Lecture 3: Introduction to C

Dr. Naureen Nizam
nnizam@cs.toronto.edu

CSCB09H3: Software Tools and Systems Programming (SY 110)
Department of Computer and Mathematical Science
Jan 21/23, 2019

UTSC Tomorrow is created here.
UNIVERSITY OF TORONTO SCARBOROUGH
1265 Military Trail, Toronto, Ontario M1C 1A4

Administrivia

- Assignment 1
  - Posted & Due: Sunday, Jan 27, 2019 at 11:59PM (SVN)
- Labs
  - Lab 1 solution posted
  - Lab 2 posted
- Piazza
  - piazza.com/utoronto.ca/winter2019/cscb09h3
- Midterm
  - Monday, **February 25** (in-class) both Lec 01 and Lec 30
  - Make-up (only those approved) – Wednesday **February 27th** (in-class)

Recap

- Shell Programming
  - Shell Scripts (.sh)
  - PATH variable
  - Variables (scope, quotes)
  - Input/Output
  - Command-line Arguments
  - Control flow (if, while, do loops)
  - Functions (scope)

Recall positional Parameters

	What it references
$0	Name of the script
$#	Number of positional parameters
$*	Lists all positional parameters
$@	Same as $* except when in quotes
"$*"	Expands to a single argument ("$1 $2 $3")
"$@"	Expands to separate arguments ("$1" "$2" "$3")
$1 .. $9	First 9 positional parameters
${10}	10th positional parameter

Subroutines

- You can create your own functions or subroutines:

```
myfunc() {
  arg1=$1
  arg2=$2
  echo $globalvar $arg1
  return 100
}
globalvar="I like to eat"
myfunc pizza spaghetti
echo $?
echo $arg2
```

- Notes:
  - Arguments are passed through positional parameters.
  - Variables defined outside the function are visible within.
  - Variables defined inside the function are visible outside
  - Return value is stored in $?

```
I like to eat pizza
100
spaghetti
```


A subtlety about return values

- What if we switch the last two commands?

```
myfunc() {
  arg1=$1
  arg2=$2
  echo $globalvar $arg1
  return 100
}
globalvar="I like to eat"
myfunc pizza spaghetti
echo $arg2
echo $?
```

- Notes:
  - Arguments are passed through positional parameters.
  - Variables defined outside the function are visible within.
  - Variables defined inside the function are visible outside
  - Return value is stored in $?

```
I like to eat pizza
spaghetti
0
```

Shell programming

- Shell scripts are useful for automating a series of commands
- But somehow this does not really feel like programming yet…
- What is missing?
  - ~~Variables~~
  - ~~Input/output~~ (e.g. printing to the screen, reading from the screen)
    - ~~echo~~
    - ~~read~~
  - ~~Command-line arguments~~
  - ~~Control flow~~
    - ~~If statements~~
    - ~~For loops~~
    - ~~While loops~~
  - ~~Functions~~

DONE!

Today

- Intro to C
  - Basic structure of a C program
  - Variables and data types
    - Pointers!!!
- Ready for take-off?

Origins of C

- C is a by-product of UNIX, developed at Bell Laboratories by Ken Thompson, Dennis Ritchie, and others.
- Thompson designed a small language named B.
- B was based on BCPL, a systems programming language developed in the mid-1960s.

Origins of C

- By 1971, Ritchie began to develop an extended version of B.
- He called his language NB (“New B”) at first.
- As the language began to diverge more from B, he changed its name to C.
- The language was stable enough by 1973 that UNIX could be rewritten in C.

Why should you learn C?

The Tiobe index for the popularity of programming languages:

Position Sep 2012	Position Sep 2011	Delta in Position	Programming Language	Ratings Sep 2012	Delta Sep 2011	Status
1	2	↑	C	19.295%	+1.29%	A
2	1	↓	Java	16.267%	-2.49%	A
3	6	↑↑↑	Objective-C	9.770%	+3.61%	A
4	3	↓	C++	9.147%	+0.30%	A
5	4	↓	C#	6.596%	-0.22%	A
6	5	↓	PHP	5.614%	-0.98%	A
7	7	=	(Visual) Basic	5.528%	+1.11%	A
8	8	=	Python	3.861%	-0.14%	A
9	9	=	Perl	2.267%	-0.20%	A
10	11	↑	Ruby	1.724%	+0.29%	A
11	10	↓	JavaScript	1.328%	-0.14%	A
12	12	=	Delphi/Object Pascal	0.993%	-0.32%	A
13	14	↑	Lisp	0.969%	-0.07%	A
14	15	↑	Transact-SQL	0.875%	+0.02%	A

Why should you learn C?

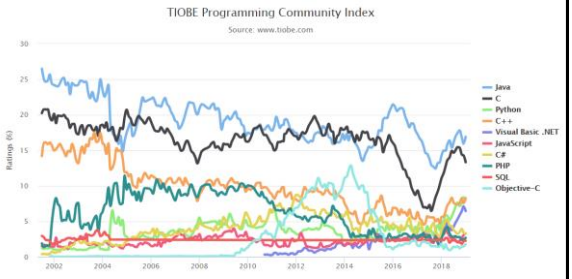
Jan 2016	Jan 2015	Change	Programming Language	Ratings	Change
1	2	↑	Java	21.465%	+5.94%
2	1	↓	C	16.036%	-0.67%
3	4	↑	C++	6.914%	+0.21%
4	5	↑	C#	4.707%	-0.34%
5	8	↑	Python	3.854%	+1.24%
6	6		PHP	2.706%	-1.08%
7	16	⇈	Visual Basic .NET	2.582%	+1.51%
8	7	↓	JavaScript	2.565%	-0.71%
9	14	⇈	Assembly language	2.095%	+0.92%
10	15	⇈	Ruby	2.047%	+0.92%
11	9	↓	Perl	1.841%	-0.42%
12	20	⇈	Delphi/Object Pascal	1.786%	+0.95%
13	17	⇈	Visual Basic	1.684%	+0.61%
14	25	⇈	Swift	1.363%	+0.62%
15	11	⇊	MATLAB	1.228%	-0.16%
16	30	⇈	Pascal	1.194%	+0.52%
17	82	⇈	Groovy	1.182%	+1.07%
18	3	⇊	Objective-C	1.074%	-5.88%

Why should you learn C?

May 2018	May 2017	Change	Programming Language	Ratings	Change
1	1		Java	16.380%	+1.74%
2	2		C	14.000%	+7.00%
3	3		C++	7.668%	+2.92%
4	4		Python	5.192%	+1.64%
5	5		C#	4.402%	+0.95%
6	6		Visual Basic .NET	4.124%	+0.73%
7	9	⌃	PHP	3.321%	+0.63%
8	7	⌄	JavaScript	2.923%	-0.15%
9	-	⌃⌃	SQL	1.987%	+1.99%
10	11	⌃	Ruby	1.182%	-1.25%
11	14	⌃	R	1.160%	-1.01%
12	18	⌃⌃	Delphi/Object Pascal	1.012%	-1.03%
13	8	⌄⌄	Assembly language	0.998%	-1.86%
14	16	⌃	Go	0.970%	-1.11%
15	15		Objective-C	0.939%	-1.16%
16	17	⌃	MATLAB	0.929%	-1.13%
17	12	⌄⌄	Visual Basic	0.915%	-1.43%
18	10	⌄⌄	Perl	0.909%	-1.69%
19	13	⌄⌄	Swift	0.907%	-1.37%
20	31	⌃⌃	Scala	0.900%	+0.18%

Why should you learn C?

Jan 2019	Jan 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.904%	+2.69%
2	2		C	13.337%	+2.30%
3	4	⌃	Python	8.294%	+3.62%
4	3	⌄	C++	8.158%	+2.55%
5	7	⌃	Visual Basic .NET	6.459%	+3.20%
6	6		JavaScript	3.302%	-0.16%
7	5	⌄	C#	3.284%	-0.47%
8	9	⌃	PHP	2.680%	+0.15%
9	-	⌃⌃	SQL	2.277%	+2.28%
10	16	⌃⌃	Objective-C	1.781%	-0.08%
11	18	⌃⌃	MATLAB	1.502%	-0.15%
12	8	⌄⌄	R	1.331%	-1.22%
13	10	⌄	Perl	1.225%	-1.19%
14	15	⌃	Assembly language	1.196%	-0.86%
15	12	⌄	Swift	1.187%	-1.19%
16	19	⌃	Go	1.115%	-0.45%
17	13	⌄⌄	Delphi/Object Pascal	1.100%	-1.28%
18	11	⌄⌄	Ruby	1.097%	-1.31%



C-Based Languages

- ***C++*** includes all the features of C, but adds classes and other features to support object-oriented programming.
- ***Java*** is based on C++ and therefore inherits many C features.
- ***C#*** is a more recent language derived from C++ and Java.
- ***Perl*** has adopted many of the features of C.

The C Programming Language

- C is a low-level language — machine access
- C is a high-level language — structured
- C is a small language, extendable with libraries
- C is permissive: assumes you know what you're doing
  - Good: efficient, powerful, portable, flexible
  - Bad: easy to make errors

A first program (gcd.c)

```
#include <stdio.h>

int gcd (int x, int y);

int main() {
  int i;
  for (i = 0; i < 20; i++) {
    printf("gcd of 12 and %d is %d\n", i, gcd(12,i));
  }
  return 0;
}
```

The rest of the file

```
int gcd(int x, int y) {
  int t;
  while (y) {
      t = x;
      x = y;
      y = t % y;
  }
  return (x);
}
```


Functions

- Every C program needs special function: `main()`
  - Returns an `int`, the exit status
- Functions must be:
  - declared before first use: tells compiler how to use it
  - defined: creates the function
- #include …
  - Adds declaration of external functions
  - E.g. `printf` function is not built-in to C (implemented in external library)

Control structures

- For-loop, while-loop, if-statement: like Java
  - Body is one statement
  - Braces { } enclose blocks
  - Blocks define scope levels
  - Can't mix declarations and non-declaration in a block
    `for (int i …` - illegal

Variables

- Need to be declared
- Have no default value
  - If not initialized, no guarantees about content …
- Only visible inside the block they were declared in
  - Exception: *global* variables declared outside of `main`
- One function's variables are not visible to other functions (i.e. they are *local*)
- Compile: `gcc -Wall -o gcd gcd.c`

Data types

- Basic data types
  - `char`
  - `int`
  - `long`

  } Integers
  - `float`
  - `double`

  } Real numbers
- `char`, `int`, `long` use the same representation, but vary in their size/capacity.
- A double has higher floating point precision than float.
- *Typical* (not guaranteed) sizes on a 32-bit machine:
  - char: 8 bits
  - int : 32 bits
  - long: 64 bits
  - float: 32 bits
  - double: 64 bits

In an assignment statement make sure that the variable on the left is at least as wide as the expression on the right!

Wait… characters are integers?

- There is an internal mapping between chars and ints, e.g. using the ASCII standard
- E.g. `char c = 'a'` is identical to `char c = 97`

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r

Other data types

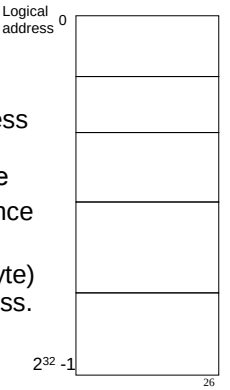
•Arrays
•Pointers
•Structs

•(Enumerations – won't cover those)
•(Unions – wont' cover those)

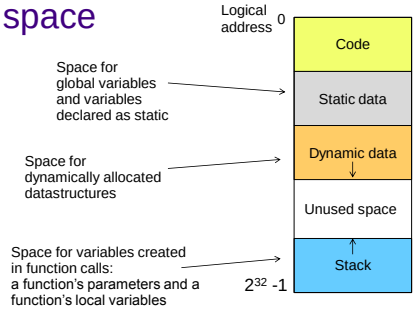
•Need some prep work on how memory is used to understand arrays and pointers

Memory model

- The memory for a process (a running program) is called it's address space
- Memory is just a sequence of bytes
- A memory location (a byte) is identified by an address.



The address space

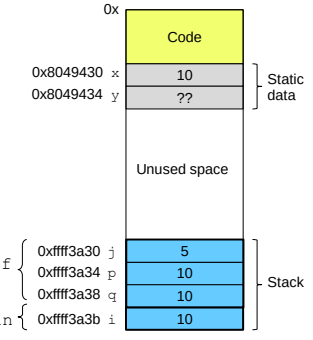


An example

```
int x = 10;
int y;

int f(int p, int q) {
    int j = 5;
    return p * q + j;
}

int main() {

    int i = x;
    y = f(i, i);
    return 0;
}
```

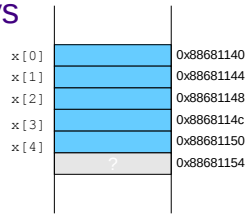
If f() were to modify p or q, will that change the value of main's `int i`?

If `f()` were to modify `x` or `y` would this change be permanent?



Arrays

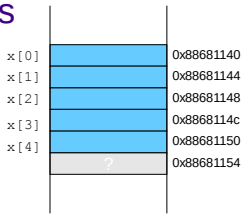
```
int x[5];
for (i = 0; i < 5; i++) {
    x[i] = i*i;
}
```



- Arrays in C are a contiguous chunk of memory that contain a list of items of the same type.
- If an array of ints contains 10 ints, then the array is 40 bytes. There is nothing extra.
- In particular, the size of the array is not stored with the array. There is *no* runtime checking.

Arrays

```
int x[5];
for (i = 0; i <= 5; i++) {
    x[i] = i*i;
}
```



- No runtime checking of array bounds
- Behaviour of exceeding array bounds is "undefined"
  - program might appear to work
  - program might crash
  - program might do something apparently random

Initializing arrays

DeclarationDefinition

```
int a[10]; /*declare a as an
             array of 10 ints*/
```

Initialization loop:

```
for(i = 0; i < N; i++) {
   a[i] = 0;
}
```

Static initialization:

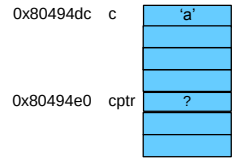
```
int numbers[4] = {1, 2, 3, 4};
char letters[4] = {'a', 'q', 'e', 'r'};
```


Pointers

- A pointer is a higher-level version of an address.
- Declaring a pointer:
  - `char *cptr; // declares a pointer to a char`
- Assigning a value to a pointer:
  - `char c = 'a';`
  - `cptr = &c;`
    - `cptr` gets the value of the address of `c`
    - the value stored in the variable `cptr` is the address of the memory location where variable `c` is stored;
- Dereferencing a pointer
  - `*cptr = 'b';`
    - Stores `'b'` at the memory address that is stored in `cptr`.
  - `char d = *cptr;`
    - Takes the contents of the memory address stored in cptr and stores them in the variable d

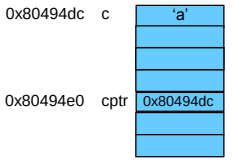
Pointer example

```
char *cptr;
char c = 'a';
cptr = &c;
*cptr = 'b';
```



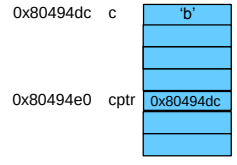
Pointer example

```
char *cptr;
char c = 'a';
cptr = &c;
*cptr = 'b';
```



Pointer example

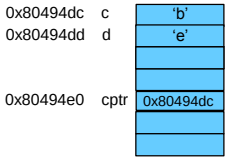
```
char *cptr;
char c = 'a';
cptr = &c;
*cptr = 'b';
```



Let's assume we had defined another `char` called `d`.

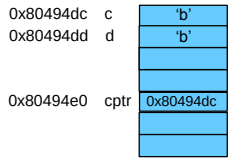
Pointer example

```
char *cptr;
char d = 'e';
char c = 'a';
cptr = &c;
*cptr = 'b';
d = *cptr;
```



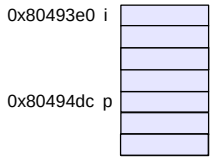
Pointer example

```
char *cptr;
char d = 'e';
char c = 'a';
cptr = &c;
*cptr = 'b';
d = *cptr;
```



Another example

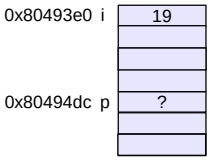
```
int i = 19;
int *p;
*p = i;
```



Do you see a problem with the code above?

Another example

```
int i = 19;
int *p;
*p = i;
```



p is not initialized and there is no memory allocated to store what the pointer points too!

Important!

- `int *p;`
- Memory is allocated to store the **pointer**
- No memory is allocated to store what the pointer points to!
- Also, `p` is **not** initialized to a valid address or null.
- I.e., `*p = 10;` is wrong unless memory has been allocated and p set to point to it.

Summary: Pointer Setup & Use

- Step 1: Create a pointer of the proper type

```
float *p;
```

- Step 2: Assign it to a variable's memory location:

```
p = &a;
```

- Step 3: Use the pointer

```
printf("%.0f",*p);
```

Note:

1. Without asterisk, pointer references an address
2. With asterisk, pointer references the value at that address
3. Always use the same type of pointer as the variable it examines (i.e., *ints for ints*, *chars for chars*)
4. Initialize the pointer before using it. Set it to an address of some variable.

Arrays vs. Pointers

- An array name in expression context decays into a pointer to the zero'th element.
- E.g.

```
int a[3] = {1, 3, 5};
int *p = a;  // same as p = &a[0];
p[0] = 10;
printf("%d %d\n", a[0], *p);
```

Pointer Arithmetic

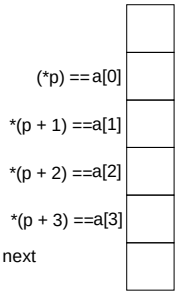
- The array access operator [ ] is really only a shorthand for pointer arithmetic + dereference
- These are equivalent in C:

```
a[i]  ==  *(a + i)
```

- C translates the first form into the second.
  - pointers and arrays are nearly the same in C!

Example

```
int a[4] = {0, 1, 2, 3};
int *p
p = a;
int i = 0;

for(i = 0; i < 4; i++) {
  printf("%d\n", *(p + i));
}
```



Why does adding 1 to `p` move it to the next spot for an int, when an int is 4 bytes?

Pointer Arithmetic

- Pointer arithmetic respects the type of the pointer.
- E.g.,

```
int i[2] = {1, 2};
int *ip;
ip = i;
*(ip + 1) += 2;
```

(really puts 4 in i[1])

```
char c[2] = {'a','z'};
char *cp;
cp = c;
*(cp + 1) = 'b';
```

(really puts b to c[1])

- C knows the size of what is being pointed at from the *type* of the pointer.

Passing Arrays as Parameters

```
int main()
{
   int i[3] = {10, 9, 8};
   printf("sum is %d\n", sum(i)); /*??*/
   return 0;
}
int sum( What goes here? ) {
}
```

- What is being passed to the function is the name of the array which decays to a pointer to the first element – a pointer of type int.

Passing Arrays as Parameters

```
int sum( int *a ) {
   int i, s = 0;
   for(i = 0; i < ??; i++)
      s += a[i];  /* this is legal */
   return s;
}
```

`sizeof(a)==4` since `a` is just a pointer here

- How do you know how big the array is?
- Remember that arrays are not objects, so knowing where the zero'th element of an array is does not tell you how big it is.
- Pass in the size of the array as another parameter.

Array Parameters

```
int sum(int *a, int size)
```

- Also legal is:

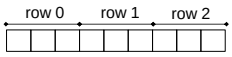
```
int sum(int a[], int size)
```

- Many advise against using this form.
  - You really are passing a pointer-to-int not an array.
  - You still don't know how big the array is.
  - Outside of a formal parameter declaration `int a[];` is illegal

⇨ `int a;` and `int a[10];` are completely different things

Multi-dimensional arrays

- Remember that memory is a sequence of bytes.




```
int a[3][3] = {  {0, 1, 2},
                 {3, 4, 5},
                 {6, 7, 8}};
```

- Arrays in C are stored in row-major order
- row-major access formula

```
int *x = (int *)a;
x[i][j] == *(x + i * n + j)
```

where `n` is the row size of `x`

But use array notation!

Summary

- The name of an array can also be used as a pointer to the zero'th element of the array.
- This is useful when passing arrays as parameters.
- Use array notation rather than pointer arithmetic whenever you have an array.

That's it for today!