上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

Course organization

- Course introduction (Week 1)
  - Code editor: Emacs
- Part I: Introduction to C programming language (Week 2 - 9)
  - **Chapter 1: Overall Introduction (Week 1-3)**
  - Chapter 2: Types, operators and expressions (Week 4)
  - Chapter 3: Control flow (Week 5)
  - Chapter 4: Functions and program structure (Week 6)
  - Chapter 5: Pointers and arrays (Week 7)
  - Chapter 6: Structures (Week 8)
  - Chapter 7: I/O (Week 9)
- Part II: Skills others than programming languages (Week 10- 11)
  - Debugging tools（Week 10）
  - Keeping projects documented and manageable （Week 11）
  - Source code managing （Week 11）
- Part III: Reports from the battle field (student forum) (week 12 – 16)

Brief Introduction to the C Programming Language

Chaochun Wei
Shanghai Jiao Tong University
Spring 2013

Introduction

- The C programming language was designed by Dennis Ritchie at Bell Laboratories in the early 1970s
- Influenced by
  - ALGOL 60 (1960),
  - CPL (Cambridge, 1963),
  - BCPL (Martin Richard, 1967),
  - B (Ken Thompson, 1970)
- Traditionally used for systems programming, though this may be changing in favor of C++
- Traditional C:
  - *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, 2nd Edition, Prentice Hall
  - Referred to as *K&R*

Standard C

- Standardized in 1989 by ANSI (American National Standards Institute) known as ANSI C
- International standard (ISO) in 1990 which was adopted by ANSI and is known as *C89*
- As part of the normal evolution process the standard was updated in 1995 (*C95*) and 1999 (*C99*)
- C++ and C
  - C++ extends C to include support for Object Oriented Programming and other features that facilitate large software development projects
  - C is not strictly a subset of C++, but it is possible to write "*Clean C*" that conforms to both the C++ and C standards.

- Feb, 26

Course organization

- Course introduction (Week 1)
  - Code editor: Emacs
- Part I: Introduction to C programming language (Week 2 - 9)
  - **Chapter 1: Overall Introduction (Week 1-3)**
  - Chapter 2: Types, operators and expressions (Week 4)
  - Chapter 3: Control flow (Week 5)
  - Chapter 4: Functions and program structure (Week 6)
  - Chapter 5: Pointers and arrays (Week 7)
  - Chapter 6: Structures (Week 8)
  - Chapter 7: I/O (Week 9)
- Part II: Skills others than programming languages (Week 10- 11)
  - Debugging tools（Week 10）
  - Keeping projects documented and manageable （Week 11）
  - Source code managing （Week 11）
- Part III: Reports from the battle field (student forum) (week 12 – 16)

Elements of a C Program

- A C development environment includes
  - *System libraries* and *headers*: a set of standard libraries and their header files. For example see `/usr/include` and `glibc`.
  - *Application Source*: application source and header files
  - *Compiler*: converts source to object code for a specific platform
  - *Linker*: resolves external references and produces the executable module
- User program structure
  - there must be one main function where execution begins when the program is run. This function is called main
    - `int main (void) { ... },`
    - `int main (int argc, char *argv[]) { ... }`
    - UNIX Systems have a 3rd way to define main(), though it is not POSIX.1 compliant
      `int main (int argc, char *argv[], char *envp[])`
  - additional local and external functions and variables

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

A Simple C Program

- *Create* example file: `try.c`
- *Compile* using gcc:
  `gcc -o try try.c`
- The standard C library *libc* is included automatically
- *Execute* program
  `./try`
- Note, I always specify an absolute path
- Normal termination:
  `void exit(int status);`
  - calls functions registered with `atexit()`
  - flush output streams
  - close all open streams
  - return status value and control to host environment

```
/* you generally want to
 * include stdio.h and
 * stdlib.h
 * */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
   printf("Hello World\n");
   exit(0);
}
```

Source and Header files

- Just as in C++, place related code within the same module (i.e. file).
- Header files (`*.h`) export interface definitions
  - function prototypes, data types, macros, inline functions and other common declarations
- Do not place source code (i.e. definitions) in the header file with a few exceptions.
  - inline'd code
  - class definitions
  - const definitions
- *C preprocessor* (`cpp`) is used to insert common definitions into source files
- There are other cool things you can do with the preprocessor

Another Example C Program

/usr/include/stdio.h

```
/* comments */
#ifndef _STDIO_H
#define _STDIO_H

... definitions and protoypes

#endif
```

/usr/include/stdlib.h

```
/* prevents including file
 * contents multiple
 * times */
#ifndef _STDLIB_H
#define _STDLIB_H

... definitions and protoypes

#endif
```

`#include` directs the preprocessor to "include" the contents of the file at this point in the source file. `#define` directs preprocessor to define macros.

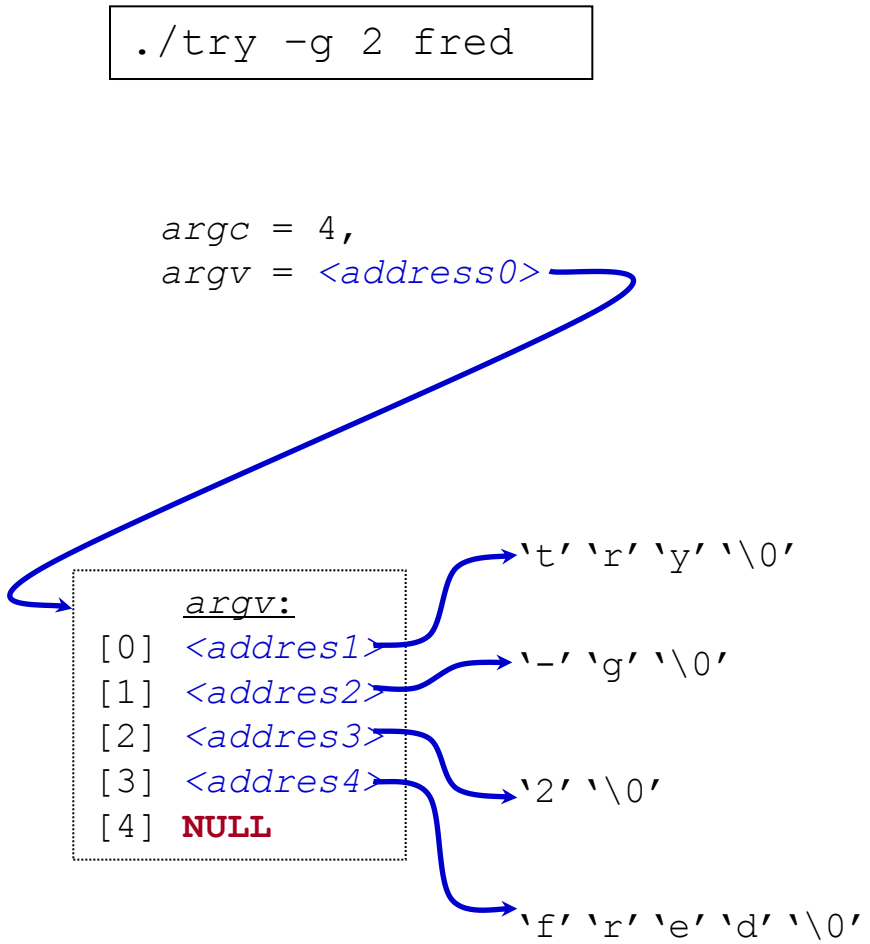
example.c

```
/* this is a C-style comment
 * You generally want to palce
 * all file includes at start of file
 * */
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char **argv)
{
  // this is a C++-style comment
  // printf prototype in stdio.h
  printf("Hello, Prog name = %s\n",
                argv[0]);
  exit(0);
}
```

Passing Command Line Arguments

- When you execute a program you can include arguments on the command line.
- The run time environment will create an argument vector.
  - `argv` is the argument vector
  - `argc` is the number of arguments
- Argument vector is an array of pointers to strings.
- a *string* is an array of characters terminated by a binary 0 (NULL or '\0').
- *argv[0]* is always the program name, so *argc* is at least 1.



C Standard Header Files you may want to use

- Standard Headers you should know about:
  - `stdio.h` – file and console (also a file) IO: *perror, printf, open, close, read, write, scanf*, etc.
  - `stdlib.h` - common utility functions: *malloc, calloc, strtol, atoi,* etc
  - `string.h` - string and byte manipulation: *strlen, strcpy, strcat, memcpy, memset*, etc.
  - `ctype.h` – character types: *isalnum, isprint, isupport, tolower,* etc.
  - `errno.h` – defines *errno* used for reporting system errors
  - `math.h` – math functions: *ceil, exp, floor, sqrt*, etc.
  - `signal.h` – signal handling facility: *raise, signal*, etc
  - `stdint.h` – standard integer: *intN_t, uintN_t*, etc
  - `time.h` – time related facility: *asctime, clock, time_t,* etc.

The Preprocessor

- The C preprocessor permits you to define simple macros that are evaluated and expanded prior to compilation.

- Commands begin with a '#'. Abbreviated list:
  - `#define` : defines a macro
  - `#undef` : removes a macro definition
  - `#include` : insert text from file
  - `#if` : conditional based on value of expression
  - `#ifdef` : conditional based on whether macro defined
  - `#ifndef` : conditional based on whether macro is not defined
  - `#else` : alternative
  - `#elif` : conditional alternative
  - `defined()` : preprocessor function: 1 if name defined, else 0

```
#if defined(__NetBSD__)
```

Preprocessor: Macros

- Using macros as functions, exercise caution:
  - flawed example: `#define mymult(a,b) a*b`
    - Source: `k = mymult(i-1, j+5);`
    - Post preprocessing: `k = i - 1 * j + 5;`
  - better: `#define mymult(a,b) (a)*(b)`
    - Source: `k = mymult(i-1, j+5);`
    - Post preprocessing: `k = (i - 1)*(j + 5);`
- Be careful of *side effects*, for example what if we did the following
  - Macro: `#define mysq(a) (a)*(a)`
  - flawed usage:
    - Source: `k = mysq(i++)`
    - Post preprocessing: `k = (i++)*(i++)`
- Alternative is to use inline'ed functions
  - `inline int mysq(int a) {return a*a};`
  - `mysq(i++)` works as expected in this case.

Preprocessor: Conditional Compilation

- Its generally better to use inline'ed functions
- Typically you will use the preprocessor to define constants, perform conditional code inclusion, include header files or to create shortcuts
- `#define DEFAULT_SAMPLES  100`
- `#ifdef __linux`

```
static inline int64_t
  gettime(void) {...}
```

- `#elif defined(sun)`

```
static inline int64_t
  gettime(void) {return (int64_t)gethrtime()}
```

- `#else`

```
static inline int64_t
  gettime(void) {... gettimeofday()...}
```

- `#endif`

- March, 7

Course organization

- Course introduction (Week 1)
  - Code editor: Emacs
- Part I: Introduction to C programming language (Week 2 - 9)
  - **Chapter 1: Overall Introduction (Week 1-3)**
  - Chapter 2: Types, operators and expressions (Week 4)
  - Chapter 3: Control flow (Week 5)
  - Chapter 4: Functions and program structure (Week 6)
  - Chapter 5: Pointers and arrays (Week 7)
  - Chapter 6: Structures (Week 8)
  - Chapter 7: I/O (Week 9)
- Part II: Skills others than programming languages (Week 10- 11)
  - Debugging tools（Week 10）
  - Keeping projects documented and manageable （Week 11）
  - Source code managing （Week 11）
- Part III: Reports from the battle field (student forum) (week 12 – 16)

Another Simple C Program

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

  return 0;
}
```

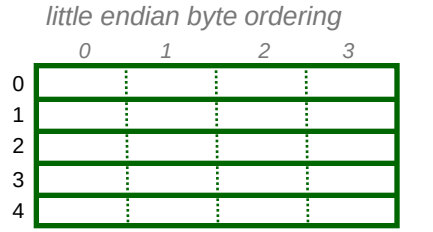
- Notice that the syntax is similar to Java
- What's new in the above simple program?
  - of course you will have to learn the new interfaces and utility functions defined by the C standard and UNIX
  - Pointers will give you the most trouble

Arrays and Pointers

- A variable declared as an array represents a contiguous region of memory in which the array elements are stored.

```
int x[5]; // an array of 5 4-byte ints.
```

- All arrays begin with an index of 0



memory layout for array x

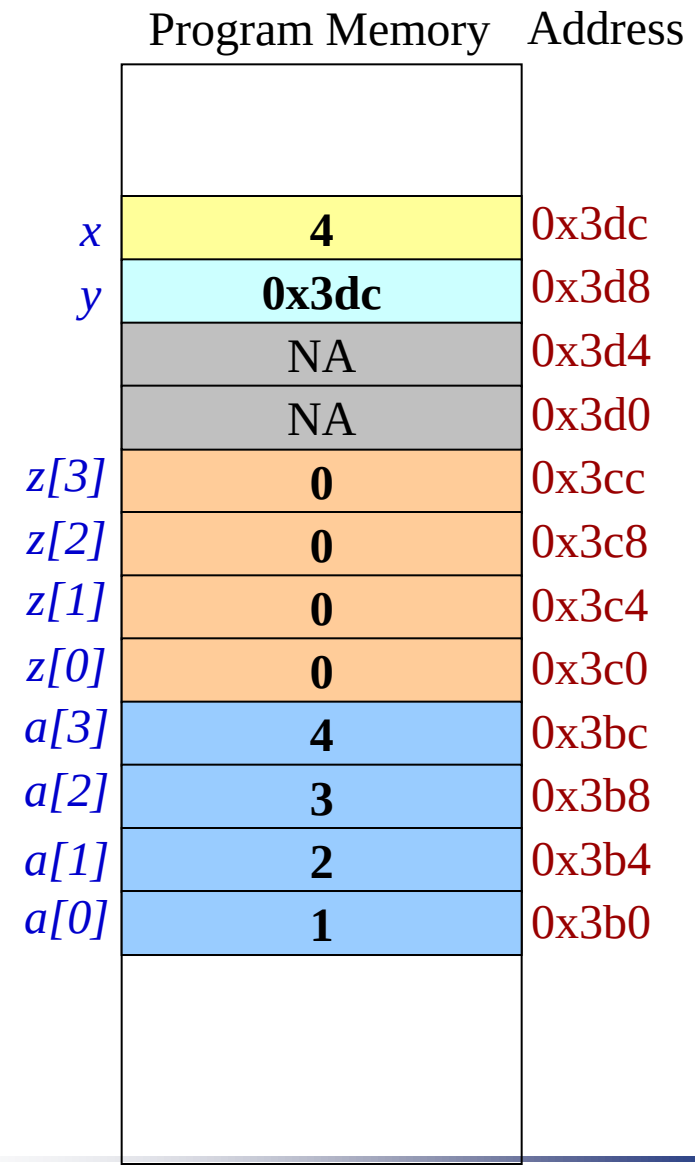
- An array identifier is equivalent to a pointer that references the first element of the array
  - `int x[5], *ptr;`
    `ptr = &x[0]` is equivalent to `ptr = x;`
- Pointer arithmetic and arrays:
  - `int x[5];`
    `x[2]` is the same as `*(x + 2)`, the compiler will assume you mean 2 objects beyond element x.

Pointers

- For any type T, you may form a pointer type to T.
  - Pointers may reference a function or an object.
  - The value of a pointer is the address of the corresponding object or function
  - Examples: `int *i; char *x; int (*myfunc)();`
- Pointer operators: ***** dereferences a pointer, *&* creates a pointer (reference to)
  - ```
    int i = 3; int *j = &i;
    *j = 4; printf("i = %d\n", i); // prints i = 4
    ```
  - ```
    int myfunc (int arg);
    int (*fptr)(int) = myfunc;
    i = fptr(4); // same as calling myfunc(4);
    ```
- Generic pointers:
  - Traditional C used (char *)
  - Standard C uses (void *) – these can not be dereferenced or used in pointer arithmetic. So they help to reduce programming errors
- Null pointers: use ***NULL*** or ***0***. *It is a good idea to always initialize pointers to NULL.*

Pointers in C (and C++)

```
Step 1:
int main (int argc, argv) {
 int  x = 4;
 int *y = &x;
 int *z[4] = {NULL, NULL, NULL, NULL};
 int  a[4] = {1, 2, 3, 4};
...
```



Note: The compiler converts z[1] or *(z+1) to
Value at address (Address of z + sizeof(int));

In C you would write the byte address as:

```
(char *)z + sizeof(int);
```

or letting the compiler do the work for you

```
(int *)z + 1;
```

Basic Types and Operators

- Basic data types
  - Types: *char, int, float and double*
  - Qualifiers: *short, long, unsigned, signed, const*
- Constant: 0x1234, 12, “Some string”
- Enumeration:
  - Names in different enumerations must be distinct
  - ```
    enum WeekDay_t {Mon, Tue, Wed, Thur, Fri};
    enum WeekendDay_t {Sat = 0, Sun = 4};
    ```
- Arithmetic: +, -, *, /, %
  - prefix ++i or --i ; increment/decrement before value is used
  - postfix i++, i--; increment/decrement after value is used
- Relational and logical: <, >, <=, >=, ==, !=, &&, ||
- Bitwise: &, |, ^ (xor), <<, >>, ~(ones complement)
    ```

Structs and Unions

- structures
  - `struct MyPoint {int x, int y};`
  - `typedef struct MyPoint MyPoint_t;`
  - `MyPoint_t point, *ptr;`
  - `point.x = 0;point.y = 10;`
  - `ptr = &point; ptr->x = 12; ptr->y = 40;`
- unions
  - `union MyUnion {int x; MyPoint_t pt; struct {int 3; char c[4]} S;};`
  - `union MyUnion x;`
  - Can only use one of the elements. Memory will be allocated for the largest element

Conditional Statements (if/else)

```
if (a < 10)
  printf("a is less than 10\n");
else if (a == 10)
  printf("a is 10\n");
else
  printf("a is greater than 10\n");
```

- If you have compound statements then use brackets (blocks)
  - ```
    if (a < 4 && b > 10) {
      c = a * b; b = 0;
      printf("a = %d, a's address = 0x%08x\n", a, &a);
    } else {
      c = a + b; b = a;
    }
    ```
- These two statements are equivalent:
  - `if (a) x = 3; else if (b) x = 2; else x = 0;`
  - `if (a) x = 3; else {if (b) x = 2; else x = 0;}`
- Is this correct?
  - ```
    if (a) x = 3; else if (b) x = 2;
    else (z) x = 0; else x = -2;
    ```

Conditional Statements (switch)

```
int c = 10;
switch (c) {
  case 0:
    printf("c is 0\n");
    break;
  ...
  default:
    printf("Unknown value of c\n");
    break;
}
```

- What if we leave the break statement out?
- Do we need the final break statement on the default case?

Loops

```
for (i = 0; i < MAXVALUE; i++) {
      dowork();
   }
while (c != 12) {
      dowork();
   }
do {
      dowork();
   } while (c < 12);
```

- flow control
  - **break** – exit innermost loop
  - **continue** – perform next iteration of loop
- Note, all these forms permit one statement to be executed. By enclosing in brackets we create a block of statements.

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Acknowledgement

- The majority contents of this ppt is from Dr. Fred Kuhns from Applied Research Laboratory, Department of Computer Science and Engineering, Washington University in St. Louis