

Miscellaneous C Topics

Pointers and Arrays

Pointers and Arrays

Pointer

- Address of a variable in memory
- Allows us to indirectly access variables
 - in other words, we can talk about its *address* rather than its *value*

Array

- A list of values arranged sequentially in memory
- Example: a list of telephone numbers
- Expression `a[4]` refers to the 5th element of the array `a`

Passing Arrays as Arguments

C passes arrays by reference

- the address of the array (i.e., of the first element) is written to the function's activation record
- otherwise, would have to copy each element

```
main() {
    int numbers[MAX_NUMS];
    ...
    mean = Average(numbers);
    ...
}
int Average(int inputValues[MAX_NUMS]) {
    ...
    for (index = 0; index < MAX_NUMS; index++)
        sum = sum + inputValues[index];
    return (sum / MAX_NUMS);
}
```

This must be a constant, e.g.,
`#define MAX_NUMS 10`

A String is an Array of Characters

Allocate space for a string just like any other array:

```
char outputString[16];
```

Space for string must contain room for terminating zero.

Special syntax for initializing a string:

```
char outputString[16] = "Result = ";
```

...which is the same as:

```
outputString[0] = 'R';  
outputString[1] = 'e';  
outputString[2] = 's';  
...
```

Relationship between Arrays and Pointers

An array name is essentially a pointer
to the first element in the array

```
char word[10];  
char *cptr;  
  
cptr = word; /* points to word[0] */
```

Difference:

Can change the contents of cptr, as in

```
cptr = cptr + 1;
```

(The identifier "word" is not a variable.)

Common Pitfalls with Arrays in C

Overrun array limits

- There is no checking at run-time or compile-time to see whether reference is within array bounds.

```
int array[10];
int i;
for (i = 0; i <= 10; i++) array[i] = 0;
```

Declaration with variable size

- Size of array must be known at compile time.

```
void SomeFunction(int num_elements) {
    int temp[num_elements];
    ...
}
```

Pointer Arithmetic

Address calculations depend on size of elements

C does size calculations under the covers, depending on size of item being pointed to:

```
double x[10];
double *y = x;
*(y + 3) = 13;
```

← allocates 20 words (80 bytes)

← same as x[3] -- base address plus 6

Data Structures

Data Structures

A **data structure** is a particular organization of data in memory.

- We want to group related items together.
- We want to organize these data bundles in a way that is convenient to program and efficient to execute.

An **array** is one kind of data structure.

In this chapter, we look at two more:

struct – directly supported by C

linked list – built from `struct` and dynamic allocation

Structures in C

A **struct** is a mechanism for grouping together related data items of **different types**.

- Recall that an array groups items of a single type.

Example:

We want to represent an airborne aircraft:

```
char flightNum[7];
int altitude;
int longitude;
int latitude;
int heading;
double airSpeed;
```

We can use a **struct** to group these data together for each plane.

Defining a Struct

We first need to define a new type for the compiler and tell it what our struct looks like.

```
struct flightType {
    char flightNum[7]; /* max 6 characters */
    int altitude;      /* in meters */
    int longitude;     /* in tenths of degrees */
    int latitude;      /* in tenths of degrees */
    int heading;       /* in tenths of degrees */
    double airSpeed;   /* in km/hr */
};
```

This tells the compiler **how big** our struct is and how the different data items (“members”) are **laid out in memory**. But it does not allocate any memory.

Declaring and Using a Struct

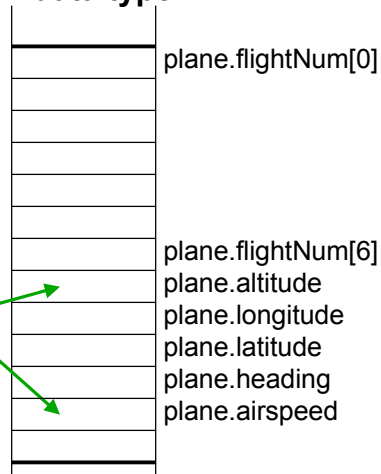
To allocate memory for a struct, we declare a variable using our new data type.

```
struct flightType plane;
```

Memory is allocated, and we can access individual members of this variable:

```
plane.airSpeed = 800.0;  
plane.altitude = 10000;
```

A struct's members are laid out in the order specified by the definition.



Defining and Declaring at Once

You can both define and declare a struct at the same time.

```
struct flightType {  
    char flightNum[7]; /* max 6 characters */  
    int altitude; /* in meters */  
    int longitude; /* in tenths of degrees */  
    int latitude; /* in tenths of degrees */  
    int heading; /* in tenths of degrees */  
    double airSpeed; /* in km/hr */  
} maverick;
```

And you can use the flightType name to declare other structs.

```
struct flightType iceMan;
```

typedef

C provides a way to define a data type by giving a new name to a predefined type.

Syntax:

```
typedef <type> <name>;
```

Examples:

```
typedef int Color;  
typedef struct flightType WeatherData;  
typedef struct ab_type {  
    int a;  
    double b;  
} ABGroup;
```

Using typedef

This gives us a way to make code more readable by giving application-specific names to types.

```
Color pixels[500];  
Flight plane1, plane2;
```

Typical practice:

Put typedef's into a header file, and use type names in main program. If the definition of Color/Flight changes, you might not need to change the code in your main program file.

Array of Structs

Can declare an array of structs:

```
Flight planes[100];
```

Each array element is a struct (7 words, in this case).

To access member of a particular element:

```
planes[34].altitude = 10000;
```

Because the [] and . operators are at the same precedence, and both associate left-to-right, this is the same as:

```
(planes[34]).altitude = 10000;
```

Pointer to Struct

We can declare and create a pointer to a struct:

```
Flight *planePtr;  
planePtr = &planes[34];
```

To access a member of the struct addressed by planePtr:

```
(*planePtr).altitude = 10000;
```

Because the . operator has higher precedence than *, this is **NOT** the same as:

```
*planePtr.altitude = 10000;
```

C provides special syntax for accessing a struct member through a pointer:

```
planePtr->altitude = 10000;
```

Passing Structs as Arguments

Unlike an array, a struct is always **passed by value** into a function.

- This means the struct members are copied to the function's activation record, and changes inside the function are not reflected in the calling routine's copy.

Most of the time, you'll want to pass a **pointer** to a struct.

```
int Collide(Flight *planeA, Flight *planeB)
{
    if (planeA->altitude == planeB->altitude) {
        ...
    }
    else
        return 0;
}
```

Dynamic Allocation

Suppose we want our weather program to handle a **variable number of planes** – as many as the user wants to enter.

- We can't allocate an array, because we don't know the maximum number of planes that might be required.
- Even if we do know the maximum number, it might be wasteful to allocate that much memory because most of the time only a few planes' worth of data is needed.

Solution:

Allocate storage for data dynamically, as needed.

malloc

The Standard C Library provides a function for allocating memory at run-time: **malloc**.

```
void *malloc(int numBytes);
```

It returns a **generic pointer** (`void*`) to a contiguous region of memory of the requested size (in bytes).

The bytes are allocated from a region in memory called the **heap**.

- The run-time system keeps track of chunks of memory from the heap that have been allocated.

Using malloc

To use `malloc`, we need to know how many bytes to allocate. The `sizeof` operator asks the compiler to calculate the size of a particular type.

```
planes = malloc(n * sizeof(Flight));
```

We also need to change the type of the return value to the proper kind of pointer – this is called “**casting**.”

```
planes =  
    (Flight*) malloc(n* sizeof(Flight));
```

Example

```
int airbornePlanes;
Flight *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

planes =
    (Flight*) malloc(sizeof(Flight) * airbornePlanes);
if (planes == NULL) {
    printf("Error in allocating the data array.\n");
    ...
}
planes[0].altitude = ...
```

If allocation fails,
malloc returns NULL.

Note: Can use array notation
or pointer notation.

free

Once the data is no longer needed,
it should be released back into the heap for later use.

This is done using the **free** function,
passing it the same address that was returned by malloc.

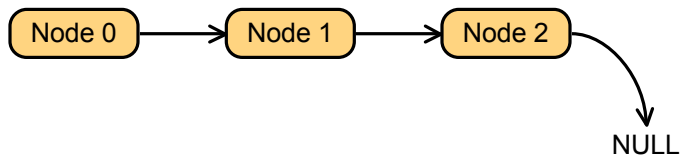
```
void free(void*);
```

If allocated data is not freed, the program might run out of
heap memory and be unable to continue.

The Linked List Data Structure

A **linked list** is an ordered collection of **nodes**, each of which contains some data, connected using **pointers**.

- Each node points to the next node in the list.
- The first node in the list is called the **head**.
- The last node in the list is called the **tail**.



Linked List vs. Array

A linked list can only be accessed **sequentially**.

To find the 5th element, for instance, you must start from the head and follow the links through four other nodes.

Advantages of linked list:

- Dynamic size
- Easy to add additional nodes as needed
- Easy to add or remove nodes from the middle of the list (just add or redirect links)

Advantage of array:

- Can easily and quickly access arbitrary elements

Example: Car Lot

Create an inventory database for a used car lot.

Support the following actions:

- **Search** the database for a particular vehicle.
- **Add** a new car to the database.
- **Delete** a car from the database.

The database must remain sorted by vehicle ID.

Since we don't know how many cars might be on the lot at one time, we choose a linked list representation.

Car data structure

Each car has the following characteristics:

vehicle ID, make, model, year, mileage, cost.

Because it's a linked list, we also need a pointer to the next node in the list:

```
typedef struct carType Car;

struct carType {
    int vehicleID;
    char make[20];
    char model[20];
    int year;
    int mileage;
    double cost;
    Car *next; /* ptr to next car in list */
}
```

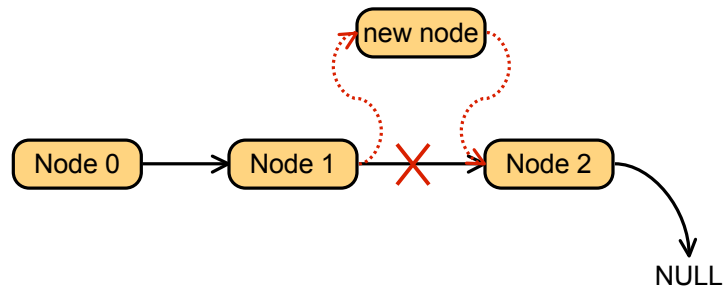
Scanning the List

Searching, adding, and deleting all require us to find a particular node in the list. We **scan** the list until we find a node whose ID is \geq the one we're looking for.

```
Car *ScanList(Car *head, int searchID)
{
    Car *previous, *current;
    previous = head;
    current = head->next;
    /* Traverse until ID  $\geq$  searchID */
    while ((current!=NULL)
           && (current->vehicleID < searchID)) {
        previous = current;
        current = current->next;
    }
    return previous;
}
```

Adding a Node

Create a new node with the proper info.
Find the node (if any) with a greater vehicleID.
“Splice” the new node into the list:



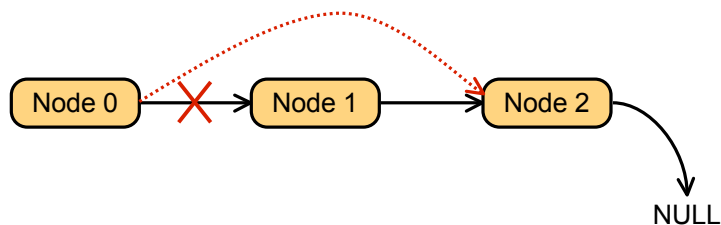
Excerpts from Code to Add a Node

```
newNode = (Car*) malloc(sizeof(Car));
/* initialize node with new car info */
...
prevNode = ScanList(head, newNode->vehicleID);
nextNode = prevNode->next;

if ((nextNode == NULL)
    || (nextNode->vehicleID != newNode->vehicleID))
    prevNode->next = newNode;
    newNode->next = nextNode;
}
else {
    printf("Car already exists in database.");
    free(newNode);
}
}
```

Deleting a Node

Find the node that **points to** the desired node.
Redirect that node's pointer to the next node (or NULL).
Free the deleted node's memory.



Excerpts from Code to Delete a Node

```
printf("Enter vehicle ID of car to delete:\n");
scanf("%d", &vehicleID);

prevNode = ScanList(head, vehicleID);
delNode = prevNode->next;

if ((delNode != NULL)
    && (delNode->vehicleID == vehicleID))
    prevNode->next = delNode->next;
    free(delNode);
}
else {
    printf("Vehicle not found in database.\n");
}
```

Building on Linked Lists

The linked list is a fundamental data structure.

- Dynamic
- Easy to add and delete nodes

The concepts described here will be helpful when learning about more elaborate data structures:

- Trees
- Hash Tables
- Directed Acyclic Graphs
- ...

I/O in C

Unix Files

A Unix **file** is a sequence of m bytes:

- $B_0, B_1, \dots, B_k, \dots, B_{m-1}$

All I/O devices are represented as files:

- `/dev/sda2` (`/usr` disk partition)
- `/dev/tty2` (terminal)

Even the kernel is represented as a file:

- `/dev/kmem` (kernel memory image)
- `/proc` (kernel data structures)

Unix File Types

Regular file

- Binary or text file.
- Unix does not know the difference!

Directory file

- A file that contains the names and locations of other files.

Character special and block special files

- Terminals (character special) and disks (block special)

FIFO (named pipe)

- A file type used for interprocess communication

Socket

- A file type used for network communication between processes

Unix I/O

The elegant mapping of files to devices allows kernel to export simple interface called Unix I/O.

Key Unix idea: All input and output is handled in a consistent and uniform way.

Basic Unix I/O operations (system calls):

- Opening and closing files
 - `open()` and `close()`
- Changing the *current file position* (seek)
 - `lseek`
- Reading and writing a file
 - `read()` and `write()`

Standard I/O Functions

The C standard library (`libc.a`) contains a collection of higher-level **standard I/O** functions

- Documented in Appendix B of K&R.

These functions are implemented on top of UNIX I/O

Most programmers do not need to use UNIX I/O

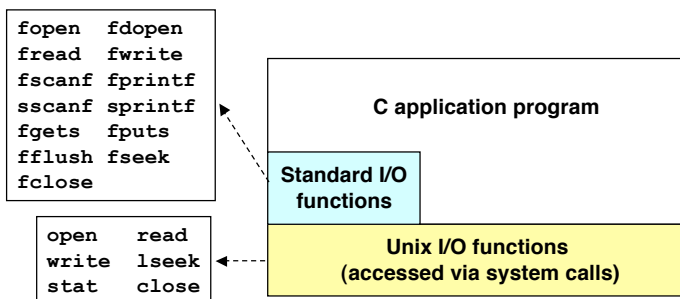
- We will revisit UNIX I/O later

Examples of standard I/O functions:

- Opening and closing files (`fopen` and `fclose`)
- Reading and writing bytes (`fread` and `fwrite`)
- Reading and writing text lines (`fgets` and `fputs`)
- Formatted reading and writing (`fscanf` and `fprintf`)

Unix I/O vs. Standard I/O

Standard I/O is implemented using low-level Unix I/O.



Standard I/O Streams

Standard I/O models open files as *streams*

- Abstraction for a file descriptor and a buffer in memory.

C programs begin life with three open streams (defined in `stdio.h`)

- `stdin` (standard input)
- `stdout` (standard output)
- `stderr` (standard error)

```
#include <stdio.h>
extern FILE *stdin; /* standard input (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

Basic I/O Functions

The standard I/O functions are declared in the `<stdio.h>` header file.

<i>Function</i>	<i>Description</i>
<code>putchar</code>	Displays an ASCII character to the screen.
<code>getchar</code>	Reads an ASCII character from the keyboard.
<code>printf</code>	Displays a formatted string,
<code>scanf</code>	Reads a formatted string.
<code>fopen</code>	Open/create a file for I/O.
<code>fprintf</code>	Writes a formatted string to a file.
<code>fscanf</code>	Reads a formatted string from a file.

Text Streams

All character-based I/O in C is performed on **text streams**.

A stream is a **sequence of ASCII characters**, such as:

- the sequence of ASCII characters printed to the monitor by a single program
- the sequence of ASCII characters entered by the user during a single program
- the sequence of ASCII characters in a single file

Characters are processed in the order in which they were added to the stream.

- E.g., a program sees input characters in the same order as the user typed them.

Standard input stream (keyboard) is called **stdin**.

Standard output stream (monitor) is called **stdout**.

Character I/O

putchar(c) Adds one ASCII character (c) to stdout.

getchar() Reads one ASCII character from stdin.

These functions deal with "raw" ASCII characters; no type conversion is performed.

```
char c = 'h';  
...  
putchar(c);  
putchar('h');  
putchar(104);
```

Each of these calls prints 'h' to the screen.

Buffered I/O

In many systems, characters are **buffered** in memory during an I/O operation.

- Conceptually, each I/O stream has its own buffer.

Keyboard input stream

- Characters are added to the buffer only when the newline character (i.e., the "Enter" key) is pressed.
- This allows user to correct input before confirming with Enter.

Output stream

- Characters are not flushed to the output device until the newline character is added.

Input Buffering

```
printf("Input character 1:\n");  
inChar1 = getchar();  
  
printf("Input character 2:\n");  
inChar2 = getchar();
```

- After seeing the first prompt and typing a single character, nothing happens.
- Expect to see the second prompt, but character not added to stdin until Enter is pressed.
- When Enter is pressed, newline is added to stream and is consumed by second `getchar()`, so `inChar2` is set to `'\n'`.

Output Buffering

```
putchar('a');  
/* generate some delay */  
for (i=0; i<DELAY; i++) sum += i;  
  
putchar('b');  
putchar('\n');
```

- User doesn't see any character output until after the delay.
- 'a' is added to the stream before the delay, but the stream is not flushed (displayed) until '\n' is added.

Formatted I/O

Printf and **scanf** allow conversion between ASCII representations and internal data types.

Format string contains text to be read/written, and **formatting characters** that describe how data is to be read/written.

%d	signed decimal integer
%f	signed decimal floating-point number
%x	unsigned hexadecimal number
%b	unsigned binary number
%c	ASCII character
%s	ASCII string

Special Character Literals

Certain characters cannot be easily represented by a single keystroke, because they

- correspond to whitespace (newline, tab, backspace, ...)
- are used as delimiters for other literals (quote, double quote, ...)

These are represented by the following sequences:

<code>\n</code>	newline
<code>\t</code>	tab
<code>\b</code>	backspace
<code>\\</code>	backslash
<code>\'</code>	single quote
<code>\"</code>	double quote
<code>\0nnn</code>	ASCII code <i>nnn</i> (in octal)
<code>\xnnn</code>	ASCII code <i>nnn</i> (in hex)

printf

Prints its first argument (format string) to stdout with all formatting characters replaced by the ASCII representation of the corresponding data argument.

```
int a = 100;
int b = 65;
char c = 'z';
char banner[10] = "Hola!";
double pi = 3.14159;

printf("The variable 'a' decimal: %d\n", a);
printf("The variable 'a' hex: %x\n", a);
printf("The variable 'a' binary: %b\n", a);
printf("'a' plus 'b' as character: %c\n", a+b);
printf("A char %c.\t A string %s\n A float %f\n",
       c, banner, pi);
```

Missing Data Arguments

What happens when you don't provide a data argument for every formatting character?

```
printf("The value of nothing is %d\n");
```

`%d` will convert and print whatever is on the stack in the position where it expects the first argument.

In other words, something will be printed, but it will be a garbage value as far as our program is concerned.

scanf

Reads ASCII characters from stdin, matching characters to its first argument (format string), converting character sequences according to any formatting characters, and storing the converted values to the addresses specified by its data pointer arguments.

```
char name[100];
int bMonth, bDay, bYear;
double gpa;

scanf("%s %d/%d/%d %lf",
      name, &bMonth, &bDay, &bYear, &gpa);
```

scanf Conversion

For each data conversion, `scanf` will skip whitespace characters and then read ASCII characters until it encounters the first character that should NOT be included in the converted value.

- `%d` Reads until first non-digit.
- `%x` Reads until first non-digit (in hex).
- `%s` Reads until first whitespace character.

Literals in format string must match literals in the input stream.

Data arguments must be pointers, because `scanf` stores the converted value to that memory address.

scanf Return Value

The `scanf` function returns an **integer**, which indicates the **number of successful conversions** performed.

- This lets the program check whether the input stream was in the proper format.

Example:

```
scanf("%s %d/%d/%d %lf",
      name, &bMonth, &bDay, &bYear, &gpa);
```

<i>Input Stream</i>	<i>Return Value</i>
Mudd 02/16/69 3.02	5
Muss 02 16 69 3.02	2



Doesn't match literal '/', so `scanf` quits after second conversion.

Bad scanf Arguments

Two problems with scanf data arguments

1. Not a pointer

```
int n = 0;
scanf("%d", n);
```

Will use the value of the argument as an address.

2. Missing data argument

```
scanf("%d");
```

Will get address from stack, where it expects to find first data argument.

If you're lucky, program will crash because of trying to modify a restricted memory location (e.g., location 0). Otherwise, your program will just modify an arbitrary memory location, which can cause very unpredictable behavior.

File I/O

A **file** is a sequence of bytes stored on some device.

- Allows us to process large amounts of data without having to type it in each time or read it all on the screen as it scrolls by.
- A file could contain ASCII characters (if it is a text file) or binary data (e.g. executables, image files, object files, etc.)
- Text files are read or written using `fscanf()` and `fprintf()`
- Binary files are read and written using `fread()` and `fwrite()`

Each file is associated with a stream.

- May be input stream or output stream (or both!).

The type of a stream is a "**file pointer**", declared as:

```
FILE *infile;
```

The `FILE` type is defined in `<stdio.h>`.

fopen

The `fopen` (pronounced "eff-open") function associates a physical file with a stream.

```
FILE *fopen(char* name, char* mode);
```

First argument: name

- The name of the physical file, or how to locate it on the storage device. This may be dependent on the underlying operating system.

Second argument: mode

- How the file will be used:
 - "r" -- read from the file
 - "w" -- write, starting at the beginning of the file
 - "a" -- write, starting at the end of the file (append)

fprintf and fscanf

Once a text file is opened, it can be read or written using `fscanf()` and `fprintf()`, respectively.

These are just like `scanf()` and `printf()`, except an additional argument specifies a file pointer.

```
fprintf(outfile, "The answer is %d\n", x);
```

```
fscanf(infile, "%s %d/%d/%d %lf",  
        name, &bMonth, &bDay, &bYear, &gpa);
```

fread and fwrite

Binary files are read and written using `fread()` and `fwrite()`

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE
             *stream)
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE
             *stream)
```

The function `fread()` reads `nmemb` objects, each `size` bytes long, from the stream pointed to by `stream`, storing them at the location given by `ptr`.

The function `fwrite()` writes `nmemb` objects, each `size` bytes long, to the stream pointed to by `stream`, obtaining them from the location given by `ptr`

Both functions advance the file position indicator for the stream by the number of bytes read or written. They return the number of objects read or written. If an error occurs, or the end-of-file is reached, the return value is a short object count (or zero).

Example showing using of `fread()` and `fwrite()` to read and write blocks of binary data

```
#include <stdio.h>
int main()
{
    int row, column;
    FILE *my_stream;
    int close_error;
    char my_filename[] = "my_numbers.dat";
    size_t object_size = sizeof(int);
    size_t object_count = 25;
    size_t op_return;
```

Example cont'd

```
int my_array[5][5] =
{
    2, 4, 6, 8, 10,
    12, 14, 16, 18, 20,
    22, 24, 26, 28, 30,
    32, 34, 36, 38, 40,
    42, 44, 46, 48, 50
};
printf ("Initial values of array:\n");
for (row = 0; row <= 4; row++) {
    for (column = 0; column <=4; column++)
        printf ("%d ", my_array[row][column]);
    printf ("\n");
}
```

Example cont'd

```
my_stream = fopen (my_filename, "w");
op_return = fwrite (&my_array, object_size, object_count, my_stream);
if (op_return != object_count)
    printf ("Error writing data to file.\n");
else
    printf ("Successfully wrote data to file.\n");

/* Close stream; skip error-checking for brevity of example */
fclose (my_stream);
printf ("Zeroing array...\n");
for (row = 0; row <= 4; row++)
    for (column = 0; column <=4; column++) {
        my_array[row][column] = 0;
        printf ("%d ", my_array[row][column]);
    }
    printf ("\n");
}
```

Example cont'd

```
printf ("Now reading data back in...\n");
my_stream = fopen (my_filename, "r");
op_return = fread (&my_array, object_size, object_count, my_stream);
if (op_return != object_count) {
    printf ("Error reading data from file.\n");
}
else
    printf ("Successfully read data from file.\n");
for (row = 0; row <= 4; row++)
    for (column = 0; column <=4; column++) {
        printf ("%d ", my_array[row][column]);
        printf ("\n");
    }
/* Close stream; skip error-checking for brevity of example */
fclose (my_stream);
return 0;
}
```

Debugging

Debugging with High Level Languages

Goals

- Examine and set values in memory
- Execute portions of program
- Stop execution when (and where) desired

Want debugging tools to operate on high-level language constructs

- Examine and set variables, not memory locations
- Trace and set breakpoints on statements and function calls, not instructions
- ...but also want access to low-level tools when needed

Types of Errors

Syntactic Errors

- Input code is not legal
- Caught by compiler (or other translation mechanism)

Semantic Errors

- Legal code, but not what programmer intended
- Not caught by compiler, because syntax is correct

Algorithmic Errors

- Problem with the logic of the program
- Program does what programmer intended, but it doesn't solve the right problem

Syntactic Errors

Common errors:

- missing semicolon or brace
- mis-spelled type in declaration

One mistake can cause an avalanche of errors

- because compiler can't recover and gets confused

```
main () {  
    int i  
    int j;  
    for (i = 0; i <= 10; i++) {  
        j = i * 7;  
        printf("%d x 7 = %d\n", i, j);  
    }  
}
```

← missing semicolon

Semantic Errors

Common Errors

- Missing braces to group statements together
- Confusing assignment with equality
- Wrong assumptions about operator precedence, associativity
- Wrong limits on for-loop counter
- Uninitialized variables

```
h  
main () {  
    int i  
    int j;  
    for (i = 0; i <= 10; i++)  
        j = i * 7;  
        printf("%d x 7 = %d\n", i, j);  
}
```

← missing braces,
so printf not part of if

Algorithmic Errors

Design is wrong,
so program does not solve the correct problem

Difficult to find

- Program does what we intended
- Problem might not show up until many runs of program

Maybe difficult to fix

- Have to redesign, may have large impact on program code

Classic example: Y2K bug

- only allow 2 digits for year, assuming 19__

Debugging Techniques

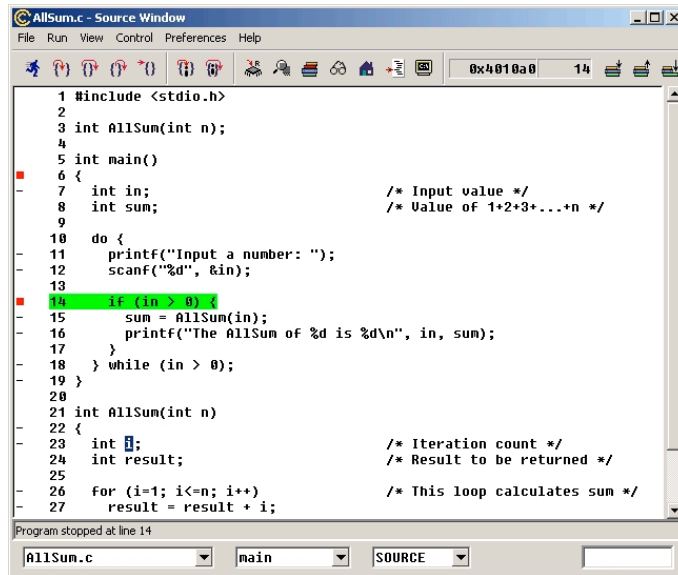
Ad-Hoc

- Insert printf statements to track control flow and values
- Code explicitly checks for values out of expected range, etc.
- Advantage:
 - No special debugging tools needed
- Disadvantages:
 - Requires intimate knowledge of code and expected values
 - Frequent re-compile and execute cycles
 - Inserted code can be buggy

Source-Level Debugger

- Examine and set variable values
- Tracing, breakpoints, single-stepping on source-code statements

Source-Level Debugger



```
1 #include <stdio.h>
2
3 int AllSum(int n);
4
5 int main()
6 {
7     int in;           /* Input value */
8     int sum;         /* Value of 1+2+3+...+n */
9
10    do {
11        printf("Input a number: ");
12        scanf("%d", &in);
13
14        if (in > 0) {
15            sum = AllSum(in);
16            printf("The AllSum of %d is %d\n", in, sum);
17        }
18    } while (in > 0);
19 }
20
21 int AllSum(int n)
22 {
23     int i;           /* Iteration count */
24     int result;     /* Result to be returned */
25
26     for (i=1; i<=n; i++) /* This loop calculates sum */
27         result = result + i;
```

main window
of Cygwin
version of gdb

Source-Level Debugging Techniques

Breakpoints

- Stop when a particular statement is reached
- Stop at entry or exit of a function
- **Conditional breakpoints:**
Stop if a variable is equal to a specific value, etc.
- **Watchpoints:**
Stop when a variable is set to a specific value

Single-Stepping

- Execute one statement at a time
- Step "into" or step "over" function calls
 - **Step into:** next statement is first inside function call
 - **Step over:** execute function without stopping
 - **Step out:** finish executing current function and stop on exit

Source-Level Debugging Techniques

Displaying Values

- Show value consistent with declared type of variable
- Dereference pointers (variables that hold addresses)
- Inspect parts of a data structure