(int x, int y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

```
void main() {  
    swap(a, b); //a is still 8, b is still 10  
}
```

Pass by value

- C functions operate on ***copies*** of arguments.
- Change the data inside the function, you change the copy. Not the original.
- In swap, x and y are names of memory locations that are copies of a and b

What if x and y held addresses of a and b?

- *x and *y would name the same memory locations that a and b did.

The swap function

```
int a = 8;  
int b = 10;
```

```
void swap(int* x, int* y) {  
    int tmp = *x; //tmp = whatever is in the  
    location that x points to.  
    *x = *y;  
    *y = tmp;  
}
```

```
void main() {  
    //remember, we have to pass addresses now,  
    not ints.  
    swap(&a, &b); //a is now 10, b is 8  
}
```

Pointers to Different Types

- What can pointers point to? any data type!
 - Basic data types,
 - Structures,
 - Functions, and
 - even Pointers!

Pointer Chains

```
int x = 7;  
int *p = x; //p points to x; *p is same  
as x.
```

```
int * * q; //q is a pointer to pointer  
to int
```

*q is same as p.

*(*q) is the same as *p, which is same as x

Pointers to Structures

```
typedef struct {  
    int year;  
    char model;  
    float acceleration; //0-60mph in seconds  
}Car;
```

```
Car t1 = {.year = 2017, .model = 'S',  
    .acceleration = 2.8 };
```

```
Car * pt1 = &t1; //now you can use *pt1  
anywhere you use t1
```

```
(*pt1).acceleration = 2.3;  
(*pt1).year = 2019;  
(*pt1).model = 'X';  
float avg_acceleration = ((*pt1).acceleration  
+ (*pt2).acceleration) / 2.0;
```

We can also use the -> operator to access structure members.

```
pt1->acceleration = 2.3;  
pt1->year = 2019;  
pt1->model = 'X'  
float avg_acceleration = (pt1->acceleration +  
pt2->acceleration) / 2.0;
```

Address of (&) operator and Type

- Adding & to a variable adds * to its type
- Example:
 - if a is an int, then &a is an int*
 - if b is an int*, then &b is an int**
 - if c is an int**, then &c is an int***
 - ...

Dereference (*) operator and Type

- Adding * to a variable subtracts * from its type
- Example:
 - if a is an int*, then *a is an int
 - if b is an int**, then *b is an int*
 - if c is an int***, then *c is an int**
 - ...

Pointers to Functions (Function Pointers)

- Every function in a C program refers to a specific address (remember disassembling code during buffer overflow attack)
- Function pointers store addresses of functions
- Syntax:

```
typedef type (*name) (argument types)
```

Function Pointers - Example

```
typedef void (*myfuncptr) (int, int)
```

- `myfuncptr` is a pointer to a function that returns an `int` and accepts two arguments of type `int`.

Function Pointers - Example

```
void swap(int x, int y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

```
myfuncptr ptrswap = swap; //initialization.
```

```
int main(int argc, char* argv[]) {  
    int a=10;  
    int b=20;  
    ptrswap(a,b); //swap called by a function  
    pointer  
}
```

Function Pointers

How about these?

```
(*ptrswap)(a,b);
```

```
(****ptrswap)(a, b)
```

C says dereferencing a function pointer returns a function pointer. Behavior different from normal '&' and '' operators.*