C for Java Programmers

Advanced Programming
Credits

- *Software Construction* (J. Shepherd)
- *Operating Systems* at Cornell (Indranil Gupta)
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)
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>
## C vs. Java

<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}</td>
</tr>
<tr>
<td>concurrency (threads)</td>
<td>OS signals</td>
</tr>
<tr>
<td></td>
<td>library functions</td>
</tr>
</tbody>
</table>
# C vs. Java

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>length of array</td>
<td>on your own</td>
</tr>
<tr>
<td>string as type</td>
<td>just bytes (char []),</td>
</tr>
<tr>
<td></td>
<td>with 0 end</td>
</tr>
<tr>
<td>dozens of common</td>
<td>OS-defined</td>
</tr>
<tr>
<td>libraries</td>
<td></td>
</tr>
</tbody>
</table>
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
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)
C vs. Java

public class hello
{
    public static void main (String args [] ) {
        System.out.println ("Hello world");
    }
}

#include <stdio.h>
int main(int argc, char *argv[])
{
    puts("Hello, World");
    return 0;
}
What does this C program do?

```c
#include <stdio.h>
struct list{int data; struct list *next};
struct list *start, *end;
void add(struct list *head, struct list *list, int data);
int delete(struct list *head, struct list *tail);

void main(void){
    start=end=NULL;
    add(start, end, 2);    add(start, end, 3);
    printf("First element: %d", delete(start, end));
}

void add(struct list *head, struct list *tail, int data){
    if(tail==NULL){
        head=tail=malloc(sizeof(struct list));
        head->data=data; head->next=NULL;
    }
    else{
        tail->next= malloc(sizeof(struct list));
        tail=tail->next; tail->data=data; tail->next=NULL;
    }
}
```
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

#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");`
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'

```
        4
argc  

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

Advanced Programming
Spring 2002
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-interpreted:
  - 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
Executing C programs

x.pl

perl

results

data

x.java

javadoc

java

args

x.c, x.cc

gcc, g++
a.out
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
  _print program.o
  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:
```c
#define MAXVALUE 100
#define check(x) ((x) < MAXVALUE)
if (check(i) { ...}
```

becomes
```c
if ((i) < 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

```c
#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

```c
#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!

```c
#define OS linux
...
#if OS == linux
  puts("Linux!");
#else
  puts("Something else");
#endif
```
C preprocessor - ifdef

- For boolean flags, easier:
  
  ```c
  #ifdef name
  code segment 1
  #else
  code segment 2
  #endif
  ```

- preprocessor checks if name has been defined
  - #define USEDB

- 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` -> type definitions
  - `#define MAXLEN` -> const
  - `#define max(a,b)` -> regular functions
  - comment out code -> 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>-65536...65535</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 – \texttt{int} is “native” size, e.g., 64 bits on 64-bit machines, but sometimes \texttt{int} = 32 bits, \texttt{long} = 64 bits

- Also, \texttt{unsigned} versions of integer types
  - same bits, different interpretation

- \texttt{char} = 1 “character”, but only true for ASCII and other Western char sets
Example

#include <stdio.h>

void main(void)
{
    int nstudents = 0; /* Initialization, required */

    printf("How many students does Columbia have ?:");
    scanf("%d", &nstudents); /* Read input */
    printf("Columbia has %d students.\n", nstudents);

    return ;
}

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

#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, 0.2 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 */
}
Explicit and implicit conversions

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

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 (n = 0) {
    printf(“something wrong”);
  }
  ```
- Assignment returns zero -> false
User-defined types

- \textbf{typedef gives names to types:}

  typedef short int smallNumber;
  typedef unsigned char byte;
  typedef char String[100];

  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:
  ```c
  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:

```c
typedef enum {
    Red, Orange, Yellow
} Color;
```

is like

```c
#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:
  
  ```c
  short int x;
  char ch;
  float pi = 3.1415;
  float f, g;
  ```

- 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
- \( ch = 'a'; x = x + 4; \)
- Always pass by value, but can pass address to function:
  - `scanf("%d%f", &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:

```c
int *foo(char x) {
    return &x;
}

pt = foo(x);
*pt = 17;
```
Data objects

- Unlike scripting languages and Java, all C data objects have a fixed size over their lifetime
  - except dynamically created objects
- size of object is determined when object is created:
  - global data objects at compile time (data)
  - local data objects at run-time (stack)
  - dynamic data objects by programmer (heap)
int x;
int arr[20];
int main(int argc, char *argv[]) {
    int i = 20;
    {into x; x = i + 7;}
}

int f(int n)
{
    int a, *p;
    a = 1;
    p = (int *)malloc(sizeof int);
}
Data object creation

- `malloc()` allocates a block of memory
- Lifetime until memory is freed, with `free()`.
- Memory *leakage* – memory allocated is never freed:

```c
char *combine(char *s, char *t) {
    u = (char *)malloc(strlen(s) + strlen(t) + 1);
    if (s != t) {
        strcpy(u, s); strcat(u, t);
        return u;
    } else {
        return 0;
    }
}
```
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:
  - `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.
Memory layout of programs

- Header info
- Code
- Data - Heap
- Data - stack
- Dynamic memory
- Local memory + function call stack

all malloc()s

all normal vars
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 small(ish) integer
  - is measured in bytes
Data objects and pointers

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

  if (condition$_1$) {statements$_1$}
  else if (condition$_2$) {statements$_2$}
  else if (condition$_{n-1}$) {statements$_{n-1}$}
  else {statements$_n$}

- evaluates statements until find one with non-zero result

- executes corresponding statements
The if statement

- Can omit {}, but careful

```c
if (x > 0)
    printf(“x > 0!”);
if (y > 0)
    printf(“x and y > 0!”);
```
The `switch` statement

- Allows choice based on a single value

```java
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)
The switch statement

Weather w;
switch(w) {
    case rain:
        printf("bring umbrella");
    case snow:
        printf("wear jacket");
        break;
    case sun:
        printf("wear sunscreen");
        break;
    default:
        printf("strange weather");
}
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 $\neq 0$</td>
</tr>
<tr>
<td>do {...} while(c)</td>
<td>one or more times, while condition is $\neq 0$</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 \texttt{continue} statement

- \texttt{continue} \textbf{skips to next iteration, ignoring rest of loop body}
- \textbf{does execute next statement}

\begin{verbatim}
for (init; condition1; next) {
    statement2;
    if (condition2) continue;
    statement2;
}
\end{verbatim}

- \textbf{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, VECSIZE);`
  - or as part of a struct (best, object-like)
  - or globally
    - `#define VECSIZE 10`
Arrays

- Array elements are accessed using the same syntax as in Java: array[index]
- Example (iteration over array):
  ```
  int i, sum = 0;
  ...
  for (i = 0; i < VECSIZE; 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)
Arrays

- 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[VECSIZE-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!
#include <stdio.h>
void main(void) {
    int number[12]; /* 12 cells, one cell per student */
    int index, sum = 0;
    /* Always initialize array before use */
    for (index = 0; index < 12; index++) {
        number[index] = index;
    }
    /* now, number[index]=index; will cause error: why ?*/
    for (index = 0; index < 12; index = index + 1) {
        sum += number[index]; /* sum array elements */
    }
    return;
}
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();
    f();
}
```
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 x; char y; float z;} rec;
  ...
  r.x = 3; r.y = 'a'; r.z= 3.1415;
  ```
structs

- Record types can be defined
  - using a tag associated with the struct definition
  - wrapping the struct definition inside a typedef

- Examples:
  ```c
  struct complex {double real; double imag;};
  struct point {double x; double y;} corner;
  typedef struct {double real; double imag;} 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
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;
    struct {         /* embedded structure */
        int month;
        int day;
        int year;
    } birth;
};

struct person me;
me.birth.year=1977;
struct person class[60];
    /* array of info about everyone in class */
class[0].name="Gun"; class[0].birth.year=1971;......
```
structs

- Often used to model real memory layout, e.g.,

```c
typedef struct {
    unsigned int version:2;
    unsigned int p:1;
    unsigned int cc:4;
    unsigned int m:1;
    unsigned int pt:7;
    u_int16 seq;
    u_int32 ts;
} rtp_hdr_t;
```
Dereferencing pointers to struct elements

- Pointers commonly to struct’s
  
  ```
  (*sp).element = 42;
  y = (*sp).element;
  ```

- Note: *sp.element doesn’t work

- Abbreviated alternative:
  
  ```
  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 \((x \& 0x8000) \gg 31\)
Unions

- **Like structs:**
  ```c
  union u_tag {
    int ival;
    float fval;
    char *sval;
  } u;
  ```
- but occupy same memory space
- can hold different types at different times
- overall size is largest of elements
int month[12]; /* month is a pointer to base address 430*/

month[3] = 7; /* month address + 3 * int elements
    => int at address (430+3*4) is now 7 */

ptr = month + 2; /* ptr points to month[2],
    => ptr is now (430+2 * int elements)= 438 */
ptr[5] = 12; /* ptr address + 5 int elements
    => int at address (434+5*4) is now 12.
    Thus, month[7] is now 12 */

ptr++; /* ptr <- 438 + 1 * size of int = 442 */

- 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);
  - putchar(‘A’);
  - int putchar(int c) {
     do 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:
  ```java
  static int x;
  static int times2(int c) {
    return c*2;
  }
  ```

- compare protected class members in Java.
Functions - const arguments

- Indicates that argument won’t be changed.
- Only meaningful for pointer arguments and declarations:
  ```c
  int c(const char *s, const int x) {
    const int VALUE = 10;
    printf("x = %d\n", VALUE);
    return *s;
  }
  ```
- Attempts to change *s will yield compiler warning.
Functions - extern

#include <stdio.h>

extern char user2line [20];  /* global variable defined in another file */
char user1line[30];           /* global for this file */
void dummy(void);

void main(void) {
    char user1line[20];       /* different from earlier user1line[30] */
    . . .                     /* restricted to this func */
}

void dummy(){
    extern char user1line[]; /* the global user1line[30] */
    . . .
}
Overloading functions – var. arg. list

- Java:
  ```java
  void product(double x, double y);
  void product(vector x, vector y);
  ```

- C doesn’t support this, but allows variable number of arguments:
  ```c
  debug("%d %f", x, f);
  debug("%c", c);
  ```

- declared as `void debug(char *fmt, ...);`
- at least one known argument
Overloading functions

- must include `<stdarg.h>`:

```c
#include <stdarg.h>
double product(int number, ...) {
    va_list list;
    double p;
    int i;
    va_start(list, number);
    for (i = 0, p = 1.0; i < number; i++) {
        p *= va_arg(list, double);
    }
    va_end(list);
}
```

- danger: `product(2,3,4)` won’t work, needs `product(2,3.0,4.0);`
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
Program with multiple files

- Library headers
  - Standard
  - User-defined

```c
#include <stdio.h>
#include "mypgm.h"

void main(void)
{
  myproc();
}
```

```c
#include <stdio.h>
#include "mypgm.h"

void myproc(void)
{
  mydata=2;
  . . . /* some code */
}
```

```c
hw.c
```

```c
mypgm.c
```

```c
void myproc(void);
int mydata;
```

```c
mypgm.h
```
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 NewQueue(void) {
    return q;
  }
  
  - Header file defines public data:
    ```c
    #ifndef QUEUE_C
    typedef struct queue_t *queue_t;
    #endif
    queue_t NewQueue(void);
    ```
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 func returning ptr to int*/
Function pointers

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

#include <stdio.h>

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:

<table>
<thead>
<tr>
<th>Category</th>
<th>Header File</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical Math Functions</td>
<td><code>&lt;math.h&gt;</code></td>
<td><code>-lm</code></td>
</tr>
<tr>
<td>Character Strings</td>
<td><code>&lt;string.h&gt;</code></td>
<td></td>
</tr>
<tr>
<td>Character Types</td>
<td><code>&lt;ctype.h&gt;</code></td>
<td></td>
</tr>
<tr>
<td>I/O</td>
<td><code>&lt;stdio.h&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
The math library

- `#include <math.h>
  - careful: `sqrt(5)` without header file may give wrong result!
- `gcc -o compute main.o f.o -lm`
- Uses normal mathematical notation:

<table>
<thead>
<tr>
<th>Math.sqrt(2)</th>
<th>sqrt(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.pow(x,5)</td>
<td>pow(x,5)</td>
</tr>
<tr>
<td>4*Math.pow(x,3)</td>
<td>4*pow(x,3)</td>
</tr>
</tbody>
</table>
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

<cctype.h> contains character classification functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum(ch)</td>
<td>alphanumeric</td>
<td>[a-zA-Z0-9]</td>
</tr>
<tr>
<td>isalpha(ch)</td>
<td>alphabetic</td>
<td>[a-zA-Z]</td>
</tr>
<tr>
<td>isdigit(ch)</td>
<td>digit</td>
<td>[0-9]</td>
</tr>
<tr>
<td>ispunct(ch)</td>
<td>punctuation</td>
<td>[~,!@#$%^&amp;...]</td>
</tr>
<tr>
<td>isspace(ch)</td>
<td>white space</td>
<td>[ \t\n]</td>
</tr>
<tr>
<td>isupper(ch)</td>
<td>upper-case</td>
<td>[A-Z]</td>
</tr>
<tr>
<td>islower(ch)</td>
<td>lower-case</td>
<td>[a-z]</td>
</tr>
</tbody>
</table>
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 \0
- A literal string ("a cat")
  - is automatically allocated memory space to contain it and the terminating \0
  - 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

char *makeBig(char *s) {
    s[0] = toupper(s[0]);
    return s;
}

makeBig(“a cat”);
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:
  
  ```c
  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);
  ```
Strings

- Can treat like arrays:

```c
char c;
char line[100];
for (i = 0; i < 100 && line[c]; i++) {
    if (isalpha(line[c])) ...
}
```
Copying strings

- Copying content vs. copying pointer to content
  - $s = t$ copies pointer — $s$ and $t$ now refer to the same memory location
  - `strcpy(s, t);` copies content of $t$ to $s$

```cpp
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\n", family, given);
    return 0;
}
```
string.h library

- Assumptions:
  - `#include <string.h>`
  - strings are NUL-terminated
  - all target arrays are large enough
- Operations:
  - `char *strcpy(char *dest, char *source)`
    - copies chars from source array into dest array up to NUL
  - `char *strncpy(char *dest, char *source, int num)`
    - copies chars; stops after num chars if no NUL before that; appends NUL
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 3rd, 4th, 5th, ... 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 3rd, 4th, 5th, ... arguments are formatted
    - return number of successful conversions
Formatted strings

- The format strings for `sscanf` 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><code>%c</code></td>
<td>matches a single character</td>
<td>char</td>
</tr>
<tr>
<td><code>%d</code></td>
<td>matches an integer in decimal</td>
<td>int</td>
</tr>
<tr>
<td><code>%f</code></td>
<td>matches a real number (ddd.dd)</td>
<td>float</td>
</tr>
<tr>
<td><code>%s</code></td>
<td>matches a string up to white space</td>
<td>char *</td>
</tr>
<tr>
<td><code>%[^c]</code></td>
<td>matches string up to next c char</td>
<td>char *</td>
</tr>
</tbody>
</table>
Formatted strings

- Formatting codes for sprintf
- Values normally right-justified; use negative field width to get left-justified

<table>
<thead>
<tr>
<th>Code</th>
<th>meaning</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%nc</td>
<td>char in field of n spaces</td>
<td>char</td>
</tr>
<tr>
<td>%%nd</td>
<td>integer in field of n spaces</td>
<td>int, long</td>
</tr>
<tr>
<td>%%nf</td>
<td>real number in width n, m decimals</td>
<td>float, double</td>
</tr>
<tr>
<td>%%mg</td>
<td>real number in width n, m digits of precision</td>
<td>float, double</td>
</tr>
<tr>
<td>%%ms</td>
<td>first m chars from string in width n</td>
<td>char *</td>
</tr>
</tbody>
</table>
Formatted strings - examples

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(“%10s %-10s”, t, s);
n = sscanf(nums, “%d %d %d”, &a, &b, &c);

printf(“%d flower%s”, 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:
  ```c
  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

  ```c
  int 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 putchar(int 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.