CS 201
Computer Systems Programming

Prof. Harry Porter

www.cs.pdx.edu/~harry/cs201

Agenda for Today

- Attendance
- Course objectives
- Syllabus
  - Textbooks, policies, class mailing list, etc.
- Assignment 1
- Review of the C programming language

About the course

How does “system software” work?

- **How is a “C” program actually executed?**
- **How is system software organized?**
- **Compiling, Assembly Language, OS organization**

The “C” Programming Language

- **Skills and knowledge of “C” programming**

Syllabus

Course home page

- cs.pdx.edu/~harry/cs201
- Syllabus with updated course schedule
- Information about instructor, TA, office hours
- Email mailing list (mailman): PorterClassList

Syllabus

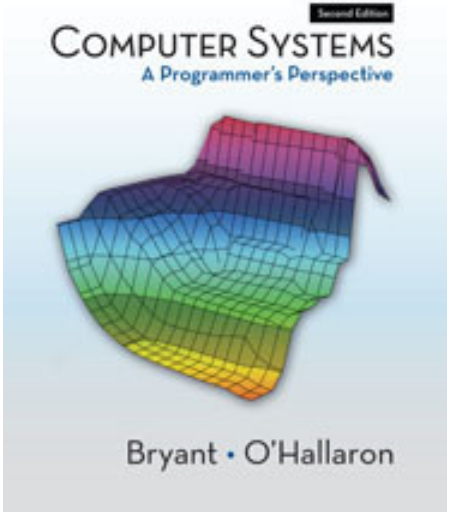
Accounts

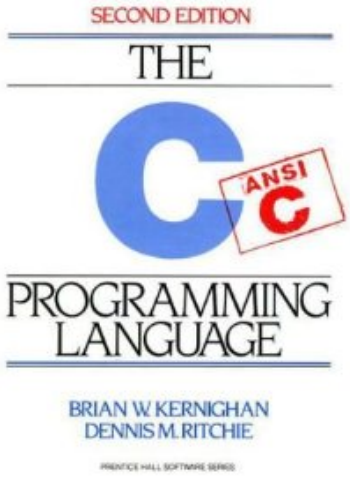
- **Instructions on course web page**
  - **Activate your account in person at CAT front desk**
  - **linuxlab.cs.pdx.edu**
    - » Linux systems in FAB 88-09 and FAB 88-10
    - » Where homework assignments will be run
- **Login remotely or in person (Basement of EB)**
  - **ssh user@linuxlab.cs.pdx.edu**
  - **ssh comes with the putty package**
    - » http://www.chiark.greenend.org.uk/~sgtatham/putty
  - **ssh comes with cygwin**
    - » http://www.cygwin.com
    - » Alternatively, you can develop your code via cygwin
    - » Code will be graded on linuxlab so make sure it works on linuxlab systems

Syllabus

Textbooks

- Randal E. Bryant and David R. O'Hallaron,
  - "Computer Systems: A Programmer's Perspective", Prentice Hall 2003.
  - csapp.cs.cmu.edu
  - All slide materials in this class are based on material provided by Bryant and O'Hallaron
- Brian Kernighan and Dennis Ritchie,
  - "The C Programming Language, Second Edition", Prentice Hall, 1988





Syllabus

Getting help

- CS Tutors
- TA and instructor office hours
- On-line resources for gdb, make, etc.

Policies

- You are responsible for everything that takes place in class
- Reading assignments will be posted with each lecture
- Homework assignments due at start of class on due date
  - Follow submission instructions on home page carefully, especially for programming assignments.
  - Late policy: 50% off, if not submitted before class time.

Syllabus

Academic integrity

- Automatic failing grade given
- Departmental guidelines available in CS office

What is not cheating?

- Discussing the design for a program is OK.
- Helping each other orally (not in writing) is OK.
- Using anything out of the textbook or my slides is OK.
- Copying code "snippets", templates for system calls, or declarations from a reference book or header files are OK

What is cheating?

- Copying code verbatim without attribution
  - Source-code plagiarism tools
- Copying someone's answer or letting someone copy your answer.
- Mailing code to the class mailing list.

Syllabus

Grading

- Reading assignments throughout the course not graded
  - Significant exam material drawn from practice problems in the textbook
- Homework:
  - Programming assignments in C – email to grader
  - Written homework – hand in hardcopy
- Two exams
  - Midterm exam
  - Final comprehensive exam (covering entire term)

Syllabus

Supporting Video Material

- Accessible through website
- To augment lectures
  - 100% Attendance is required
- “Binary Numbers”
- “Assembly Language and Processor Architecture”
- “Echo-360”
  - Video capture of classroom
  - DO NOT DEPEND ON THIS
  - DO NOT SKIP CLASSES

More Environment

Programming style – K&R C

Makefiles are required for all programming assignments

- See the course web page for some examples

The Grader will run and read your programs

- Poorly written code, improperly formatted code, and an absence of comments will prevent you from getting full credit

Homework Assignment 1

The specification is on course web site

- cs.pdx.edu/~harry/cs201
- HW 1: Small C program
  - Will use several system calls

    rand, gettimeofday, printf, scanf, strlen, etc…
- Makefile required
- Grader will run and read your program
  - Poorly written code, improperly formatted code, and an absence of comments will prevent you from getting full credit
- Due in two weeks

If you are unable to complete this program on time, you should consider dropping the course.

Introduction to C Programming

Why C?

Used prevalently

- Operating systems (e.g. Windows, Linux, FreeBSD/OS X)
- Web servers (apache)
- Web browsers (firefox)
- Mail servers (sendmail, postfix, uw-imap)
- DNS servers (bind)
- Video games (any FPS)
- Graphics card programming

  (OpenCL GPGPU programming based on C)

Why?

- Performance
- Portability
- Wealth of programmers

Why C?

Compared to other high-level languages

- Maps almost directly into hardware instructions making code potentially more efficient
  - Provides minimal set of abstractions compared to other HLLs
  - HLLs make programming simpler at the expense of efficiency

Compared to assembly programming

- Abstracts out hardware (i.e. registers, memory addresses) to make code portable and easier to write
- Provides variables, functions, arrays, complex arithmetic and boolean expressions

Why assembly?

Learn how programs map onto underlying hardware

- Allows programmers to write efficient code

Perform platform-specific tasks

- Access and manipulate hardware-specific registers
- Interface with hardware devices
- Utilize latest CPU instructions

Reverse-engineer unknown binary code

- Analyze security problems caused by CPU architecture
- Identify what viruses, spyware, rootkits, and other malware are doing
- Understand how cheating in on-line games work

The C Programming Language

One of many programming languages

Imperative (procedural) programming language

- Computation consisting of statements that change program state
- Language makes explicit references to state (i.e. variables)
- Computation broken into modular components ("procedures" or "functions") that can be called from any point

Declarative programming languages

- Describes what something is like, rather than how to create it
- Implementation left to other components
- Example: HTML, SQL

The C Programming Language

Simpler than C++, C#, Java

- No support for
  - Objects
  - Memory management
  - Array bounds checking
  - Non-scalar operations
- Simple support for
  - Typing
  - Structures
- Basic utility functions supplied by libraries
  - libc, libpthread, libm
- Low-level, direct access to machine memory (pointers)
- Easier to write bugs, harder to write programs, typically faster
  - Looks better on a resume

C based on updates to ANSI-C standard

- Current version: C99 → C11

The C Programming Language

Compilation down to machine code as in C++

- Compiled, assembled, linked via gcc

Compared to interpreted languages…

- Perl
  - Commands interpreted by perl interpreter software
  - Interpreter runs natively
- Java
  - Compilation to virtual machine “byte code”
  - Byte code interpreted by virtual machine software
  - Virtual machine runs natively
  - Exception: “Just-In-Time” (JIT) compilation to machine code

Our environment

All programs must run on the CS Linux Lab machines

- ssh *user*@linuxlab.cs.pdx.edu

Architecture will be 32-bit x86 (IA32)

GNU gcc compiler

- gcc -m32 –o hello hello.c
- (-m32 = compiles to 32-bit)

GNU gdb debugger

- ddd is a graphical front end to gdb
- “gdb -tui” is a graphical curses interface to gdb
- Must use “-g” flag when compiling and remove –O flags
  - gcc –g hello.c
  - Add debug symbols and do not reorder instructions for performance

Variables

Named using letters, numbers, some special characters

- By convention, not all capitals

Must be declared before use

- Contrast to typical scripting languages (Perl, Python, PHP, JavaScript)
- C is statically typed (for the most part)

Data Types and Sizes

char – single byte integer

- 8-bit character, hence the name
- Strings implemented as arrays of char and referenced via a pointer to the first char of the array

short – short integer

- 16-bit (2 bytes) on IA32, not used much

int – integer, size varies by architecture

- 32-bit (4 bytes) on IA32
- Qualifiers: ‘unsigned’, ‘short’, ‘long’

float – single precision floating point

- 32-bit (4 bytes) on IA32

double – double precision floating point

- 64 bit (8 bytes) on IA32

Constants

Integer literals

1234

0xFE, 0xab78

Character constants

'a' – numeric value of character 'a'

char letterA = 'a';

int asciiA = 'a';

What's the difference?

String Literals

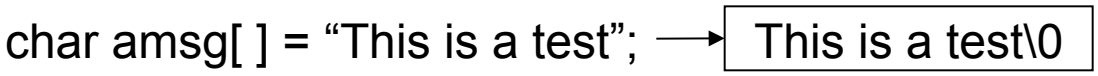
"I am a string"

"" // empty string

Constant pointers

Used for static arrays

- **Symbol that points to a fixed location in memory**



- **Can change change characters in string (amsg[3] = 'x';)**
- **Can not reassign amsg to point elsewhere (i.e. amsg = p)**

Declarations and Operators

Variable declaration

```
int foo;
char *ptr;
float ff;
```

Can include initialization

```
int foo = 34;
char *ptr = “fubar”;
float ff = 34.99;
```

Arithmetic operators

- +, - , *, /, %
- Modulus operator (%)
- Arithmetic operators associate left to right

Expressions

In C, oddly, assignment is an expression

- "x = 4" has the value 4

```
if (x == 4)
    y = 3;      /* sets y to 3 if x is 4 */

if (x = 4)
    y = 3;      /* always sets y to 3 */

while ((c=getchar()) != EOF) …
```

Expressions

But on Nov. 5, 2003, Larry McVoy **noticed** that there was a code change in the CVS copy that did not have a pointer to a record of approval. Investigation showed that the change had never been approved and, stranger yet, that this change did not appear in the primary BitKeeper repository at all. Further investigation determined that someone had apparently broken in (electronically) to the CVS server and inserted this change.

What did the change do? This is where it gets really interesting. The change modified the code of a Linux function called wait4, which a program could use to wait for something to happen. Specifically, it added these two lines of code:

```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
        retval = -EINVAL;
```

https://freedom-to-tinker.com/blog/felten/the-linux-backdoor-attempt-of-2003/

Increment and Decrement

Comes in prefix and postfix flavors

- i++, ++i
- i--, --i

Makes a difference in evaluating complex statements

- A major source of bugs
- Prefix: increment happens before evaluation
- Postfix: increment happens after evaluation

When the actual increment/decrement occurs is important to know about

- Is “i++*2” the same as “++i*2” ?

Comparison to Java

Operators same as Java:

- **Arithmetic**
  - `i = i+1; i++; i--; i *= 2;`
  - `+, -, *, /, %,`
- **Relational and Logical**
  - `<, >, <=, >=, ==, !=`
  - `&&, ||, &, |, !`

Control flow syntax same as in Java:

- `if ( ) { } else { }`
- `while ( ) { }`
- `do { } while ( );`
- `for(i=1; i <= 100; i++) { }`
- `switch ( ) {case 1: … }`
- `continue; break;`

Simple data types are the same

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**
float	**4**	**3.4E+/-38 (7 digits)**
double	**8**	**1.7E+/-308 (15 digits long)**

Exact details of size are "implementation dependent"!

Java programmer gotchas (1)

Must declare variables ahead of time in C

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

NOT

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

Java programmer gotchas (2)

Uninitialized variables in C allowed

- **catch with `-Wall` compiler option**

```
#include <stdio.h>

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

Java programmer gotchas (3)

Error handling

- No “throw/catch” exceptions for functions in C
- Must look at return values or install global signal handlers (see B&O Chapter 8)

Java programmer gotchas (4)

Dynamic memory

- Managed languages such as Java perform memory management (i.e., garbage collection) for programmers
- C requires the programmer to *explicitly* allocate and deallocate memory
- No "new" for a high-level object
- Memory can be allocated dynamically during run-time with `malloc()` and deallocated using `free()`
- Must supply the size of memory you want explicitly

“Good evening”

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

```
$ ./goodevening
Good evening!
$
```

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, …)
```

- Call function from libc library
- Prints out a string, specified by the format string and the arguments.

Command Line Arguments (1)

main has two arguments from the command line

```
int main(int argc, char* argv[])
```

`argc`

- Number of arguments (including program name)

`argv`

- Pointer to an array of string pointers
  - `argv[0]`: = program name
  - `argv[1]`: = first argument
  - `argv[argc-1]`: last argument

- Example: find . –print
  - argc = 3
  - argv[0] = “find”
  - argv[1] = “.”
  - argv[2] = “-print”

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

Arrays

```
char foo[80];
```

- An array of 80 characters (stored contiguously in memory)

  `sizeof(foo)`
  = 80 × `sizeof(char)`
  = 80 × 1 = 80 bytes

```
int bar[40];
```

- An array of 40 integers (stored contiguously in memory)

  `sizeof(bar)`
  = 40 × `sizeof(int)`
  = 40 × 4 = 160 bytes

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 prof;
  prof.name = "Harry Porter";
  prof.age = 58;

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

Pointers

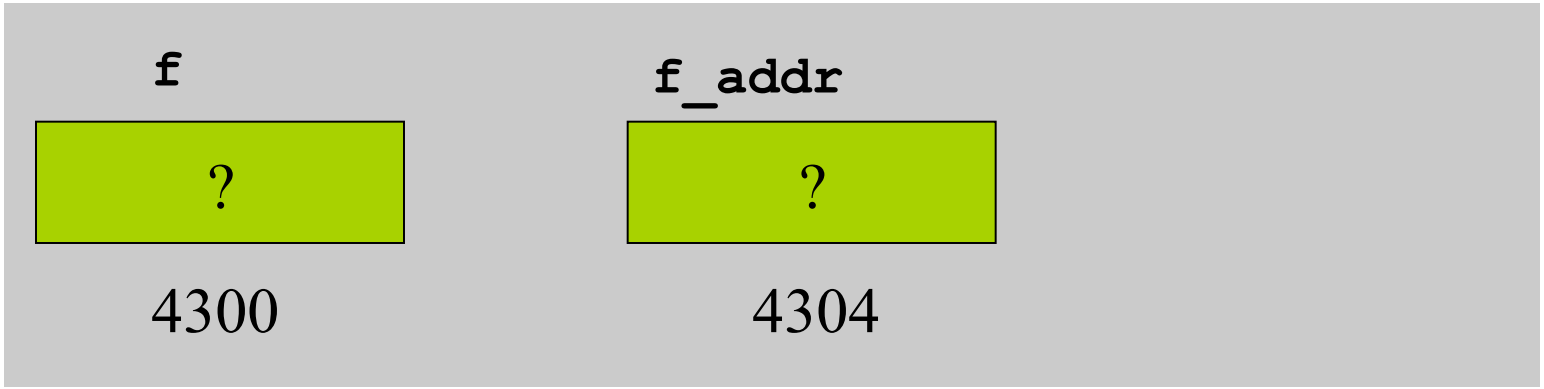
Pointers are variables that hold an address in memory.

That address contains another variable.

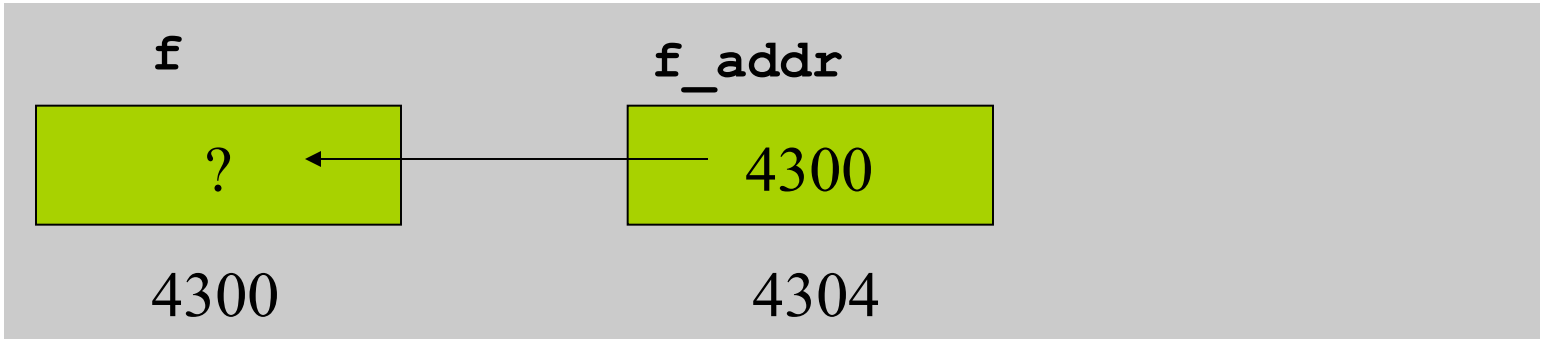
Unique to C

Using Pointers (1)

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

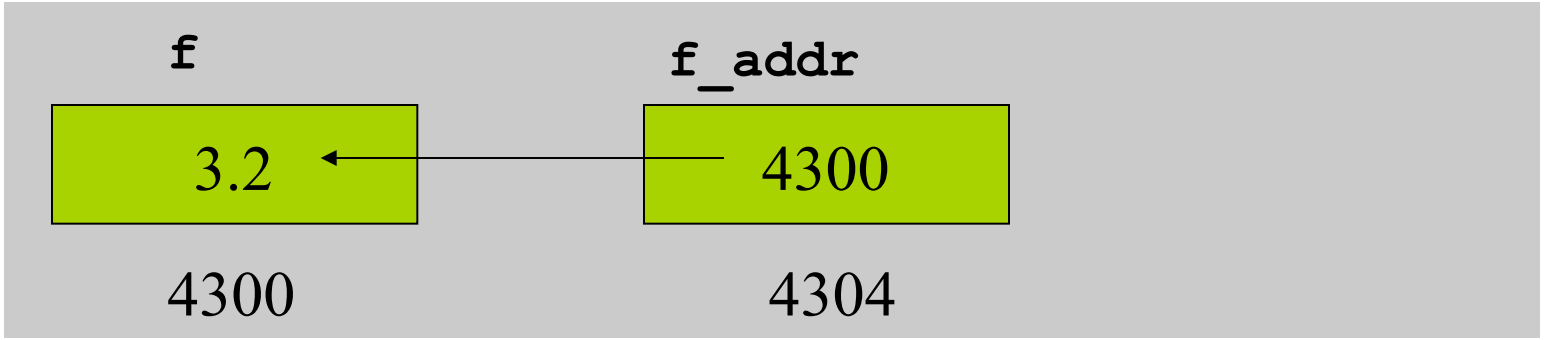



```
f_addr = &f;      /* & = address operator */
```

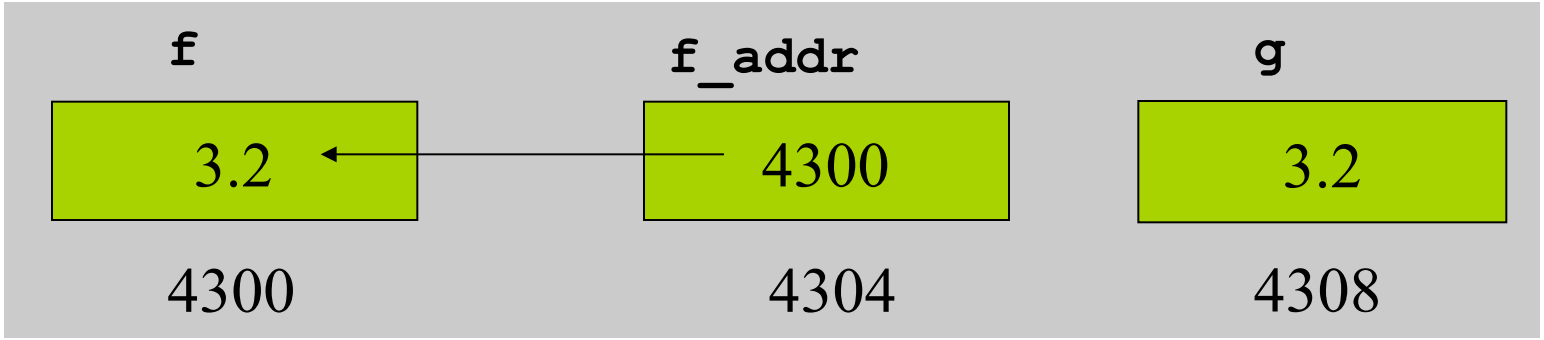


Using Pointers (2)

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

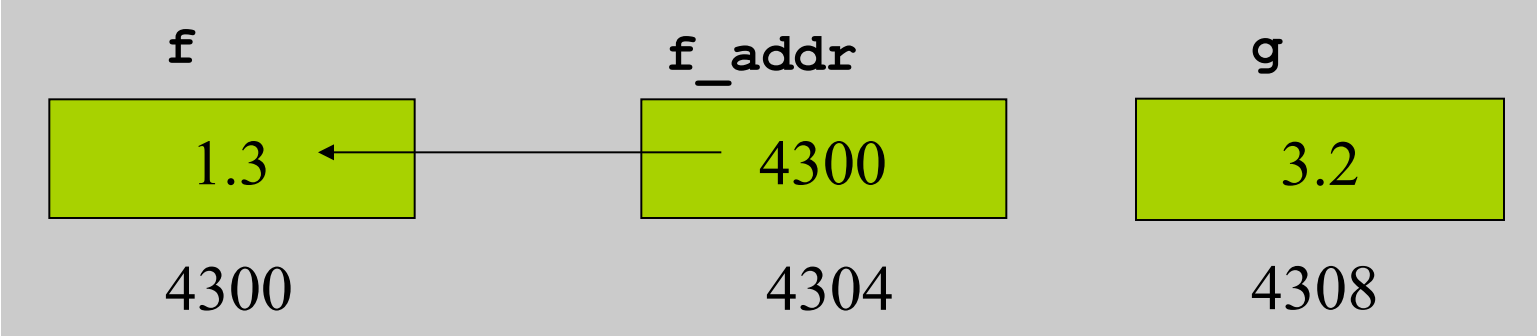



```
float g = *f_addr;/* indirection: g is now 3.2 */
```



Using Pointers (3)

```
f = 1.3;                    /* but g is still 3.2 */
```



Function calls (static)

Calls to functions normally resolved statically

(“Static” means done at compile-time.)

```
void print_ints(int a, int b)   {
  printf(“%d %d\n”,a,b);
}

int main(int argc, char* argv[]) {
   int i=3;
   int j=4;
   print_ints(i,j);
}
```

Function call 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.**

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 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 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;

Call by value vs. reference in C

Call by reference implemented via pointer passing

```
void swap(int* px, int* py) {
  int tmp;
  tmp = *px;
  *px = *py;
  *py = tmp;
}
```

- Swaps the values of the variables x and y if px is &x and py is &y
- Uses integer pointers instead of integers

Otherwise, call by value...

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

Function calls (dynamic)

Using function pointers, C can support late-binding of functions where calls are determined at run-time

```
#include <stdio.h>
void print_even(int i){ printf("Even %d\n",i);}
void print_odd(int i) { printf("Odd %d\n",i); }

int main(int argc, char **argv) {
     void (*fp)(int);
     int i = argc;

     if !(argc%2)
          fp=print_even;
     else
          fp=print_odd;
     fp(i);
}

% ./funcp a
Even 2
% ./funcp a b
Odd 3
```