

Welcome!

- welcome to C-C++-R!
- what I don't want this course to be:
 - lecture is a process where notes of the teacher gets transferred to the notes of the student without going through the mind of either
- what would I like instead:
 - a very interactive learning by sharing (from both sides of the table) kind of thing

Disclaimer

- “I don’t know / remember the syntax of blah” means “I don’t have it on top of my head but could and would look it up for you or ask you to do the same”, one head, multiple languages + Statistics, not a happy crowd, trust me!
- do I make mistakes on the board / on slides? YES, plenty, sometimes very confidently! so, argue till the death of my ego, truth/fact will win, I guarantee (even if it takes days!)
- do I love emails? NO! caveat: only to fix appointments to discuss problems in detail
- emailing code and asking me (for that matter anybody) to debug, is it a good idea? NO! meeting works great.

C-C++

- C: developed at Bell Labs during the early 1970's by Dennis Ritchie
 - best *reference*: “The C Programming Language (2e)” by Kernighan and Ritchie
- C++: developed at Bell Labs during the early 1980's by Bjarne Stroustrup
 - best *reference*: “The C++ Programming Language (3e)” by Bjarne Stroustrup
 - why is it called C++: it stands for $C = C + 1$, what?

Programming Quotes

- “C is quirky, flawed and an enormous success” --Dennis Ritchie
- “ C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg” --Bjarne Stroustrup
- “The joy of writing a good piece of code shouldn’t last long!” --gopi!
- more quotes? go to: <http://www.sysprog.net/quotec.html>
- a fun one: “Java is, in many ways, C++--” --Michael Feldman
 - I mention the above quote: does not imply I hate Java!

R/S-PLUS

- S: developed at Bell Labs during the early 1980's by John Chambers (he is Harvard Stats grad!)
 - best *reference*: “The New S Language” by R. A. Becker, J. M. Chambers and A. R. Wilks, known as the “blue book”
- R: a GNU project which implements the S in C, its a free software
 - best *reference*: the manual section of <http://www.r-project.org/>
- S-PLUS: commercial sister of R, also implmented in C
 - best *reference*: “Modern Applied Statistics with S (4e)” by William N. Venables and Brian D. Ripley

Base Systems

- “bits” are the smallest unit of storage
- base-2: inside a computer all the numbers $[0 - 9]$ and letters $[a - zA - Z]$ (and special characters) are stored in binary (base-2) format internally using a “byte” i.e. a consecutive array of 8 “bits”
 - so there are $2^8 = 256$ different characters
- base-16: a hexadecimal (base-16) format is used to “nagivate” inside the processor, the brain of the computer
 - hexadecimal digits: $0, 1, \dots, 9, a, b, c, d, e, f$

Address Space

- address space of a 32-bit processor: (each cell below is a byte)

4294967295	0xffffffff	<input type="text"/>
4294967294	0xfffffff	<input type="text"/>
⋮	⋮	<input type="text"/>
⋮	⋮	<input type="text"/>
0	0x00000000	<input type="text"/>

- note number of possible addresses: 2^{32} and the highest possible address: $2^0 + 2^1 + \dots + 2^{31} = 2^{32} - 1 = 4294967295 = 0xffffffff$, which has 1s in all the 32 bits
- we enumerate these possible addresses as: $0, 1, \dots, 4294967295$ or as: $0x00000000, 0x00000001, \dots, 0xffffffff$
- so we have 2^{32} many addresses which gives us as many bytes:
 $2^{32}B = 2^{22}KB = 2^{12}MB = 2^2GB = 4GB$

Variables I

- variable declaration:

```
int foo = 0, bar = 1;
```

foo	bar
name: ``foo``	name: ``bar``
address: 0xdeadbeef	address: 0xdeaddeed
value: 0	value: 1

- so, a variable has a name, an address and a value
- it has a type too, for the time being pretend we live in the world of integers or ints *only* and so we do not carry around the type!

Variables II

- `for int foo = 0;`

foo

name: <code>''foo''</code>
address: <code>0xdeadbeef</code>
value: <code>0</code>

- how to get the value? use the name: `foo` fetches `0`
- how to get the address? use the name and the *address operator* or the *reference operator*: `"&"`, `&foo` fetches `0xdeadbeef`

An Example I

- first complete program: simple printing

```
#include <stdio.h>

int
main (int argc, char **argv)
{
    int ivall = 1;

    /* This is a simple
       multi-line or block comment
    */
    // This is another simple one-line comment
    printf("ivall:value = %d\n", ivall);
    printf("ivall:address = %p\n", &ivall);

    return 0;
}
```

- write the above lines (program) to a file, say, prog1.c and issue the following at the prompt:
 - gcc prog1.c
 - ./a.out (Linux/UNIX/OSX) or ./a.exe (Cygwin on Windows)

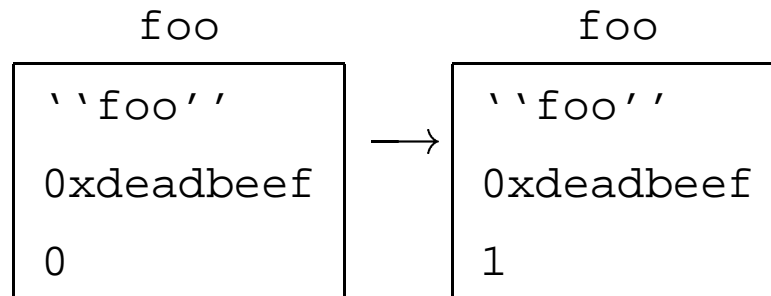
An Example II

- quick comments about the program:
 - `#include <stdio.h>` is always the first “non-empty” line in a C program, its does “something important”!
 - `main` is a sacred keyword, its a name of a function (coming much later) too!
 - “`;`”s are very important: they are “expression” delimiters
 - multi-line or block comments: use `/* blah blah */`
 - single line comments: use `// blah`
 - `printf` is the “thing” (actually a function!) you use to print “stuff”
 - `return 0;` as always the last-but-one line of `main`
 - “`{`” and “`}`” are used to “blockify” a bunch of “expressions”

Variables III

- variable manipulation:

```
int foo = 0, bar = 1; foo = bar;
```



Pointers I

- pointer is a special kind of variable whose value is an address
- pointer declaration:

```
int *fooPtr = NULL;
```

fooPtr

name: ``fooPtr``
address: 0xdeadfeed
value: NULL

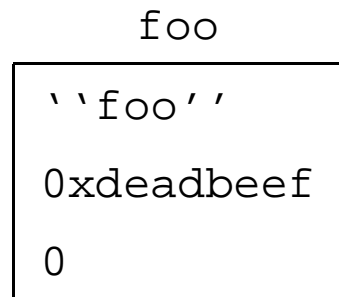
- special type of declaration: “*” is used to declare a pointer
- NULL is a special keyword and an address: the 0 in the address world, if you will

Pointers II

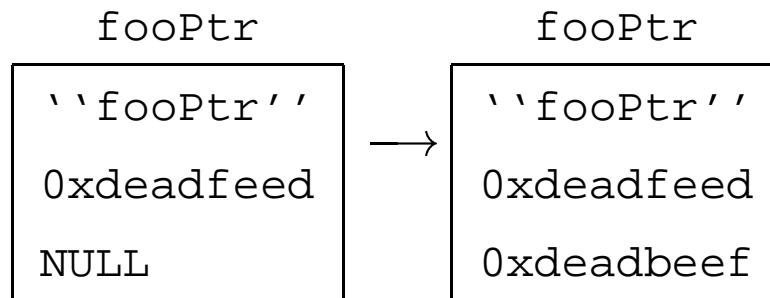
- pointer manipulation:

```
int foo = 0, *fooPtr = NULL;
fooPtr = &foo;
```

- recap foo is:



- then we have:

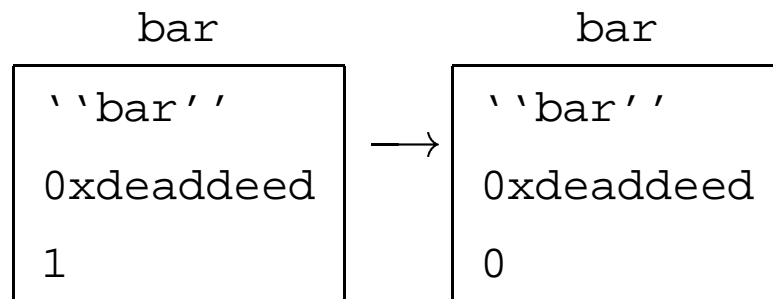


- we say `fooPtr` *points to* the address `0xdeadbeef` or the variable `foo`

Pointers III

- the *dereferencing operator*: “*”, same character used for pointer declaration but have a different job

```
int foo = 0, bar = 1, *fooPtr = NULL;
fooPtr = &foo;
bar = *fooPtr;
```

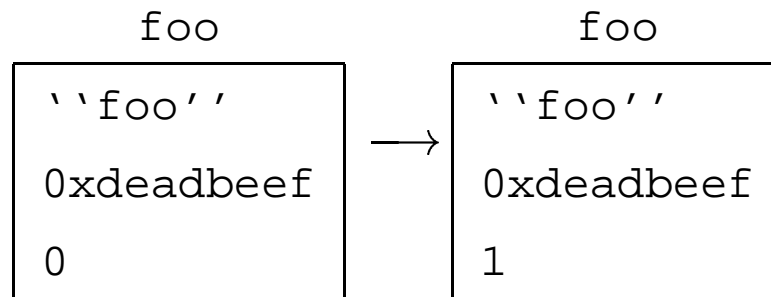


- so, *fooPtr fetches the value stored at the address which is the value of fooPtr:
 - value of fooPtr: 0xdeadbeef i.e. fooPtr is 0xdeadbeef
 - value stored at 0xdeadbeef: 0, i.e. *fooPtr is 0
- so, bar now has the value 0

Pointers IV

- another use of the *dereferencing operator*: “*”

```
int foo = 0, bar = 1, *fooPtr = NULL;  
fooPtr = &foo;  
*fooPtr = bar;
```



- so, the value `*fooPtr` is changed to the value of `bar`
- thus, `foo` would have value 1 as opposed to 0, why?

Pointers V

- pointer arithmetic:

```
int foo = 0, *fooPtr = NULL;
fooPtr = &foo;
```

- what would `fooPtr + k` (k , any positive number, e.g. 1 or 10) mean?
 - go to the address pointed to by `fooPtr` which is `0xdeadbeef`
 - now because `fooPtr` is a pointer to an `int` move as many bytes forward as needed to store k many integers and report the address
 - fact: we need 4 bytes to store an `int`
 - so here `fooPtr + k` would give us the address `0xdeadbeef + 4 * k`, whatever that comes out to be in hexadecimal form
 - e.g. for $k = 1$, `fooPtr + k` would be `0xdeadbeef + 4` i.e. `0xdeadbef3`

Arrays I

- array declaration:

```
int fooArr[10];
```

fooArr:

addresses:	fooArr	fooArr + 1	...	fooArr + (10 - 1)
values:	fooArr[0]	fooArr[1]	...	fooArr[10 - 1]

- here `fooArr`, `fooArr + 1`, ..., `fooArr + 9` are addresses of the elements and hence are pointers!
- here `fooArr[0]`, `fooArr[1]`, ..., `fooArr[9]` are the values stored in the addresses i.e. values of the elements
- note the index varies between 0 and 9 as opposed to between 1 and 10!

Arrays II

- so for

```
int fooArr[10];
```

fooArr:

addresses:	fooArr	fooArr + 1	...	fooArr + (10 - 1)
values:	fooArr[0]	fooArr[1]	...	fooArr[10 - 1]

- what are `*fooArr`, `*(fooArr + 1)`, ...?
- the value `*fooArr` should be same as `fooArr[0]`
- the value `*(fooArr + 1)` should be same as `fooArr[1]` and so on
- so `fooArr[i]` is short for `*(fooArr + i)`

Arrays III

- array initialization:

```
int fooArr1[] = { 1, 12, 123 };  
int fooArr2[3];
```

```
fooArr2[0] = 1; fooArr2[1] = 12; fooArr2[2] = 123;
```

- so we can either do one of:
 - initialize in the declaration statement and not specify the number of elements of the array
 - declare the array with its length and then assign values to each of the elements

Pointers and Arrays I

- arrays are pointers!

```
int fooArr1[] = { 1, 12, 123 }, *fooArr2 = NULL;  
fooArr2 = fooArr1;
```

- remember `fooArr1` is the address of the first element so
`fooArr2 = fooArr1` makes `fooArr2` point to the same element as well
- `fooArr1[i]` and `fooArr2[i]` for all `i` would give us the same values
- we can change the elements of `fooArr1` by changing the elements of `fooArr2`

Pointers and Arrays II

- example:

```
int
main (int argc, char **argv)
{
    int *fooPtr = NULL;
    int fooArr[SMALL_LEN] = { 1, 12 };

    printf("fooArr: [ %d, %d ]\n", fooArr[0], fooArr[1]);
    printf("fooArr: [ %d, %d ]\n", *fooArr, *(fooArr + 1));

    fooPtr = fooArr;
    printf("fooPtr: [ %d, %d ]\n", *fooPtr, *(fooPtr + 1));

    *fooPtr = 12;
    *(fooPtr + 1) = 1;
    printf("fooPtr: [ %d, %d ]\n", *fooPtr, *(fooPtr + 1));
    printf("fooArr: [ %d, %d ]\n", fooArr[0], fooArr[1]);

    return 0;
}
```

- any guesses as to the output of this? lets read line-by-line

A Special Type of Pointer

- `void *` is a special type of pointer (and a keyword, coming later!)
- its used to refer to a pointer of any type, e.g. `int *`, `char *` `float *`, `double *` etc.
- use of this coming later

Builtin Types I

- so far we have only talked about `int`, there are other types too:

<code>char</code>	character	8 bits (#)
<code>short</code>	small integer	16 bits
<code>int</code>	integer	32 bits (#)
<code>long</code>	large integer	32 bits
<code>float</code>	real number	32 bits
<code>double</code>	high-precision real number	64 bits (#)
<code>long double</code>	very high-precision real number	128 bits
<code>pointer</code>	variable holding address	32 bits (#)

Table 1: Usual storage on a 32 bit machine (# ed types would be mostly used)

Builtin Types II

- get the storage sizes: use the sizeof function:

```
int
main (int argc, char **argv)
{
    printf("size of a char is %d\n", sizeof(char));
    printf("size of a short is %d\n", sizeof(short));
    printf("size of a int is %d\n", sizeof(int));
    printf("size of a long is %d\n", sizeof(long));
    printf("size of a float is %d\n", sizeof(float));
    printf("size of a double is %d\n", sizeof(double));
    printf("size of a long double is %d\n", sizeof(long double));
    printf("size of a void * is %d\n", sizeof(void *));

    return 0;
}
```

Printing Builtin Types

- the following modifiers are needed:

char	%c
char *	%s
short	%d
int	%d
long	%ld
float	%f or %g or %e
double	%f or %g or %e
long double	%lf or %lg or %le
pointer	%p

Table 2: Standard format characters

Printing Special Characters

- the following modifiers are needed:

newline	\n
tab	\t
vertical tab	\v
carriage return	\r
backslash	\\
double quote	\"
percent sign	%%
alert	\a
backspace	\b

Table 3: Format characters for special characters

Code Files

prog1.c
prog11.c