C Primer

http://csapp.cs.cmu.edu/public/CPrimer/CPrimer.ppt

(revised K. Louden 2/06)

(revised T. Howell 1/21/14)

Outline

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

Like Java, like C

- Operators same as Java:
  - Arithmetic
    - `i = i+1; i++; i--; i *= 2;`
    - `+, -, *, /, %,`
  - Relational, Logical, Conditional
    - `<, >, <=, >=, ==, !=`
    - `&&, ||, !, &, |, ~, ^`
    - `?:`
- Syntax same as in Java:
  - `if ( ) { } else { }`
  - `while ( ) { }`
  - `do { } while ( );`
  - `for(i=1; i <= 100; i++) { }`
  - `switch ( ) {case 1: … }`
  - `continue; break;`

Simple Data Types in C

datatype	**size***	**values** ***in bytes = 8 bits**
char	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long (or long int)	4	-2,147,483,648 to 2,147,483,647
long long	8	-9,223,372,036,854,775,808 to 9,...,807
float	4	3.4E+/-38 (7 digits)
double	8	1.7E+/-308 (15 digits long)
long double	10	??

Compare to Java:

datatype	**size**	**values**
boolean	1	true or false
byte	1	-128 to 127
char	2	0 to 65,535 (\u0000 to \uffff unicode)
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	8	-9,223,372,036,854,775,808 to 9,...,807
float	4	$(2-2^{-23})\cdot 2^{127}$ to 2^{-149}
double	8	$(2-2^{-52})\cdot 2^{1023}$ to 2^{-1074}

Simple Data Types in C, cont.

C also has *unsigned* versions of the integral types:

datatype	size	values
unsigned char	1	0 to 255
unsigned short	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
unsigned long long	8	0 to 18,446,744,073,709,551,615

Note no boolean!

Instead, C uses ints, 1 represents "true" and 0 represents "false": 0 == 1 has the value 0, and 0 == 0 has the value 1. In if, while, etc., the test must be convertible to an int: nonzero means "true" and zero means "false": if ('a') … is ok in C (and always executes the true case).

Java programmer gotchas (1)

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

NOT

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

[But this is allowed in the C99 standard – see later slide]

Java programmer gotchas (2)

- Uninitialized variables
  - catch with `-Wall` compiler option (maybe)

```
#include <stdio.h>

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


Java programmer gotchas (3)

- Functions with no parameters must use `void` in declaration in C (except for `main`!):

```
int f(void) // <- without the void here
{ return 1;}

int main()
{ f(10); // <- no compilation error here
  return 0;
}
```

Java programmer gotchas (4)

- Error handling
  - No exceptions
  - Must look at return values

"Hello, CS 47 class!"

```
#include <stdio.h>
int main(int argc, char* argv[])
{
  /* print a greeting */
  printf("Hello, CS 47 class!\n");
  return 0;
}
```

```
C:> hello
Hello, CS 47 class!
C:>
```

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, %x, %i`
    - `float` `%f`
- For more details (Cygwin): `man printf`
- Now Java has it: `PrintStream.printf`

C Preprocessor

```
#define HELLO_CLASS \
  "Hello, CS 47 class!\n"

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


After the preprocessor (gcc -E)

```
int main(int argc, char* argv[])
{
  printf("Hello, CS 47 class!\n");
  return 0;
}
```

Conditional Compilation

```
#define CS47

int main(int argc, char* argv[])
{
  #ifdef CS47
  printf("Introduction to Computer Systems\n");
  #else
  printf("Some other class\n");
  #endif
  return 0;
}
```


After the preprocessor (gcc -E)

```
int main(int argc, char* argv[])
{
  printf("Introduction to Computer Systems\n");

  return 0;
}
```

Command Line Arguments (1)

- `int main(int argc, char* argv[])`
- **`argc`**
  - Number of arguments (including program name)
- **`argv`**
  - Array of char*s (that is, an array of 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)

```
C:> cmdline Introduction to Computer Systems
5 arguments
  0: c:/cs47prg/cmdline.exe
  1: Introduction
  2: to
  3: Computer
  4: Systems
```


Arrays

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

Arrays (cont.)

- Array variables must always be declared with a size: `int a[];` is an error.
- Array parameters can be declared without a size, since the arrays are allocated elsewhere: `void sort(int size, int a[])` is fine.
- Arrays are really pointers, and if you want a variable-length array you declare it as a pointer (see later slides on pointers):

```
int* a =
   (int*) malloc(sizeof(int)*len);
```

- Now `a` can be used just like an array of `len` `int`s: `a[len-1] = 42;` etc….

Strings

- String in C are variable-length arrays of characters, thus are declared as `char*`
- Double-quoted strings are allocated automatically by the system:

```
char* hello = "Hello";
```

- There are actually six characters in `hello`: the last is always a null character `'\0'`
- That is how programs can find the end, since the size of a string cannot otherwise be determined:

```
int i = 0;
while(hello[i] != '\0')
   printf("%c\n",hello[i++]);
```

Structures

- Aggregate data

```
#include <stdio.h>

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

int main(int argc, char* argv[])
{ struct person potter;
  potter.name = "Harry Potter";
  potter.age = 17;

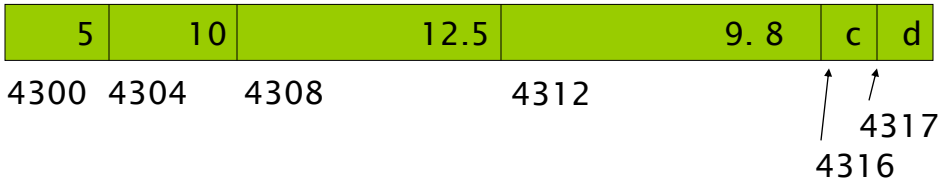
  printf("%s is %d years old\n", potter.name,
  potter.age);
  return 0;
}
```


Pointers

- Pointers are variables that hold an address in memory.
- That address contains another variable.
- Similar to Java object references
- Actual addresses replace the (invisible) Java references
- Use 0 for null (sometimes aliased in C programs as NULL).

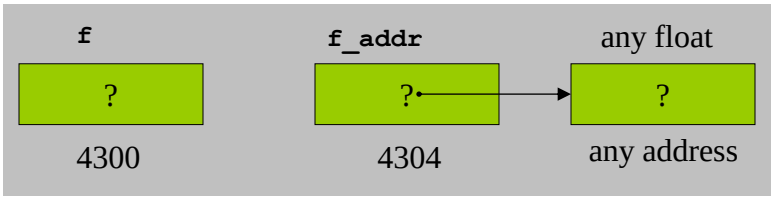
Memory layout and addresses

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

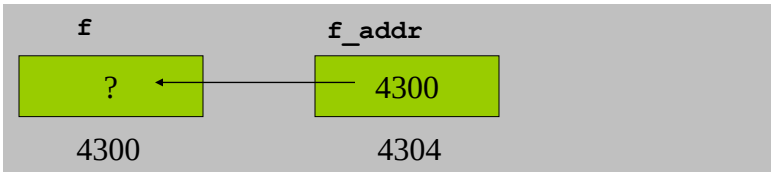


Using Pointers (1)

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

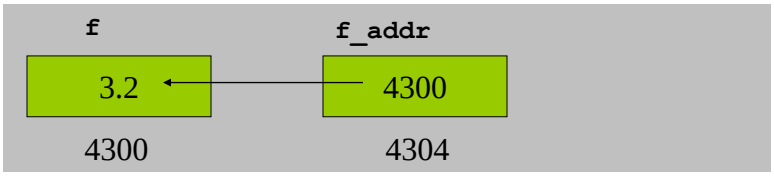



```
f_addr = &f;    /* & = address operator */
```

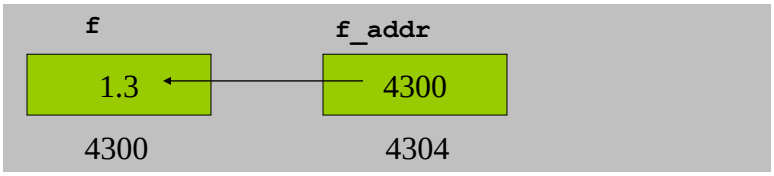


Pointers made easy (2)

```
*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 */
```



27

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, but its private copy.
- Three examples

Example 1: swap_1

```
void swap_1(int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Q: Let x=3, y=4, after swap_1(x,y); x =? y=?

~~A1: x=4; y=3;~~

A2: x=3; y=4;

Example 2: swap_2

```
void swap_2(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Q: Let x=3, y=4, after swap_2(&x,&y); x =? y=?

~~A1: x=3; y=4;~~

A2: x=4; y=3;

Example 3: scanf

```
#include <stdio.h>

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

Q: Why using pointers in scanf?

A: We need to assign the value to x.

Dynamic Memory

- 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

- 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));
      ```

free

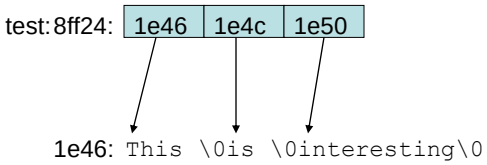
- Deallocates memory in heap.
- Pass in a pointer that was returned by **`malloc`**.
- Example
  - ```
    int *iptr =
      (int *) malloc(sizeof(int));
    free(iptr);
    ```
- Caveat: don't free the same memory block twice!

C99 – New standard (old was C89)

- Adds many useful C++ features (but not classes or booleans!)
- // … comments
- for (int i =… [noted previously]
- Declarations anywhere in a block:
  ```
  { printf("entering block\n");
    int i; // illegal in C89
  ```
- Use -std=c99 option to gcc.
  ```

Question from class

- What space is allocated for array of strings?
- char *test[3] = {"This ","is ","interesting"};



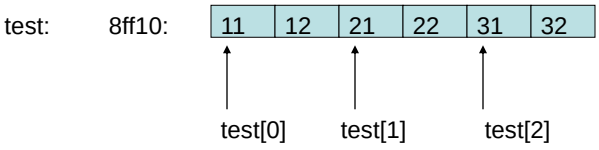
sizeof(test) = 12

sizeof (test[i]) = 4

sizeof(*test[i]) = 1, size of string is determined by \0

Question from class (2)

- What space is allocated for 2-dim int arrays?
- int test[3][2] = {{11, 12}, {21, 22}, {31, 32}};



sizeof(test) = 24

sizeof(test[i]) = 8

sizeof(*test[i]) = 4