Lecture 1
C primer
What we will cover

- A crash course in the basics of C
- You should read the K&R C book for lots more details
- Various details will be exemplified later in the course

Outline

- Overview comparison of C, Java and Python
- Hello world
- Preprocessor
- Command line arguments
- Arrays and structures
- Pointers and dynamic memory

Operators in Python and C

Python	C	Comment
`+, -, *, /, %, >>, <<`	The same	
`**`	Does not exist	Have to use function
Does not exist	`++, --`	`x++, ++x`
`=, +=, *=, ....`	The same	
`<, <=, >, >=, ==, !=`	The same	
`<>`	Does not exist	Use !=
`&, |, ^`	The same	
`and, or, not`	`&&, ||, !`	
`in`	Does not exist	Have to use function
Does not exist	`? :`	Conditional

If-statements and blocks in Python and C

Python	C	Comment
`if test1:`	`if (test1) {`	`{ ...`
`statement1`	`statement1;`	`... }` is a block, it
`statement2`	`statement2; }`	can be used to put several
`elif test2:`	`else if (test2)`	statements where one
`statement3`	`statement3;`	statement is expected.
`else`	`else {`	In Python this is done by
`statement4`	`statement4;`	indentation. In C identation
`statement5`	`statement5; }`	is purely for readability.

Statements in C are ended by ;.

Other control flow in Python and C

Python	C	Comment
`while test:` `   statement`	`while (test)` `  statement;`	or a block of statements.
	`do statement;` `while (test);`	or a block. statement is at least executed once.

`for i in range(100):`
`   statement`
	`int i;`
	`for (i=1; i <= 100; i++)`
	`  statement;`
		or a block.

	`switch (){` `  case 1: ...}`	Multiple choise.
`break;`	`break;`	Jump out of loop.
	`continue;`	Go directly to next round.

Variable declaration in C

A file of C-code consists of global variables and functions.

In Python a variable is created when you first use it. In C a variable must be declared before it can be used.

A variable is either global or local. A global variable is declared outside of functions. A local variable is a function parameter or declared at the beginning of a block inside a function.

The content of a block consists of some variable declarations and then some statements.

Functions are always global, they can't be declared inside a function.

```
int x;
x = 5;
foo (int y, float z) {
  int i;
  int x;
  ...
}
```

Global variable

Local variables

This local variable shadows the global variable. This means that inside the function `foo`, the local variable `x` is used not the global variable `x`.

Simple Data Types

datatype	**size**	**values**
char	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647
(long long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,087)
float	4	3.4E+/-38 (7 digits)
double	8	1.7E+/-308 (15 digits long)

Gotchas (1)

```
{
  int i;
  for(i = 0; i < 10; i++)
    …
```

NOT (as in Java)

```
{
  for(int i = 0; i < 10; i++)
    …
```

Gotchas (2)

- Uninitialized variables
  - catch with **-Wall** compiler option

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  int i;
  factorial(i);
  return 0;
}
```

This variable is declared but has not a given value when it is used here.

Gotchas (3)

- Error handling
  - No exceptions. No `try` but there is `longjump` which has similarities.
  - Must look at return values

Hello World

```
#include <stdio.h>
int main(int argc, char* argv[])
{
  /* print a greeting */
  printf("Hello World!\n");
  return 0;
}
```

The file helloworld.c

```
$> gcc -o helloworld helloworld.c   # Compile the file

$> ./helloworld                     # Run the program
Hello World!
$>
```


Edit, Compile, Run

C is a compiled language. Program in 3 steps:

- Edit file. Creates `helloworld.c`
- Compile. `helloworld.c` -> `helloworld`
- Run program `helloworld`

An interpreted language like Python is used in 2 steps: Edit and Run

Breaking down the code

- **#include <stdio.h>**
  - Include the contents of the file stdio.h
    - Case sensitive – lower case only
  - No semicolon at the end of line
- **int main(…)**
  - The OS calls this function when the program starts running.
- **printf(format_string, arg1, …)**
  - Prints out a string, specified by the format string and the arguments.

format_string

- Composed of ordinary characters (not %)
  - Copied unchanged into the output
- Conversion specifications (start with %)
  - Fetches one or more arguments
  - For example
    - **char** **%c**
    - **char*** **%s**
    - **int** **%d**
    - **float** **%f**
- For more details: **man -s 3 printf**

Compilation steps

Compilation is done in several steps:	Command	Converts
• Preprocessor	`gcc -E prog.c > progE.c`	prog.c -> progE.c
• Actual compilation	`gcc -x cpp-output -S progE.c`	progE.c -> prog.s
• Assembling	`gcc -c prog.s`	prog.s -> prog.o
• Linking	`gcc -o prog prog.o`	prog.o -> prog

C Preprocessor

```
#define FIFTEEN_TWO_THIRTEEN \
  "The Class That Gives CMU Its Zip\n"

int main(int argc, char* argv[])
{
  printf(FIFTEEN_TWO_THIRTEEN);
  return 0;
}
```

After the preprocessor (gcc –E)

```
int main(int argc, char* argv)
{
  printf("The Class That Gives CMU Its Zip\n");
  return 0;
}
```


Conditional Compilation

```
#define CS213

int main(int argc, char* argv)
{
#ifdef CS213
  printf("The Class That Gives CMU Its Zip\n");
#else
  printf("Some other class\n");
#endif /* CS213 */
  return 0;
}
```


After the preprocessor (gcc –E)

```
int main(int argc, char* argv)
{
  printf("The Class That Gives CMU Its Zip\n");
  return 0;
}
```


Command Line Arguments (1)

- **int main(*int argc, char* argv[]*)**
- **argc**
  - Number of arguments (including program name)
- **argv**
  - Array of char* (that is, an array of ‘c’ strings)
  - **argv[0]**: = program name
  - **argv[1]**: = first argument
  - **…**
  - **argv[argc-1]**: last argument

Command Line Arguments (2)

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  int i;
  printf("%d arguments\n", argc);
  for(i = 0; i < argc; i++)
    printf("  %d: %s\n", i, argv[i]);
  return 0;
}
```


Command Line Arguments (3)

```
$ ./cmdline The Class That Gives CMU Its Zip
8 arguments
  0: ./cmdline
  1: The
  2: Class
  3: That
  4: Gives
  5: CMU
  6: Its
  7: Zip
$
```


Arrays

- **char foo[80];**
  - An array of 80 characters
  - **sizeof(foo)**
    = 80 x **sizeof(char)**
    = 80 x 1 = 80 bytes
- **int bar[40];**
  - An array of 40 integers
  - **sizeof(bar)**
    = 40 x **sizeof(int)**
    = 40 x 4 = 160 bytes

Aggregate data: structures

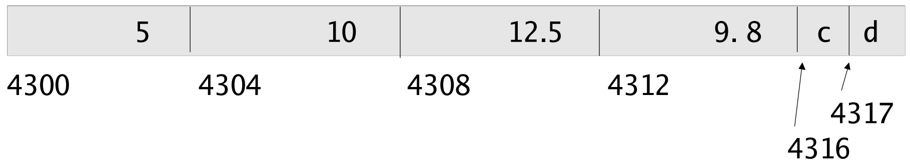
```
#include <stdio.h>

struct person
{
  char*     name;
  int       age;
}; /* <== DO NOT FORGET the semicolon */


int main(int argc, char* argv[])
{
  struct person bovik;
  bovik.name = "Harry Bovik";
  bovik.age = 25;

  printf("%s is %d years old\n", bovik.name,
  bovik.age);
  return 0;
}
```

Pointers

- Pointer variables are variables that hold an address in memory.
- That address can be the address of another variable.

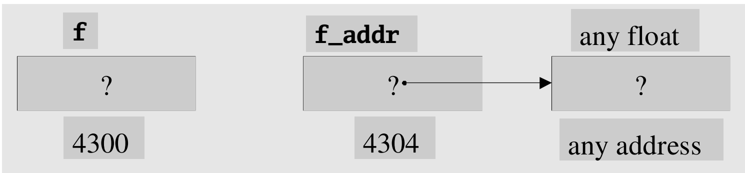
Memory layout and addresses

```
int x = 5, y = 10;
float f = 12.5, g = 9.8;
char c = 'c', d = 'd';
```

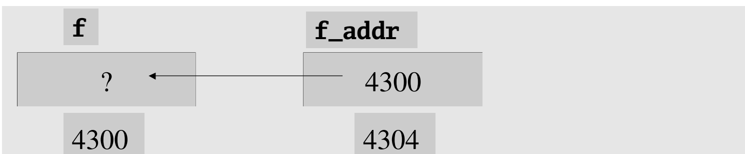


Using Pointers (1)

```
float f;        /* data variable */
float *f_addr;  /* pointer variable */
```

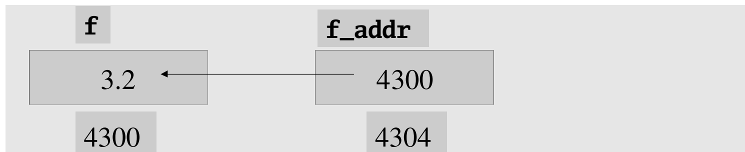



```
f_addr = &f;    /* & = address operator */
```

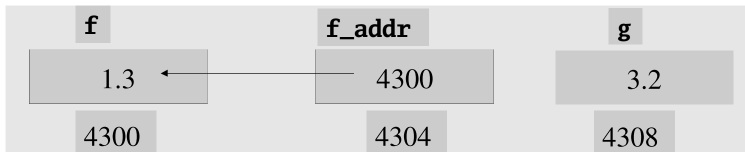


Pointers made easy (2)

```
*f_addr = 3.2;    /* * = indirection operator */
```




```
float g = *f_addr;/* indirection: g is now 3.2 */
f = 1.3;          /* but g is still 3.2 */
```



Function Parameters

- Function arguments are passed "by value".
- What is "pass by value"?
  - The called function is given a copy of the arguments.
- What does this imply?
  - The called function can't alter a variable in the caller function, only its private copy.
- Three examples

Example 1: swap_1

```
int x = 3;
int y = 4;
void swap_1(int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Q: After swap_1(x,y);
x =? y=?

A1: x=4; y=3;

A2: x=3; y=4;

Example 2: swap_2

```
int x = 3;
int y = 4;
void swap_2(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Q: After swap_2(&x,&y);
x =? y=?

A1: x=4; y=3;

A2: x=3; y=4;

Example 3: scanf

```
#include <stdio.h>

int main()
{
  int  x;
  scanf("%d\n", &x);
  printf("%d\n", x);
}
```

Q: Why using pointers in scanf?

A: We need to assign the value to x.

Dynamic Memory

- Pyton and Java manages memory for you, C does not
  - C requires the programmer to *explicitly* allocate and deallocate memory
  - Unknown amounts of memory can be allocated dynamically during run-time with **malloc()** and deallocated using **free()**

Not like Java or Python

- No **new**
- No garbage collection
- You ask for *n* bytes
  - Not a high-level request such as "I'd like an instance of class **String**"

malloc

- Allocates memory in the heap
  - Lives between function invocations
- Example
  - Allocate an integer
    - `int *iptr = (int *) malloc(sizeof(int));`
  - Allocate a structure
    - `struct name *nameptr = (struct name *) malloc(sizeof(struct name));`

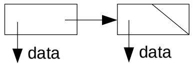
You have to do a cast as `malloc` always returns a `char *` pointer

free

- Deallocates memory in heap.
- Pass in a pointer that was returned by **malloc**.
- Example
  - `int *iptr =(int *) malloc(sizeof int));`
    `free(iptr);`
- Caveat: don't use freed memory and don't free the same memory block twice!
- Must remember to free allocated memory

Linked list

- In C a linked list of strings is constructed with structs in dynamic memory thus:




```
#include <stdio.h>

struct node
{
  char        *data;
  struct node *next;
} *list;

int main(int argc, char* argv[])
{
  list = NULL;
  struct node *p;
  for (i = 1; i < argc; i++)
  {
    p = (struct node*) malloc(sizeof(struct node));
    p -> next = list;    p -> data = argv[i];
    list = p;
  }
  while (list != NULL)
  {
    printf(" %s", list -> data);
    p = list;
    list = list -> next;
    free(p);
  }
  putchar('\n');
  return 0;
}
```