C for Java Programmers

Advanced Programming

Overview

- Why learn C after Java?
- A brief background on C
- C preprocessor
- Modular C programs

Why learn C (after Java)?

- Both high-level and low-level language
  - OS: user interface to kernel to device driver
  - Better control of low-level mechanisms
- Memory allocation, specific memory locations
- Performance sometimes better than Java (Unix, NT!)
  - usually more predictable (also: C vs. C++)
- Java hides many details needed for writing OS code
  - But,...
  - Memory management responsibility
  - Explicit initialization and error detection
  - generally, more lines for same functionality
  - More room for mistakes

Why learn C, cont’d.

- Most older code is written in C (or C++)
  - Linux, *BSD
  - Windows
  - Most Java implementations
  - Most embedded systems
- Philosophical considerations:
  - Being multi-lingual is good!
  - Should be able to trace program from UI to assembly (EEs: to electrons)

C pre-history

- 1960s: slew of new languages
  - COBOL for commercial programming (databases)
  - FORTRAN for numerical and scientific programs
  - PL/I as second-generation unified language
  - LISP, Simula for CS research, early AI
  - Assembler for operating systems and timing-critical code
- Operating systems:
  - OS/360
  - MIT/GE/Bell Labs Multics (PL/I)

Credits

- Software Construction (J. Shepherd)
- Operating Systems at Cornell (Indranil Gupta)
**C pre-history**

- Bell Labs (research arm of Bell System -> AT&T -> Lucent) needed own OS
- BCPL as Multics language
- Ken Thompson: B
- Unix = Multics – bits
- Dennis Ritchie: new language = B + types
- Development on DEC PDP-7 with 8K 16-bit words

**C history**

- C
  - Dennis Ritchie in late 1960s and early 1970s
  - systems programming language
  - make OS portable across hardware platforms
  - not necessarily for real applications – could be written in Fortran or PL/I
- C++
  - Bjarne Stroustrup (Bell Labs), 1980s
  - object-oriented features
- Java
  - James Gosling in 1990s, originally for embedded systems
  - object-oriented, like C++
  - ideas and some syntax from C

**C for Java programmers**

- Java is mid-90s high-level OO language
- C is early-70s procedural language
- C advantages:
  - Direct access to OS primitives (system calls)
  - Fewer library issues – just execute
- (More) C disadvantages:
  - language is portable, APIs are not
  - memory and “handle” leaks
  - preprocessor can lead to obscure errors

**C vs. C++**

- We’ll cover both, but C++ should be largely familiar
- Very common in Windows
- Possible to do OO-style programming in C
- C++ can be rather opaque: encourages “clever” programming

**Aside: “generations” and abstraction levels**

- Binary, assembly
- Fortran, Cobol
- PL/I, APL, Lisp, ...
- C, Pascal, Ada
- C++, Java, Modula3
- Scripting: Perl, Tcl, Python, Ruby, ...
- XML-based languages: CPL, VoiceXML

**C vs. Java**

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>object-oriented</td>
<td>function-oriented</td>
</tr>
<tr>
<td>strongly-typed</td>
<td>can be overridden</td>
</tr>
<tr>
<td>polymorphism (+, = =)</td>
<td>very limited (integer/float)</td>
</tr>
<tr>
<td>classes for name space</td>
<td>(mostly) single name space, file-oriented</td>
</tr>
<tr>
<td>macros are external, rarely used</td>
<td>macros common (preprocessor)</td>
</tr>
<tr>
<td>layered I/O model</td>
<td>byte-stream I/O</td>
</tr>
</tbody>
</table>
## Java vs. C

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic memory management</td>
<td>function calls (C++ has some support)</td>
</tr>
<tr>
<td>no pointers</td>
<td>pointers (memory addresses) common</td>
</tr>
<tr>
<td>by-reference, by-value</td>
<td>by-value parameters</td>
</tr>
<tr>
<td>exceptions, exception handling</td>
<td>if (f()) &lt; 0) {error} OS signals</td>
</tr>
<tr>
<td>concurrency (threads)</td>
<td>library functions</td>
</tr>
</tbody>
</table>

---

## C program

- collection of functions
- one function – `main()` – is starting function
- running executable (default name a.out) starts main function
- typically, single program with all user code linked in – but can be dynamic libraries (.dll, .so)

```java
#include <stdio.h>

int main(int argc, char *argv[]) {
    System.out.println("Hello, World");
    return 0;
}
```

---

## What does this C program do?

```c
#include <stdio.h>

struct list { int data; struct list *next; }

void main() {
    void add(struct list *head, struct list *list, int data) {
        struct list *tail = NULL;
        int tail_data;
        for (int i = 0; i < 10; i++) {
            tail_data = i;
            add(head, tail, tail_data);
        }
    }

    int main() {
        return 0;
    }
}
```

---

## C vs. Java

- Java program
  - collection of classes
  - class containing main method is starting class
  - running `java StartClass` invokes `StartClass.main` method
  - JVM loads other classes as required

```java
public class hello {
    public static void main(String args[]) {
        System.out.println("Hello, World");
        return 0;
    }
}
```
What does this C program, do - cont’d?

```c
void delete (struct list *head, struct list *tail){
    struct list *temp;
    if(head==tail){
        free(head); head=tail=NULL;
    } else{
        temp=head->next; free(head); head=temp;
    }
}
```

---

Simple example

```c
#include <stdio.h>

void main(void) {
    printf("Hello World. \n \t and you \n ");
    /* print out a message */
    return;
}
```

$Hello World.
and you !$

---

Dissecting the example

- `#include <stdio.h>`
- Include header file `stdio.h`
- # lines processed by pre-processor
- No semicolon at end
- Lower-case letters only — C is case-sensitive
- `void main(void)` is the only code executed
- `printf("/ message you want printed ");`
- `\n = newline, \t = tab`
- `\ in front of other special characters within printf.`
  - `printf("Have you heard of "The Rock"? \n\n");`

---

Executing the C program

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

- `argc` is the argument count
- `argv` is the argument vector
- Array of strings with command-line arguments
- The `int` value is the return value
- Convention: 0 means success, > 0 some error
- Can also declare as void (no return value)

---

Executing a C program

- Name of executable + space-separated arguments
- `$ a.out 1 23 'third arg'`

```
argc 4
argv
  a.out 1 23 "third arg"
```

---

Executing a C program

- If no arguments, simplify:
  ```c
  int main() {
    puts("Hello World");
    exit(0);
  }
  ```
- Uses `exit()` instead of return — same thing.
Executing C programs

- Scripting languages are usually interpreted
  - perl (python, Tcl) reads script, and executes it
  - sometimes, just-in-time compilation – invisible to user
- Java programs semi-interpretated:
  - javac converts foo.java into foo.class
  - not machine-specific
  - byte codes are then interpreted by JVM
- C programs are normally compiled and linked:
  - gcc converts foo.c into a.out
  - a.out is executed by OS and hardware

The C compiler gcc

- gcc invokes C compiler
- gcc translates C program into executable for some target
- default file name a.out
- also "cross-compilation"

```
$ gcc hello.c
$ a.out
Hello, World!
```

gcc

- Behavior controlled by command-line switches:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o file</td>
<td>output file for object or executable</td>
</tr>
<tr>
<td>-Wall</td>
<td>all warnings – use always!</td>
</tr>
<tr>
<td>-c</td>
<td>compile single module (non-main)</td>
</tr>
<tr>
<td>-g</td>
<td>insert debugging code (gdb)</td>
</tr>
<tr>
<td>-p</td>
<td>insert profiling code</td>
</tr>
<tr>
<td>-l</td>
<td>library</td>
</tr>
<tr>
<td>-E</td>
<td>preprocessor output only</td>
</tr>
</tbody>
</table>

Using gcc

- Two-stage compilation
  - pre-process & compile: gcc -c hello.c
  - link: gcc -o hello hello.o
- Linking several modules:

```
gcc -c a.c  a.o
gcc -c b.c  b.o
gcc -o hello a.o b.o
```

Using math library

```
gcc -o calc calc.c -lm
```

Error reporting in gcc

- Multiple sources
  - preprocessor: missing include files
  - parser: syntax errors
  - assembler: rare
  - linker: missing libraries
Error reporting in gcc

- If gcc gets confused, hundreds of messages
  - fix first, and then retry – ignore the rest
- gcc will produce an executable with warnings
  - don't ignore warnings – compiler choice is often not what you had in mind
- Does not flag common mindos
  - if (x = 0) VS. if (x == 0)

gcc errors

- Produces object code for each module
- Assumes references to external names will be resolved later
- Undefined names will be reported when linking:
  - undefined symbol first referenced in file
  - ld fatal: Symbol referencing errors
  - No output written to file.

C preprocessor

- The C preprocessor (cpp) is a macro-processor which
  - manages a collection of macro definitions
  - reads a C program and transforms it
- Example:
  - `#define MAXVALUE 100`
  - `#define check(x) (x < MAXVALUE)`
  - `if (check(x)) {...}` becomes
  - `if ((x) < 100) {...}`

C preprocessor

- Preprocessor directives start with # at beginning of line:
  - define new macros
  - input files with C code (typically, definitions)
  - conditionally compile parts of file
- gcc -E shows output of preprocessor
- Can be used independently of compiler

C preprocessor

- `#define name const-expression`
- `#define name (param1,param2,...) expression`
- `#undef symbol`
  - replaces name with constant or expression
  - textual substitution
  - symbolic names for global constants
  - in-line functions (avoid function call overhead)
  - mostly unnecessary for modern compilers
  - type-independent code

C preprocessor

- Example: `#define MAXLEN 255`
- Lots of system .h files define macros
- invisible in debugger
  - `getchar()`, `putchar()` in stdio library
- Caution: don't treat macros like function calls
  - `#define valid(x) ((x) > 0 && (x) < 20)`
  - `if (valid(x++)) {...}
  - `valid(x++) -> (x++) > 0 && (x++) < 20`
**C preprocessor – file inclusion**

- `#include <filename.h>`
- `#include <filename.h>`
- inserts contents of filename into file to be compiled
- "filename" relative to current directory
- `<filename>` relative to `/usr/include`
- `gcc -i` flag to re-define default
- import function prototypes (cf. Java import)
- Examples:
  - `#include <stdio.h>`
  - `#include "mydefs.h"`
  - `#include "/home/alice/program/defs.h"`

**C preprocessor – conditional compilation**

- `#if expression`
- `code segment 1`
- `#else`
- `code segment 2`
- `#endif`
- preprocessor checks value of expression
- if true, outputs code segment 1, otherwise code segment 2
- machine or OS-dependent code
- can be used to comment out chunks of code – bad!
- `#define OS linux`
  - `#if OS == linux`
    - `puts("Linux!");`
  - `#else`
    - `puts("Something else");`
  - `#endif`

**C preprocessor – ifdef**

- For boolean flags, easier:
  - `#ifdef name`
  - `code segment 1`
  - `#else`
  - `code segment 2`
  - `#endif`
- preprocessor checks if name has been defined
  - `#define USEDDB`
- if so, use code segment 1, otherwise 2

**Advice on preprocessor**

- Limit use as much as possible
- Subtle errors
- Not visible in debugging
- Code hard to read
- Much of it is historical baggage
- There are better alternatives for almost everything:
  - `#define INT16` for type definitions
  - `#define MAXLEN` for constants
  - `#define max(a,b)` for regular functions
  - Comment out code with CVS functions
- Limit to `.h` files, to isolate OS & machine-specific code

**Comments**

- `/* any text until */`
- `// C++-style comments – careful!`
- No `/** */`, but doc++ has similar conventions
- Convention for longer comments:
  - `/*`
  - `AverageGrade()`
  - Given an array of grades, compute the average.
  - `*/`
- Avoid `****` boxes – hard to edit, usually look ragged.

**Numeric data types**

<table>
<thead>
<tr>
<th>type</th>
<th>bytes (typ.)</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>-128 ... 127</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>-32768 ... 32767</td>
</tr>
<tr>
<td>int, long</td>
<td>4</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>2^64</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>3.4E-38 (7 digits)</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>1.7E-308 (15 digits)</td>
</tr>
</tbody>
</table>
Remarks on data types

- Range differs – int is “native” size, e.g., 64 bits on 64-bit machines, but sometimes int = 32 bits, long = 64 bits
- Also, unsigned versions of integer types
- same bits, different interpretation
- char = 1 "character", but only true for ASCII and other Western char sets

Example

```c
#include <stdio.h>

void main(void)
{
    int nstuedents = 0; /* Initialization, required */
    printf("How many students does Columbia have ?: ");
    scanf("%d", &nstuedents); /* Read input */
    printf("Columbia has %d students.
", nstuedents);
    return ;
}
```

$ How many students does Columbia have ?: 20000 (enter)
Columbia has 20000 students.

Explicit and implicit conversions

- Implicit: e.g., s = a (int) + b (char)
- Promotion: char -> short -> int -> ...
- If one operand is double, the other is made double
- If either is float, the other is made float, etc.
- Explicit: type casting – (type)
- Almost any conversion does something – but not necessarily what you intended

Type conversion

```c
#include <stdio.h>

void main(void)
{
    int i, j = 12; /* i not initialized, only j */
    float f1, f2 = 1.2;
    i = (int) f2; /* explicit: i <= 1.0 lost */
    f1 = i; /* implicit: f1 <= 1.0 */
    f1 = f2 + (int) j; /* explicit: f1 <= 1.2 + 12.0 */
    f1 = f2 + j; /* implicit: f1 <= 1.2 + 12.0 */
}
```

Type conversion

```c
int x = 100000;
short s;

s = x;
printf("%d %d\n", x, s);
```

100000 -31072

C - no booleans

- C doesn’t have booleans
- Emulate as int or char, with values 0 (false) and 1 or non-zero (true)
- Allowed by flow control statements:
  ```c
  if (x = 0) {
      printf("something wrong");
  }
  ```
- Assignment returns zero -> false
**User-defined types**

- **typedef** gives names to types:
  - `typedef short int smallNumber;`
  - `typedef unsigned char byte;`
  - `typedef char String[100];`

  ```c
  smallNumber x;
  byte b;
  String name;
  ```

**Defining your own boolean**

- **typedef char boolean;**
- `#define FALSE 0`
- `#define TRUE 1`
- Generally works, but beware:
  - check = x > 0;
  - if (check == TRUE) {...}
- If x is positive, check will be non-zero, but may not be 1.

**Enumerated types**

- Define new integer-like types as enumerated types:
  - `typedef enum { Red, Orange, Yellow, Green, Blue, Violet } Color;`
  - `enum weather {rain, snow=2, sun=4);`
- Look like C identifiers (names)
- Are listed (enumerated) in definition
- Treated like integers
  - Can add, subtract – even `color + weather`
  - Can’t print as symbol (unlike Pascal)
  - But debugger generally will

**Enumerated types**

- Just syntactic sugar for ordered collection of integer constants:
  - `typedef enum { Red, Orange, Yellow } Color;`
- *is like*
  - `#define Red 0`
  - `#define Orange 1`
  - `#define Yellow 2`
- `typedef enum {False, True} boolean;`

**Objects (or lack thereof)**

- C does not have objects (C++ does)
- Variables for C’s primitive types are defined very similarly:
  - `short int x;`
  - `char ch;`
  - `float pi = 3.1415;`
- Variables defined in {} block are active only in block
- Variables defined outside a block are global (persist during program execution), but may not be globally visible (static)

**Data objects**

- Variable = container that can hold a value
  - In C, pretty much a CPU word or similar
- Default value is (mostly) undefined – treat as random
  - Compiler may warn you about uninitialized variables
- `ch = 'g'; x = x + 4;`
- Always pass by value, but can pass address to function:
  - `scanf("%s", &x, &f);`
Data objects

- Every data object in C has
  - a name and data type (specified in definition)
  - an address (its relative location in memory)
  - a size (number of bytes of memory it occupies)
  - visibility (which parts of program can refer to it)
  - lifetime (period during which it exists)
- Warning:
  ```
  int *foo(char x) {
    return &x;
  }
  pt = foo(x);
  *pt = 17;
  ```

Data object creation

```c
int x;
int arr[20];
int main(int argc, char *argv[]) {
  int i = 20;
  for (x = i; x > 1; x--)
    
  int f(int n)
  {
    int a, *p;
    a = 1;
    p = (int *)malloc(sizeof int);
  }
```

Memory allocation

- Note: malloc() does not initialize data
- void *calloc(size_t n, size_t elsize) does initialize (to zero)
- Can also change size of allocated memory blocks:
  ```c
  void *realloc(void *ptr, size_t size)
  ptr points to existing block, size is new size
  New pointer may be different from old, but content is copied.
  ```
Data objects and pointers

- The memory address of a data object, e.g., int x
  - can be obtained via &x
  - has a data type int (* in general, type *)
  - has a value which is a large (4/8 byte) unsigned integer
  - can have pointers to pointers: int **
- The size of a data object, e.g., int x
  - can be obtained via sizeof(x) or sizeof(x)
  - has data type size_t, but is often assigned to int (bad!)
  - has a value which is a smallish integer
  - is measured in bytes

Data objects and pointers

- Every data type T in C/C++ has an associated pointer type T *
- A value of type * is the address of an object of type T
- If an object int *xp has value &x, the expression *xp dereferences the pointer and refers to x, thus has type int

```c
int *xp = 42;
int x;
```

Data objects and pointers

- If p contains the address of a data object, then *p allows you to use that object
  *p is treated just like normal data object
  ```c
  int a, b, c, d;
  *d = 17; /* BAD idea */
  a = 2; b = 3; c = a; d = &b;
  if (*c == *d) puts("Same value");
  *c = 3;
  if (*c == *d) puts("Now same value");
  c = d;
  if (c == d) puts("Now same address");
  ```

void pointers

- Generic pointer
- Unlike other pointers, can be assigned to any other pointer type:
  ```c
  void *v;
  char *s = v;
  ```
- Acts like char * otherwise:
  ```c
  v++, sizeof(*v) = 1;
  ```

Control structures

- Same as Java
- sequencing: ;
- grouping: {...}
- selection: if, switch
- iteration: for, while

Sequencing and grouping

- statement1 ; statement2; statement n;
  - executes each of the statements in turn
  - a semicolon after every statement
  - not required after a (...) block
- { statements} {declarations statements}
  - treat the sequence of statements as a single operation (block)
  - data objects may be defined at beginning of block
The **if** statement

- Same as Java
  
  ```c
  if (condition) { statements; }
  else if (condition) { statements; }
  else { statements; }
  ```

- Evaluates statements until find one with non-zero result
- Executes corresponding statements

---

**The switch statement**

- Allows choice based on a single value
  
  ```c
  switch(expression) {
    case const1: statements1; break;
    case const2: statements2; break;
    default: statementsn;
  }
  ```

- Effect: Evaluates integer expression
- Looks for case with matching value
- Executes corresponding statements (or defaults)

---

**Repetition**

- C has several control structures for repetition

<table>
<thead>
<tr>
<th>Statement</th>
<th>Repeats an action…</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(c){ }</td>
<td>zero or more times, while condition is ≠ 0</td>
</tr>
<tr>
<td>do (...) while(c)</td>
<td>one or more times, while condition is ≠ 0</td>
</tr>
<tr>
<td>for (start; cond; upd)</td>
<td>zero or more times, with initialization and update</td>
</tr>
</tbody>
</table>

---

**The **break** statement**

- Break allows early exit from one loop level

```c
for (init; condition; next) {
  statements1;
  if (condition2) break;
  statements2;
}
```
The continue statement

- continue skips to next iteration, ignoring rest of loop body
- does execute next statement
  for (init; condition; swx) {
    statement2;
    if (condition2) continue;
    statement2;
  }
- often better written as if with block

Structured data objects

- Structured data objects are available as

<table>
<thead>
<tr>
<th>object</th>
<th>property</th>
</tr>
</thead>
<tbody>
<tr>
<td>array ()</td>
<td>enumerated, numbered from 0</td>
</tr>
<tr>
<td>struct</td>
<td>names and types of fields</td>
</tr>
<tr>
<td>union</td>
<td>occupy same space (one of)</td>
</tr>
</tbody>
</table>

Arrays

- Arrays are defined by specifying an element type and number of elements
  - int vec[100];
  - char str[30];
  - float m[10][10];
- For array containing N elements, indexes are 0..N-1
- Stored as linear arrangement of elements
- Often similar to pointers

Arrays

- C does not remember how large arrays are (i.e., no length attribute)
- int x[10]; x[10] = 5; may work (for a while)
- In the block where array A is defined:
  - sizeof A gives the number of bytes in array
  - can compute length via sizeof A / sizeof A[0]
- When an array is passed as a parameter to a function
  - the size information is not available inside the function
  - array size is typically passed as an additional parameter
    - printArray(A, VEC_SIZE);
    - or as part of a struct (best, object-like)
    - or globally
    - #define VEC_SIZE 10

Arrays

- Array elements are accessed using the same syntax as in Java: array[index]
- Example (iteration over array):
  ```c
  int i, sum = 0;
  ...
  for (i = 0; i < VEC_SIZE; i++)
    sum += vec[i];
  ```
- C does not check whether array index values are sensible (i.e., no bounds checking)
  - vec[-1] or vec[10000] will not generate a compiler warning!
  - if you’re lucky, the program crashes with Segmentation fault (core dumped)
- C references arrays by the address of their first element
  - array is equivalent to &array[0]
- can iterate through arrays using pointers as well as indexes:
  ```c
  int *v, *last;
  int sum = 0;
  last = &vec[VEC_SIZE-1];
  for (v = vec; v <= last; v++)
    sum += *v;
  ```
2-D arrays

- 2-dimensional array
  ```
  int weekends[52][2];
  ```
- `weekends[2][1]` is same as *(weekends+2*2+1)
- `NOT *weekends+2*2+1` : this is an int!

Aside: void, void *

- Function that doesn't return anything declared as void
- No argument declared as void
- Special pointer *void can point to anything

```c
#include <stdio.h>

extern void *f(void);
void *f(void) {
  printf("the big void\n");
  return NULL;
}
int main(void) {
  f();
}
```

Arrays - example

```c
#include <stdio.h>

void main(void) {
  /* 12 cells, one cell per student */
  int index, row = 0;
  /* Always initialize array before use */
  for (index = 0; index < 12; index++) {
    number[index] = index;
  }
  /* row, number[index]=index, will cause error why ??*/
  for (index = 0; index < 12; index = index + 1) {
    sum += number[index];
  }
  return;
}
```

Overriding functions - function pointers

- overriding: changing the implementation, leave prototype
- in C, can use function pointers, returnType (*ptrName)(arg1, arg2, ...);
- for example, int (*fp)(double x); is a pointer to a function that return an integer
- double * (*gp)(int) is a pointer to a function that returns a pointer to a double

structs

- Similar to fields in Java object/class definitions
- components can be any type (but not recursive)
- accessed using the same syntax struct.field
- Example:
  ```
  struct { int a; char y; float z; } rec;
  ...
  r.x = 3; r.y = 'a'; r.z = 3.1415;
  ```

- Record types can be defined
  - using a tag associated with the struct definition
  - wrapping the struct definition inside a typedef
- Examples:
  ```
  struct complex 
  { double real; double imag; }
  struct point 
  { double x; double y; } corner;
  typedef struct complex Complex;
  struct complex a, b;
  Complex c,d;
  ```
- `a` and `b` have the same size, structure and type
- `a` and `c` have the same size and structure, but different types

```c
struct
```
**structs**

- Overall size is sum of elements, plus padding for alignment:
  ```c
  struct {
    char x;
    int y;
    char z;
  } s1;  sizeof(s1) = ?
  struct {
    char x, z;
    int y;
  } s2;  sizeof(s2) = ?
  ```

**structs - example**

```c
struct person {
  char name[41];
  int age;
  float height;
} p;
```

```c
struct {
  /* embedded structure */
  int month;
  int day;
  int year;
  } bday;
```

```c
struct person me;  
  me.name = "Gun";  
  me.bday.year = 1971;
  /
```

```c
struct person class[60];
```

```c
/* array of info about everyone in class */
class[0].name = "Gun";  
class[0].bday.year = 1971;  
```

**Dereferencing pointers to struct elements**

- Pointers commonly to struct's
  ```c
  (*sp).element = 42;
  y = (*sp).element;
  ```
- Note: `*sp.element` doesn't work
- Abbreviated alternative:
  ```c
  sp->element = 42;
  y = sp->element;
  ```

**Bit fields**

- On previous slides, labeled integers with size in bits (e.g., pt:7)
- Allows aligning struct with real memory data, e.g., in protocols or device drivers
- Order can differ between little/big-endian systems
- Alignment restrictions on modern processors - *natural alignment*
- Sometimes clearer than `(x & 0x8000) >> 31`

**Unions**

- Like structs:
  ```c
  union u_bag {
    int i_val;
    float f_val;
    char *val;
  } u;
  ```
- but occupy same memory space
- can hold different types at different times
- overall size is largest of elements
More pointers

```c
int month[12]; /* month is a pointer to base address 430*/
month[3] = 7; /* month address + 3 * int elements */
ptr = month + 2; /* ptr points to month[1] */
ptr[1] = 12; /* ptr address + 1 * int elements */
Thus, month[7] is now 12 */

ptr++; /* ptr += 4.3 + 1 * size of int = 44 */
ptr + 4) = 12; /* accessing ptr(4) i.e., array[4] */

Now, month[6], *(month+6), (month+4)[2],
ptr[3], *(ptr+3) are all the same integer variable.
```

Functions - why and how?

- If a program is too long
- Modularization - easier to code
- debug
- Code reuse
- Passing arguments to functions
  - By value
  - By reference
- Returning values from functions
  - By value
  - By reference

Functions

- Prototypes and functions (cf. Java interfaces)
  - extern int putchar(int c);
  - int putchar(int c) {
    something interesting here
  }
- If defined before use in same file, no need for prototype
- Typically, prototype defined in .h file
- Good idea to include <.h> in actual definition

Functions

- static functions and variables hide them to those outside the same file:
  - static int x;
  - static int times2(int c) {
    return c*2;
  }
- compare protected class members in Java.

Functions - extern

```c
#include <stdio.h>
extern char userLine[20]; /* global variable defined in another file */
char userLine[20]; /* global for this file */
void dummy(void);

void main(void) {
  char userLine[20]; /* different from earlier userLine[30] */
  . . . /* restricted to this Func */
}
void dummy()
  extern char userLine[]; /* the global userLine[30] */
  . . .
}```
Overloading functions

- **Java:**
  - void product(double x, double y);
  - void product(vector x, vector y);
- **C** doesn't support this, but allows variable number of arguments:
  - debug("%d %f", x, f);
  - declared as void debug(char *fmt, ...);
- at least one known argument

---

Overloading functions

- **Limitations:**
  - cannot access arguments in middle
  - needs to copy to variables or local array
  - client and function need to know and adhere to type

---

Data hiding in C

- **C** doesn't have classes or private members, but this can be approximated
- Implementation defines real data structure:
  ```c
  #define QUEUE_C
  #include "queue.h"
  typedef struct queue_t {
    struct queue_t *next;
    int data;
  } queue_t, queuestruct_t;
  queue_t *mallocQueue( void ) {
    return q;
  }
  ```
- Header file defines public data:
  ```c
  #define QUEUE_C
  typedef struct queue_t *queue_t;
  ```

---

Program with multiple files

- **Library headers**
  - Standard
  - User-defined

---

Pointer to function

- `int func(); /*function returning integer*/`
- `int *func(); /*function returning pointer to integer*/`
- `int (*func)(); /*pointer to function returning integer*/`
- `int *(*func)(); /*pointer to function returning ptr to int*/`
Function pointers

```c
#include <stdio.h>

int (*fp)(void);
double* (*gp)(int);
int f(void)

double *g(int);

fp=f;
gp=g;

int i = fp();
double *g = (*gp)(17); /* alternative */
```

Pointer to function - example

```c
void myproc (int d);
void mycaller (void (*f)(int), int param);

void main(void)
{
    myproc(10); /* call myproc with parameter 10 */
    mycaller(myproc, 10); /* and do the same again */
}

void mycaller (void (*f)(int), int param)
{
    (*f)(param); /* call function *f with param */
}

void myproc (int d)
{
    . . . /* do something with d */
}
```

Libraries

- C provides a set of standard libraries for
  - Numerical math functions: `<math.h>`
  - Character strings: `<string.h>`
  - Character types: `<ctype.h>`
  - I/O: `<stdio.h>`

The math library

- Include `<math.h>`
  - Careful: `sqrt(x)` without header file may give wrong result!
  - `gcc -o compute main.o f.o -lm`
  - Uses normal mathematical notation:
    - `Math.sqrt(2)`
    - `sqrt(2)`
    - `Math.pow(x, 5)`
    - `pow(x, 5)`
    - `4*Math.pow(x, 3)`
      - `4* pow(x, 3)`

Characters

- The char type is an 8-bit byte containing ASCII code values (e.g., 'A' = 65, 'B' = 66, ...)
- Often, char is treated like (and converted to) int
- `<ctype.h>` contains character classification functions:
  - `isalnum(ch)`
  - `isalpha(ch)`
  - `isdigit(ch)`
  - `ispunct(ch)`
  - `isspace(ch)`
  - `isupper(ch)`
  - `islower(ch)`

Strings

- In Java, strings are regular objects
- In C, strings are just char arrays with a NUL (`\0`) terminator
  - "a cat" = `a c a t`
  - A literal string ("a cat")
    - is automatically allocated memory space to contain it and the terminating NUL
    - has a value which is the address of the first character
    - can't be changed by the program (common bug)
    - All other strings must have space allocated to them by the program
Strings

- We normally refer to a string via a pointer to its first character:
  ```c
  char *str = "my string";
  char *s;
  s = &str[0]; // s = str;
  
  C functions only know string ending by \0:
  char *str = "my string";
  ... 
  int i;
  for (i = 0; str[i] != '\0'; i++)
    putchar(str[i]);
  char *s;
  for (s = str; *s; s++) putchar(*s);
  ```

- Can treat like arrays:
  ```c
  char c;
  char line[100];
  for (i = 0; i < 100 && line[i]; i++) {
    if (!isalpha(line[i])) ... 
  }
  ```

Copying strings

- Copying content vs. copying pointer to content:
  ```c
  s = t; copies pointer – s and t now refer to the same memory location
  strcpy(s, t); copies content of t to s
  char mybuffer[100];
  ... 
  mybuffer = "a cat";
  is incorrect (but appears to work!)
  Use strcpy(mybuffer, "a cat") instead
  ```

Example string manipulation

```c
#include <stdio.h>
#include <string.h>
int main(void) {
  char line[100];
  char *family, *given, *gap;
  printf("Enter your name:"); fgets(line, 100, stdin);
  given = line;
  for (gap = line; *gap; gap++)
    if (isspace(*gap)) break;
  *gap = '\0';
  family = gap + 1;
  printf("Your name: %s, %s", family, given);
  return 0;
}
```
string.h library

- int strlen(const char *source)
  - returns number of chars, excluding NUL
- char *strchr(const char *source, const char ch)
  - returns pointer to first occurrence of ch in source; NUL if none
- char *strstr(const char *source, const char *search)
  - return pointer to first occurrence of search in source

Formatted strings

- String parsing and formatting (binary from/to text)
- int sscanf(char *string, char *format, ...)
  - parse the contents of string according to format
  - placed the parsed items into 3%, 4%, 5%, ... argument
  - return the number of successful conversions
- int sprintf(char *buffer, char *format, ...)
  - produce a string formatted according to format
  - place this string into the buffer
  - the 3%, 4%, 5%, ... arguments are formatted
  - return number of successful conversions

Formatted strings

- The format strings for scanf and sprintf contain
  - plain text (matched on input or inserted into the output)
  - formatting codes (which must match the arguments)
- The sprintf format string gives template for result string
- The sscanf format string describes what input should look like

Formatted strings

- Formatting codes for sscanf

<table>
<thead>
<tr>
<th>Code</th>
<th>meaning</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>matches a single character</td>
<td>char</td>
</tr>
<tr>
<td>%d</td>
<td>matches an integer in decimal</td>
<td>int</td>
</tr>
<tr>
<td>%f</td>
<td>matches a real number (ddd.dd)</td>
<td>float</td>
</tr>
<tr>
<td>%s</td>
<td>matches a string up to white space</td>
<td>char *</td>
</tr>
<tr>
<td>%[^c]</td>
<td>matches string up to next cchar</td>
<td>char *</td>
</tr>
</tbody>
</table>

Formatted strings - examples

```c
char *msg = “Hello there”;  
char *nums = “1 3 5 7 9”; 
char s[10], t[10];  
int a, b, c, n;  

n = sscanf(msg, “%s %s”, s, t);  
n = printf(“%s %s %s”, t, s);  
n = sscanf(nums, “%d %d %d”, &a, &b, &c);  

printf(“%d flowers%”, n, n > 1 ? “s” : “”);  
printf(“a = %d, answer = %d\n”, a, b+c);  
```
The stdio library

- Access stdio functions by
  - using #include <stdio.h> for prototypes
  - compiler links it automatically
- defines FILE * type and functions of that type
- data objects of type FILE *
  - can be connected to file system files for reading and writing
  - represent a buffered stream of chars (bytes) to be written or read
- always defines stdin, stdout, stderr

The stdio library: fopen(), fclose()

- Opening and closing FILE * streams:
  FILE *fopen(const char *path, const char *mode)
  - open the file called path in the appropriate mode
  - modes: "r" (read), "w" (write), "a" (append), "r+" (read & write)
  - returns a new FILE * if successful, NULL otherwise
  - fclose(FILE *stream)
  - close the stream FILE *
  - return 0 if successful, EOF if not

stdio - character I/O

int getchar()
- read the next character from stdin; returns EOF if none
int fgetc(FILE *in)
- read the next character from FILE in; returns EOF if none
int putc(char c)
- write the character c onto stdout; returns c or EOF
int fputc(int c, FILE *out)
- write the character c onto out; returns c or EOF

stdio - line I/O

char *fgets(char *buf, int size, FILE *in)
- read the next line from in into buffer buf
  - halts at ‘\n’ or after size-1 characters have been read
  - the ‘\n’ is read, but not included in buf
  - returns pointer to strbuf if ok, NULL otherwise
  - do not use gets(char *) – buffer overflow
int fputs(const char *str, FILE *out)
- writes the string str to out, stopping at ‘\0’
- returns number of characters written or EOF

stdio - formatted I/O

int fscanf(FILE *in, const char *format, ...)
- read text from stream according to format
int fprintf(FILE *out, const char *format, ...)
- write the string to output file, according to format
int printf(const char *format, ...)
- equivalent to fprintf(stdout, format, ...)
- Warning: do not use fscanf(...); use fgets(str, ...); sscanf(str, ...);

Before you go....

- Always initialize anything before using it (especially pointers)
- Don’t use pointers after freeing them
- Don’t return a function’s local variables by reference
- No exceptions – so check for errors everywhere
  - memory allocation
  - system calls
- Murphy’s law, C version: anything that can’t fail, will fail
- An array is also a pointer, but its value is immutable.