

INTRODUCTION TO THE
C
PROGRAMMING LANGUAGE

Federico Ficarelli

CINECA ([f.ficarelli@cineca.it](mailto:f.ficarelli@ Cineca.it))

AGENDA

Agenda

- Overview
- A C program
- Basic Types and Operators
- Control Structures
- Derived Data Types
- Pointers and Memory
- Functions
- Dynamic Memory Management
- Strings
- The Standard Library
- I/O
- Notes on mixing FORTRAN and C
- Case Study: Linked Lists
- A bit of “hands-on”

OVERVIEW

The Early Days



- Designed in 1972
- Former design guidelines:
 - Allow straightforward compilation
 - Support for extremely tiny systems
 - Low level access to system resources
 - Allow efficient mapping to assembly language instructions
 - Portability

The main goal was to build up a convenient and portable alternative to assembly code for system programming.

Description of C

- General-purpose language
- Procedural (= functions + data)
- Relatively small, simple to learn, difficult to use
- Error checks ONLY at compile time
- NO error check at run time
- Cross-platform language, single-platform compilers (unlike Java)

The programmer knows **exactly** what he wants to do and how to use the language constructs to achieve that goal. The language lets the expert programmer express what they want in the minimum time.

These lectures have been produced with an heavy-modification of the Stanford CS Education Library Course: <http://cslibrary.stanford.edu>

Why C?

- Prior to C, two broad types of languages:
 - Applications languages
 - High-level
 - COBOL, etc.
 - Portable but inefficient
 - Systems languages
 - Low-level
 - Assembly
 - Efficient but not portable
- Goal of C: efficient *and* portable
- How: abstract **above** hardware, but not **far from it!**

C vs. Java and C++

- C is fast (in part) because there's so **little error-checking**
- No garbage collection
- No boolean* or string types
 - Booleans “implemented” as numbers
 - Strings are “just” arrays of characters
- Java is safe and elegant, but **slow**
- C++ is unsafe and fast, but **highly complex**
- C is **unsafe**, but **succinct** and **fast**

C is alive and kicking

- Due to its extreme minimalism, it gets older better than others.
- Still widely used (UNIX, scientific codes, drivers, intermediate code for modern languages, embedded devices, etc...)
- Still actively developed
- Base language for the vast majority of novel paradigms/platforms (e.g.: accelerators, manycores)

Standard specifications:

- K&R (1978)
- ANSI C89
- ANSI C90 (ISO/IEC 9899:1990)
- C99 (ISO/IEC 9899:1999)
- C11 (ISO/IEC 9899:2011)

Position May 2012	Position May 2011	Delta in Position	Programming Language	Ratings May 2012	Delta May 2011	Status
1	2	↑	C	17.346%	+1.18%	A
2	1	↓	Java	16.599%	-1.56%	A
3	3	=	C++	9.825%	+0.68%	A
4	6	↑↑	Objective-C	8.309%	+3.30%	A
5	4	↓	C#	6.823%	-0.72%	A
6	5	↓	PHP	5.711%	-0.80%	A

Source: TIOBE Community Index

A C PROGRAM

C Syntax and Hello World

Can your program have more than one .c file?

`#include` inserts another file. “.h” files are called “header” files. They contain stuff needed to interface to libraries and code in other “.c” files.

This is a comment. The compiler ignores this.

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```

The `main()` function is always where your program starts running.

Blocks of code (“lexical scopes”) are marked by `{ ... }`

Return ‘0’ from this function

Print out a message. ‘\n’ means “new line”.

Hello, World

- In C, a program has ≥ 1 functions
- `main()` is the entry point
 - Contains the instructions performed when program is run
- The body of a function is demarcated by braces (“curly brackets”):

```
{ }
```

- Each instruction line ends with “;”
- Inside, instructions to be performed when the function is *called*
- `main()` called on program start-up

- **First program: Hello, World**

```
#include <stdio.h>
main() {
    printf("Hello, world.\n");
}
```

main ()

- The execution of a C program **begins with function named main()**.
- **All of the files and libraries** for the C program are compiled together to build a single program file.
- That file must contain **exactly one** main() function which the operating system uses as the starting point for the program.
- Main() returns an int which, by convention, is 0 if the program completed successfully and **non-zero** if the program exited due to some error condition.

success == 0

fault != 0

Is a de-facto standard for error-related return values!

Multiple files

For a program of any size, it's convenient to separate the functions into several separate files.

To allow the functions in separate files to cooperate, and yet allow the compiler to work on the files independently, C programs typically depend on two features:

- **Prototyping**
- **Preprocessing**

Prototypes

A "prototype" for a function gives its name and arguments but not its body. In order for a caller, in any file, to use a function, the caller must have seen the prototype for that function.

```
int Twice(int num);  
void Swap(int* a, int* b);
```

All functions requires a prototype, except "static" functions:

```
static int foo()
```

a static function is visible **only** in the same file where it is defined,
Do NOT confuse with **static variables!!!**

Preprocessor

The preprocessing step happens to the C source before it is fed to the compiler.

indicates command to C **Preprocessor**

The preprocessor language consists in:

- Directives to be executed
- Macros to be expanded

Primary functions:

- Inclusion of header files
- Macro expansion
- Conditional compilation
- Line control
- Diagnostic

It's a **stand-alone text processor!**
It's widely used to preprocess other languages.

#include

The "#include" directive brings in (paste) text from different files during compilation.

#include is very **unintelligent and unstructured**

Usually a header file (.h) acts as an interface describing the publicly available functions in the .c file, It contains declarations and macro definitions

```
#include "foo.h" // refers to a "user" foo.h file --  
                // in the originating directory for the compile
```

```
#include <foo.h> // refers to a "system" foo.h file --  
                // in the compiler's directory somewhere or  
                // specified by -I option
```

foo.c vs foo.h

for a file named "foo.c" containing a bunch of functions:

- A separate file named foo.h will contain the prototypes for the functions in foo.c which clients may want to call.
- Near the top of foo.c will be the following line which ensures that the function definitions in foo.c see the prototypes in foo.h which ensures the "**prototype before definition**" rule above.

```
#include "foo.h" // show the contents of "foo.h"  
                // to the compiler at this point
```

- Any xxx.c file which wishes to call a function defined in foo.c **must include** the following line to see the prototypes, ensuring the "**clients must see prototypes**" rule above.

```
#include "foo.h"
```

Macros and #define

A macro is a fragment of code with a **name**.

Whenever that name is used, it is replaced by the content of the macro.

Macros are created with the "#define" directive

Two classes of macros:

1. *Object-like* macros, resembles data objects:

```
#define BUFFER_SIZE 1024  
int myarray[BUFFER_SIZE];
```

Is replaced by:

```
int myarray[1024];
```

2. *Function-like* macros, resemble function calls:

```
#define min(X, Y) ((X)<(Y) ? (X):(Y))  
x=min(a,b);    => x=((a)<(b) ? (a):(b));  
z=min(a+28, *p);    => z=((a+28)<(*p) ? (a+28):(*p));
```

Once-Only headers

The same header file could be included more than once

The compiler process it more than once → **COMPILING ERROR!!!**

Solution: *Wrapper #ifndef*

```
/* File foo.h */  
#ifndef FILE_FOO_SEEN  
#define FILE_FOO_SEEN
```

Body of the file

```
#endif /* !FILE_FOO_SEEN */
```

FILE_FOO_SEEN is called "*header guard*"

For system header files guard, macro starts with "__" to avoid conflicts

Predefined macros

Some predefined macros are available

`__FILE__` : current source file name (string)

`__LINE__` : current line number (integer)

`__DATE__` : current date (string)

`__TIME__` : current time (string)

Conditionals

Conditionals are directive which make the preprocessor **include** **or not** chunks of code. This in order to:

- Adapt the compiled code to the system.
- Support different versions of the same code (e.g for production run or for debug).
- Exclude some code at compiling time keeping it, as a comment, in the source.
- Useful to avoid the compilation of unused code.

Conditionals directives

`# if expression (true/false 1/0), # if defined MACRO`

Expression can include integer constant, arithmetic and logical operators, macros. They can be combined (differently from `ifdef`).

Examples:

➤ `#if defined (__SP5__) || defined (__CLX__)`

➤ `#if defined BUFSIZE && BUFSIZE >= 1024`

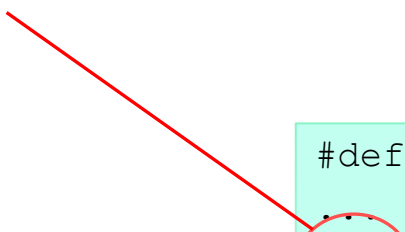
```
#ifndef DEBUG_MODE //defined in the code or at compile time (-D option))
    // debug code
#elif defined(HARDCORE_CODE) && defined(KEEP_PRINTS)
    // production code with log prints
#else
    // naive code
#endif
```

Preprocessor: boolean value test

The `#if` test can be used at compile-time to look at those symbols and turn on and off which lines the compiler uses

Warning:
this test is about boolean value,
not definition!

```
#define NAIVE_DGEMM 1
...
#if NAIVE_DGEMM
mydgemm(...);
#else
mkl_dgemm(...);
#endif
```



Preprocessor: definition test

A useful different version is `ifdef/ifndef` (old school) or `defined/!defined` (C99 preprocessor operators)

```
#ifdef USE_FAST_DGEMM
    #ifdef HAVE_MKL
        mkl_dgemm(...);
    #else
        fast_dgemm(...);
    #endif
```

```
#elif FALLBACK_VANILLA_BLAS
    dgemm(...);
#else
    my_kickass_dgemm(...);
#endif
```

```
#if defined(USE_FAST_DGEMM) && defined(HAVE_MKL)
    mkl_dgemm(...);
#endif
#if defined(USE_FAST_DGEMM) && !defined(HAVE_MKL)
    fast_dgemm(...);
#endif
#elif FALLBACK_VANILLA_BLAS
    dgemm(...);
#else
    my_kickass_dgemm(...);
#endif
```

Errors and Warning

The directive

```
#error
```

causes the preprocessor to report a fatal error and exit

```
#ifdef __vax__
```

```
#error "This code does not work on Vax architectures"
```

```
#endif
```

The directive

```
#warning
```

makes the preprocessor report a warning without exiting

Pragmas

Pragmas provide to the compiler additional information. Three forms of pragmas are defined by the C standard. Other can be implemented specific to the compilers.

1. `#pragma GCC dependency`

Compares the dates of the current file and another file

```
#pragma GCC dependency "parse.c"
```

2. `#pragma GCC poison`

Used to remove a certain identifier from the source. E.g.

```
#pragma GCC poison printf
printf("Hello world\n");
```

Cause a error

3. `#pragma GCC system_header`

Cause the rest of the code in the current file to be treated as a system header (special treatment...)

VARIABLES, BASIC TYPES AND OPERATORS

What is “Memory”?

Memory is like a big table of numbered slots where bytes can be stored.

The number of a slot is its **Address**.
One byte **Value** can be stored in each slot.

Some “logical” data values span more than one slot, like the character string “Hello\n”

A **Type** gives a logical meaning to a span of memory. Some simple types are:

<code>char</code>	a single character (1 slot)
<code>char [10]</code>	an array of 10 characters
<code>int</code>	signed 4 byte integer
<code>float</code>	4 byte floating point

Addr	Value
0	
1	
2	
3	
4	'H' (72)
5	'e' (101)
6	'l' (108)
7	'l' (108)
8	'o' (111)
9	'\n' (10)
10	'\0' (0)
11	
12	

Variables

A variable corresponds to an area of memory which can store a value of the given type

The variable must be **declared** and **defined**: a variable **declaration names a resource** letting the compiler know about its existence **somewhere**); a variable **definition** reserves all the **needed resources** (memory) to hold its value.

Names in C are **case sensitive** so "x" and "X" refer to different variables. Names can contain digits and underscores (_), but cannot begin with a digit.

Definitions and Declarations

- Var definition+declaration pattern:

data-type var-name;

```
int low;
```

```
int low, upr;    // combined
```

```
int step = 20;   // with init */
```

- Var declaration example:

```
extern int low;
```

- Var names:
 - Like all else, case-sensitive
 - Usually: all lower-case*
 - Can't begin with a number

What is a Variable?

A Variable defines a place in memory where you store a Value of a certain Type.

You first Define a variable by giving it a name and specifying the type, and optionally an initial value

```
char x;  
char y='e';
```

Initial value of x is undefined

Initial value

Name

Type is single character (char)

The compiler puts them somewhere in memory.

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	'e' (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

Multi-byte Variables

Different types consume different amounts of memory. Most architectures store data on “word boundaries”, or even multiples of the size of a primitive data type (int, char)

```
char x;  
char y='e';  
int z = 0x01020304;
```

0x means the constant is written in hex

padding

An int consumes 4 bytes

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	'e' (101)
	6	
	7	
z	8	4
	9	3
	10	2
	11	1
	12	

Primitive Data Types

Type	Meaning	Example
char	Typographical symbol (replaced by 1-byte integer, in ASCII). the "smallest addressable unit" for the machine: each byte in memory has its own address	'A', 65
int	Integer	3
float	Non-whole number	12345.6700
double	Double-precision number	12345.678930499

The char type

A char constant is written with single quotes (') like 'A' or 'z'. The char constant 'A' is really just a synonym for the ordinary integer value 65 which is the ASCII value for uppercase 'A'. There are special case char constants, such as '\t' for tab, for characters which are not convenient to type on a keyboard.

- 'A' uppercase 'A' character
- '\n' newline character
- '\t' tab character
- '\0' the "null" character -- integer value 0 (different from the char digit '0' -> 48)

The Integer type

- short int
- long int
 - “int” usually **omitted**

- Rule (in *bits*):

$|char| = 8 \leq |short| \leq |int| \leq |long| \leq |long\ long|$

usually: 16, 32, 32, 64

ancient: 16, 16, 32, 32

Notice possible portability problems!!!

- Integer can be signed: +/-
- Or unsigned: non-negative
- Unsigned char: 0...255, 8 bits $\rightarrow 2^8 = 256$ possible values
- 1 bit can be used for sign (“sign bit”)
 - 7 bits left for magnitude $2^7 = 128$
- Non-neg vals: 128 poss vals
 - $0..2^7-1 = 0..127$

Floats and Doubles

- float - Single precision floating point number typical size: **32 bits**
- double - Double precision floating point number typical size: **64 bits**
- long double - Possibly even bigger floating point number

Constants in the source code such as 3.14 have a **default double type** unless they are suffixed with an **'f'** (float – 3.14f) or **'l'** (long double - 3.14l).

The choice between float and double should be driven by the **precision needs** of the algorithm.

Floats representation (IEEE standard)

Single precision numbers in a 32-bit machine

The bit pattern $b_1b_2b_3\dots b_9b_{10}b_{11}\dots b_{32}$ of a word in a 32-bit machine represents the real number

$$(-1)^s \times 2^{e-127} \times (0.f)_2$$

where $s = b_1$, $e = (b_2\dots b_9)_2$, and $f = b_{10}b_{11}\dots b_{32}$.

sign bit	biased exponent	fraction from normalized mantissa
1 bit	8 bits	23 bit
s	e	f

IEEE 754
Language agnostic standard

Note that only the fraction from the normalized mantissa is stored and so there is a hidden bit and the mantissa is actually represented by 24 binary digits.

Double precision numbers in a 32-bit machine

The bit pattern $b_1b_2b_3\dots b_{12}b_{13}b_{14}\dots b_{64}$ of two words in a 32-bit machine represents the real number

$$(-1)^s \times 2^{e-1023} \times (0.f)_2$$

where $s = b_1$, $e = (b_2\dots b_{12})_2$, and $f = b_{13}b_{14}\dots b_{64}$.

sign bit	biased exponent	fraction from normalized mantissa
1 bit	11 bits	52 bit
s	e	f

Little and Big Endian

"Little Endian" means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. (The little end comes first.) For example, a 4 byte Long Int

Byte3 Byte2 Byte1 Byte0

will be arranged in memory as follows:

Base Address+0	Byte0
Base Address+1	Byte1
Base Address+2	Byte2
Base Address+3	Byte3

Intel processors (those used in PC's) use "Little Endian" byte order.

"Big Endian" means that the high-order byte of the number is stored in memory at the lowest address, and the low-order byte at the highest address. (The big end comes first.) Our integer would then be stored as:

Base Address+0	Byte3
Base Address+1	Byte2
Base Address+2	Byte1
Base Address+3	Byte0

Booleans

The typical implementation of booleans: **int** is used instead. **The language treats integer 0 as false and all non-zero values as true.** So the statement...

```
i = 0;  
while (i - 10) {  
    ...  
}
```

will execute until the variable *i* takes on the value 10 at which time the expression $(i - 10)$ will become false (i.e. 0).

The C99 standard introduced the **bool** type but it's still a placeholder for int!

Type combination and Promotion

- The integral types may be mixed together in arithmetic expressions since they are all basically just integers with variation in their width. For example, char and int can be combined in arithmetic expressions such as ('b' + 5).
- the compiler "**promotes**" the smaller type (char) to be the same size **as the larger type** (int) before combining the values.
- Promotions **do not lose information**

Int Overflow

- try to compute the expression $(k * 1024)$ where k is a 16 bits int. Since k and 1024 were both int, there is no promotion. **For values of $k \geq 32$, the product is too big to fit in the 16 bit int (max 32768) resulting in an overflow.** The compiler can do whatever he wants in overflow situations -- typically the high order bits just vanish. One way to fix the code was to rewrite it as **$(k * 1024L)$** -- the long constant forced the promotion of the int.

Truncation

Truncation moves a value from a type to a smaller type. The compiler just **drops the extra bits**.

It may or may not generate a compile time warning of the **loss of information**.

Assigning from an integer to a smaller integer (e.g.. long to int, or int to char) drops the most significant bits.

```
char ch;  
int i;  
i = 321;  
ch = i; // truncation of an int value to fit in a char  
// ch is now 65  
321 -> 101000001 trunc -> 01000001 -> 65
```

The assignment will drop the upper bits of the int 321. The lower 8 bits of the number 321 represents the number 65 (321 - 256 or 101000001-100000000).

Truncation (cont.ed)

The assignment of a floating point type to an integer type will drop the fractional part of the number:

```
double pi;  
int i;  
pi = 3.14159;  
i = pi; // truncation of a double to fit in an int  
// i is now 3
```

Implicit Casting

Implicit Casting (automatic transformation) works in a way that a variable (operand) of data type that is smaller in length than data type of second variable (operand), transforms internally to variable of data type with longer number length.

For example:

short int -> int -> unsigned int -> long int -> unsigned long int -> float -> double -> long double

Example:

```
int a;
```

```
unsigned long b;
```

```
float f, g;
```

```
double d;
```

```
g = a + f; // a transforms to float
```

```
d = a + b;
```

```
// a transform to unsigned long, adding
```

```
// is produced in unsigned long domain and then
```

```
// the result type unsigned long is transformed
```

```
// to double
```

Explicit Casting

Says explicitly to convert types

General form: put desired type name in parentheses, prepend to expr:

- (new-type) expr;
- (float) i;

```
{
int score;
...// suppose score gets set in the range 0..20 somehow

score = (score / 20) * 100;          // NO -- score/20 truncates to 0
...
}
```

Score will almost always be set to 0 for this code because the integer division in the expression (score/20) will be 0 for every value of score less than 20. The fix is to force the quotient to be computed as a floating point number...

```
score = ((double)score / 20) * 100; // OK -- floating point division from cast
score = (score / 20.0) * 100; // OK -- floating point division from 20.0
score = (int)(score / 20.0) * 100; // NO -- the (int) truncates the floating
                                   // quotient back to 0
```

Comments

Comments in C are enclosed by as:

```
/* .. comments .. */
```

The comment may cross multiple lines.

C++ introduced a form of comment started by two slashes and extending to the end of the line:

```
// comment until the line end
```

The // comment form is so handy that the ISO committee introduced it as standard in C99.

Assignment Operator =

The assignment operator is the single equals sign (=).

```
i = 6;
```

```
i = i + 1;
```

The assignment operator copies the value from its right hand side to the variable on its left hand side. The assignment also acts as an expression which returns the newly assigned value.

```
y = (x = 2 * x); // double x, and also put x's new value in y
```

Binary arithmetic & bool operations

- +
- -
- *
- /
- % - mod/remainder $5\%3 == 2$
- && - boolean AND
- || - boolean OR
- ! - boolean NOT

Unary Increment Operators

var++ increment "post" variant
++*var* increment "pre" variant
var-- decrement "post" variant
--*var* decrement "pre" variant

```
int i = 42;  
i++; // increment on i  
// i is now 43  
i--; // decrement on i  
// i is now 42
```

```
int i = 42;  
int j;  
j = (i++ + 10);  
// i is now 43  
// j is now 52 (NOT 53)  
j = (++i + 10)  
// i is now 44  
// j is now 54
```

Other shortcut operations

Generally, shortcut ops for all standard binary ops:

`a += 2;` \Leftrightarrow `a = a + 2;`

`a *= 2;`

`a /= 2;`

etc.

Relational Operators

These operate on integer or floating point values and return a 0 or 1 boolean value.

- == Equal
- != Not Equal
- > Greater Than
- < Less Than
- >= Greater or Equal
- <= Less or Equal

To see if x equals 3, write:

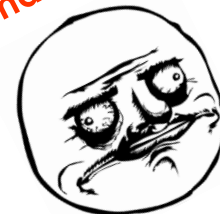
```
if (x == 3) ...
```

Common error:

```
if (x = 3) ... ALWAYS TRUE!!!!!!
```

does not test if x is 3. This sets x to the value 3...

*...try to Google «history's worst software bugs»
and have fun!*



Pay attention to boolean expressions!

- During the evaluation of an usual arithmetic expression, operands are **exhaustively** evaluated in an **unspecified order**
- All boolean operators use the **short circuit evaluation**:
 - The boolean operators are **SYNCHRONIZATION POINTS** and follow the so called **SHORT CIRCUIT SEMANTICS**:
 - The evaluation moves from left to right and stops as soon as the result of the boolean expression is known

```
int a(void), b(void), c(void);  
if( a() && b() && c() ) {  
    ...  
}  
else{  
    ...  
}
```

The evaluation stops on the first false!



- FORTRAN's operators behaviour is **eager only!**

Bitwise operations

Manipulate numbers as bitstrings, not as numbers. Safe for unsigned integer types **only!** (unsigned int, unsigned char, etc.)

~ complement

| bitwise or

^ xor

& bitwise and

<< shift left

>> shift right

NOT, OR, XOR, AND

A **bitwise NOT** or **complement**, is a unary operation which performs logical negation on each bit, forming the complement. Digits which were 0 become 1, and vice versa. For example:

NOT 0111 = 1000

A **bitwise OR** takes two bit patterns of equal length, and produces another one of the same length by matching up corresponding bits and performing the logical OR operation on each pair of corresponding bits. In each pair, **the result is 1 if the first bit is 1 OR the second bit is 1 (or both)**, and otherwise the result is 0. For example:

0101 OR 0011 = 0111

A **bitwise exclusive OR** takes two bit patterns of equal length and performs the logical XOR operation on each pair of corresponding bits. **The result in each position is 1 if the two bits are different, and 0 if they are the same.** For example:

0101 XOR 0011 = 0110

A **bitwise AND** takes two binary representations of equal length and performs the logical AND operation on each pair of corresponding bits. In each pair, **the result is 1 if the first bit is 1 AND the second bit is 1. Otherwise, the result is 0.** For example:

0101 AND 0011 = 0001

Left and Right Shifts << , >>

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted.

The << (bitwise left shift) operator indicates the bits are to be shifted to the left. The >> (bitwise right shift) operator indicates the bits are to be shifted to the right.

The << operator fills vacated bits with zeros.

If `l_op` is:

```
00000000000000000000000111110110011
```

The expression `l_op << 3` yields:

```
0000000000000000000000111110110011000
```

The >> operator fills vacated bits with the sign bit of the unshifted value:

`l_op` (= negative number):

```
1111111111111111111111111111111100111
```

Vacated bits are filled with ones, and the expression `l_op >> 3` yields:

```
11111111111111111111111111111111100
```

Operator precedence and associativity

	Operators	Associativity
↑ Precedence	() [] -> .	-> left to right
	! ~ ++ -- +(unary) -(unary) & *(deref) (cast) sizeof	<- right to left
	* / %	-> left to right
	+ -	-> left to right
	<< >>	-> left to right
	< <= > >=	-> left to right
	== !=	-> left to right
	&	-> left to right
	^	-> left to right
		-> left to right
	&&	-> left to right
		-> left to right
	?:	<- right to left
	= += -= *= /= %= &= ^= = <<= >>=	<- right to left
	,	-> left to right

If you want to chill out and avoid mistakes, remember:

1. Arithmetic operators behave naturally;
2. Don't be afraid to put extra parantheses to make associativity explicit

Avoid obfuscated statements!

```
for(int i='-''-''-'; i < 5; i += '/'/'/'/'/')
```


CONTROL STRUCTURES

If Statement

Both an if and an if-else are available in C. The `<expression>` can be any valid expression. The parentheses around the expression are required, even if it is just a single variable.

```
if (<expression>) <statement>
```

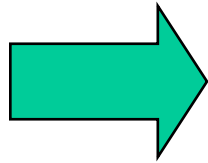
```
if (<expression>) {  
<statement>  
<statement>  
}
```

```
if (<expression>) {  
<statement>  
}  
else if (<expression>) {  
<statement>  
}  
else {  
<statement>  
}
```

Conditional Expression (Ternary Operator)

This is an expression, not a statement, so it represents a value:

```
if (x < y) {  
min = x;  
}  
else {  
min = y;  
}
```



```
min = (x < y) ? x : y;
```

Switch Statement

The switch statement is a sort of specialized form of if used to efficiently separate different blocks of code based on the value of an integer.

```
switch (<expression>) {  
case <const-expression-1>:  
<statement>  
break;  
  
case <const-expression-2>:  
<statement>  
break;  
  
case <const-expression-3>: // here we combine case 3 and 4  
case <const-expression-4>:  
<statement>  
break;  
  
default: // optional  
<statement>  
}
```

Switch Statement (cont.ed)

Once execution has jumped to a particular case, the program will keep running through all the cases **from that point down**

The explicit break statements are necessary to exit the switch. Omitting the break statements is a common error

```
int main()
{
    const char c = '2';
    int n;
    switch(c)
    {
        case '0': n=0; break;
        case '1': n=1; break;
        case '2': n=2; break;
        case '3': n=3; break;
        case '4': n=4; break;
        case '5': n=5; break;
        case '6': n=6; break;
        case '7': n=7; break;
        case '8': n=8; break;
        case '9': n=9; break;
        default: error();
    }
}
```

While Statements

The while loop evaluates the test expression before every loop, so it can execute zero times if the condition is initially false. It requires the parenthesis like the if.

```
while (<expression>) {  
<statement>  
}
```

Like a while, but with the test condition at the bottom of the loop. The loop body will always execute at least once.

```
do {  
<statement>  
} while (<expression>)
```

For Loop

The for loop in C is the most general looping construct. The loop header contains three parts: **an initialization, a continuation condition, and an action.**

```
for (<initialization>; <continuation>; <action>) {  
<statement>  
}
```

```
for (i = 0; i < 10; i++) {  
<statement>  
}
```

Each of the three parts of the for loop can be made up of **multiple expressions** separated by commas. Expressions separated by commas are executed **in order**, left to right

Break Statement

The `break` statement will move control **outside** a loop or switch statement

```
while (<expression>) {  
  <statement>  
  <statement>  
  if (<condition which can only be evaluated here>)  
    break;  
  <statement>  
  <statement>  
}  
// control jumps down here on the break
```


Continue Statement

The continue statement causes control to jump to the **bottom** of the loop (not outside), effectively skipping over any code below the continue

```
while (<expression>) {  
    ...  
    if (<condition>)  
        continue;  
    ...  
    ...  
    // control jumps here on the continue  
}
```

DERIVED DATA TYPES

Arrays

One-dimensional arrays are declared and accessed as:

```
int scores[100];  
scores[0] = 13; // set first element  
scores[99] = 42; // set last element
```

It's a very **common error** to try to refer to non-existent `scores[100]` element. C **does not do any run time or compile time bounds checking** in arrays. At run time the code will just access or corrupt whatever memory it happens to hit and crash or misbehave in some **unpredictable** way **thereafter**

NOTICE NUMBERING FROM 0 TO N-1

Multidimensional Arrays

Two (N)-dimensional arrays are declared and accessed as:

```
int board [10][10];  
board[0][0] = 13;  
board[9][9] = 13;
```

The implementation of the array stores all the elements in a single contiguous block of memory. In memory, the array is arranged with the elements of the **rightmost** index next to each other. In other words, `board[1][8]` comes right before `board[1][9]` in memory.

ROW MAJOR ORDERING

Struct

```
struct fraction {  
int numerator;  
int denominator;  
}; // Don't forget this semicolon!
```

This declaration introduces the type struct fraction (both words are required) as a **new type**. C uses the period (.) to access the fields in a record. You can copy two records of the same type using a single assignment statement, **however == does not work on structs** (error at compiling time).

```
struct fraction f1, f2; // defines two fractions  
f1.numerator = 22;  
f1.denominator = 7;  
f2 = f1; // this copies the whole struct
```

Arrays of Structs

```
struct fraction {  
    int numerator;  
    int denominator;  
};  
  
struct fraction numbers[1000];  
numbers[0].numerator = 22; /*set the 0 struct fraction */  
numbers[0].denominator = 7;
```

TypeDef

A typedef statement introduces a **shorthand name** for a type.

The syntax is:

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

```
typedef struct fraction Fraction; // From now on the type  
// identifier "Fraction" refers to "struct fraction"
```

```
typedef struct treeNode* Tree;  
struct treeNode {  
int data;  
Tree smaller, larger; // equivalently, this line could say  
}; // "struct treeNode *smaller, *larger"
```

POINTERS

Pointers

- A pointer is a variable which stores the address of another variable
- a pointer stores a **reference** to another value. The variable the pointer refers to is sometimes known as its "pointee"

A pointer type in C is just the pointee type followed by a asterisk (*)...

```
int*           // type: pointer to int
float*        // type: pointer to float
float**       // type: pointer to pointer to float
struct foo*   // type: pointer to struct foo
struct foo**  // type: pointer to pointer to
              // struct foo
float***      // type: just let the rule scale!
```

Make a pointer point a variable

The “&” operator is one of the ways that pointers are set to point to things.

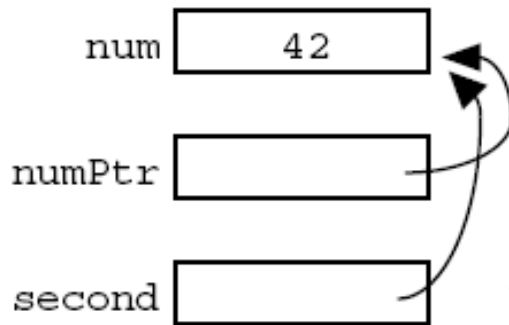
The & operator computes a pointer to the argument to its right.

```
void foo() {  
int* p;      // p is a pointer to an integer  
int i;      // i is an integer  
p = &i;     // Set p to point to i  
*p = 13;    // Change what p points to -- in this case i  
                                     -- to 13  
// At this point i is 13. So is *p. In fact *p is i.  
}
```

ALIASING

Pointer Assignment

The assignment operation (**=**) between two pointers makes them point to the same pointee.



A second pointer `ptr` initialized with the assignment `second = numPtr;`. This causes `second` to refer to the same pointee as `numPtr`.

After assignment, the `==` test comparing the two pointers will return true

Uninitialized Pointers

Declaring a pointer allocates space for the pointer itself, **but it does not allocate space for the pointee. The pointer must be set to point to something** before you can use it.

every pointer starts out with a bad value!!!

There are three things which must be done for a pointer/pointee relationship to work...

- 1. The pointer must be defined**
- 2. The pointee must be defined**
- 3. The pointer (1) must be initialized so that it points to the pointee (2)**

The most common pointer related error of all time is the following: define the pointer (step 1). Forget step 2 and/or 3. Start using the pointer as if it has been setup to point to something. Code with this error **frequently compiles fine, but the runtime results are disastrous.** Unfortunately the pointer **does not point anywhere good** unless (2) and (3) are done,

The NULL pointer

The constant NULL is a special pointer value which encodes the idea of "**points to nothing**".

It is a **runtime error** to dereference a NULL pointer.

The C language uses the symbol NULL for this purpose. NULL is equal to the integer constant 0, so NULL can play the role of a boolean false.

Dereferencing a pointer

the unary * to the left of a pointer dereferences it to retrieve the value it points to:

```
int var;  
int * var_pointer;  
  
var = 385;  
var_pointer = &var;  
  
printf("---- %d\n", *var_pointer);
```

It will print:

```
---- 385
```

```
*ptr_a == *ptr_b  compare pointee values  
ptr_a == ptr_b    compare addresses
```

Dereferencing a pointer to a struct

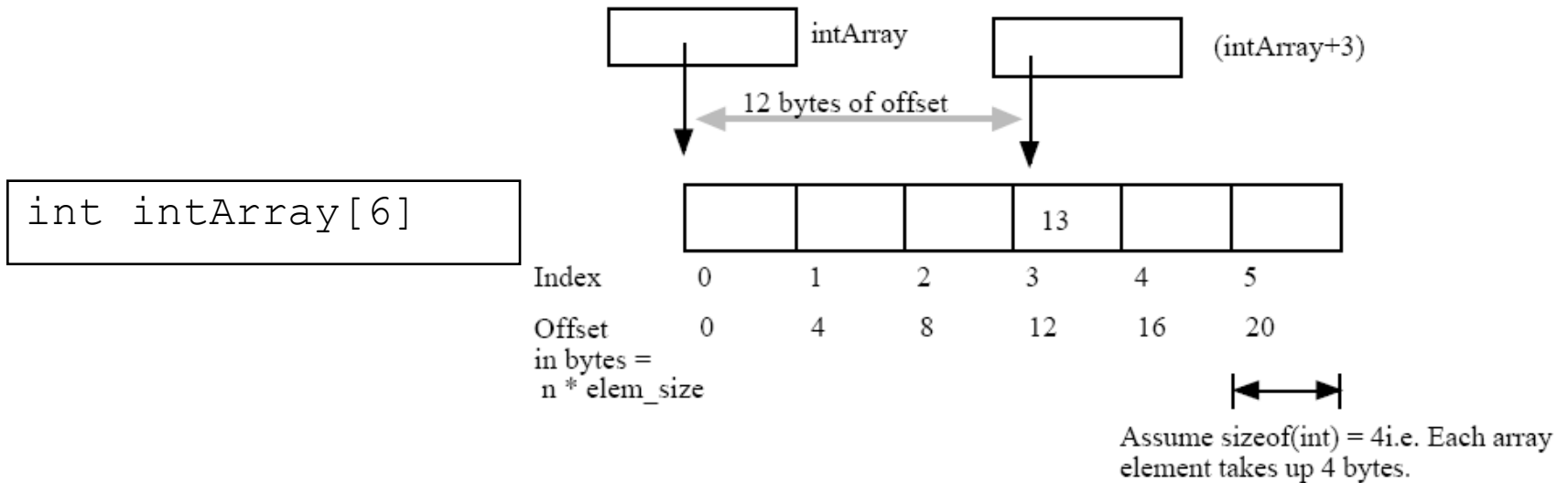
```
struct fraction* f1
```

Expression	Type
f1	struct fraction*
*f1	struct fraction
(*f1).numerator	int

There's an alternate, more readable syntax available for dereferencing a pointer to a struct. A "->" at the right of the pointer can access any of the fields in the struct. So the reference to the numerator field could be written

f1->numerator equals to **(*f1).numerator**

Pointers vs Arrays



intArray (with no brackets) is the address of the first byte of the array
The expression (intArray + 3) is a pointer to the integer intArray[3]

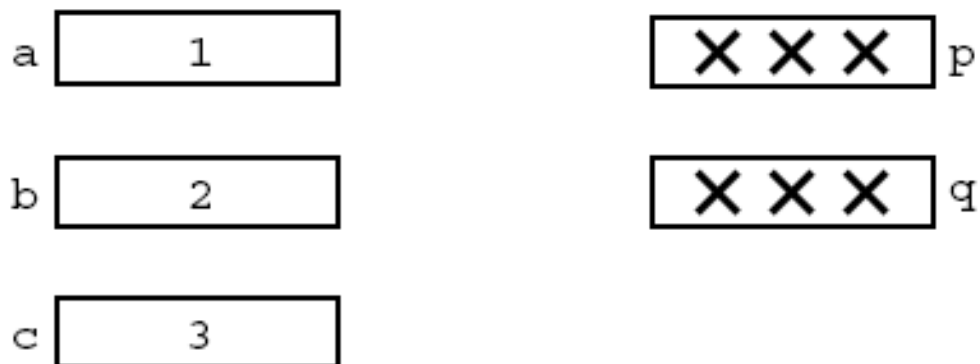
intArray[3] IS A NUMBER

(intArray + 3) IS AN ADDRESS

***(intArray + 3) is the same as intArray[3]**

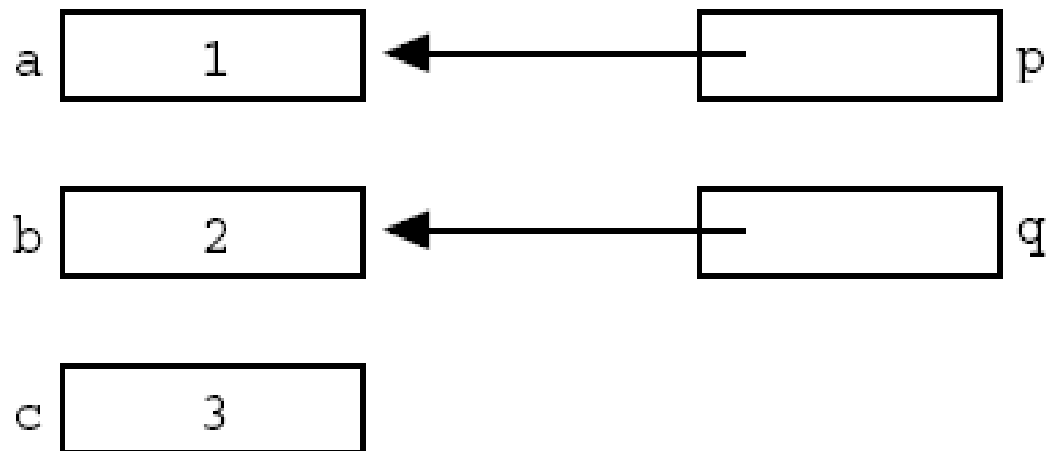
Example

```
void PointerTest () {  
    // allocate three integers and two pointers  
    int a = 1;  
    int b = 2;  
    int c = 3;  
    int* p;  
    int* q;  
  
    // Here is the state of memory at this point.  
    // T1 -- Notice that the pointers start out bad...
```



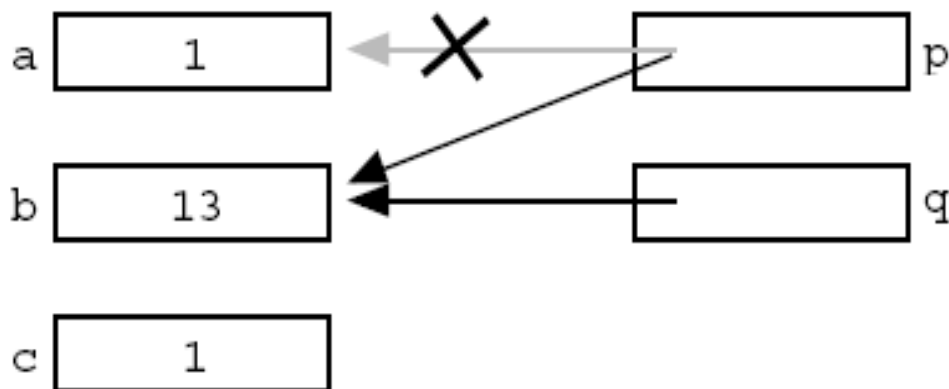
Example

```
p = &a;    // set p to refer to a  
q = &b;    // set q to refer to b  
// T2 -- The pointers now have pointees
```



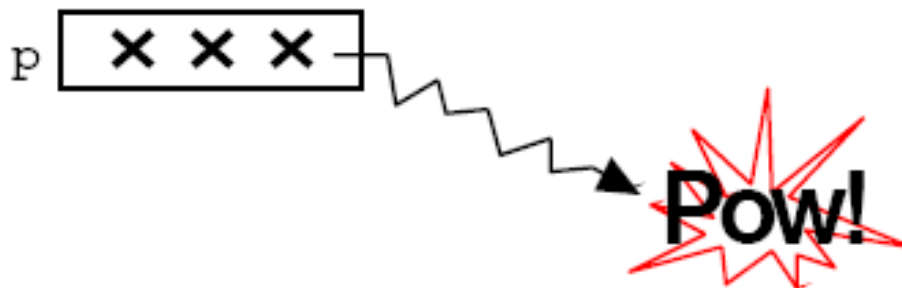
Example

```
// Now we mix things up a bit...  
c = *p; // retrieve p's pointee value (1) and put it in c  
p = q; // change p to share with q (p's pointee is now b)  
*p = 13; // dereference p to set its pointee (b) to 13 (*q is now 13)  
// T3 -- Dereferences and assignments mix things up
```



Bad Pointer Example

```
void BadPointer() {  
    int* p;      // allocate the pointer, but not the pointee  
  
    *p = 42;    // this dereference is a serious runtime error  
}  
// What happens at runtime when the bad pointer is dereferenced...
```



Pointing a pointer

Can we point a pointer?

Let's see what happens using another pointer

```
int ipointed = 3;
int *ip;
int *ipp;
ip = &ipointed;
ipp = &ip;
```

`ip` = address of `ipointed`

`*ip` = value of `ipointed`

`ipp` = address of `ip`

`*ipp` = address of `ipointed` **BUT** this would be interpreted as an integer value and **NOT** as an address!!!!!!!

We need something special...

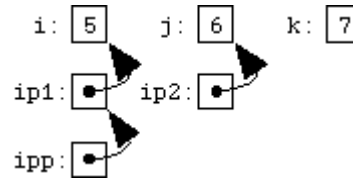
Pointers to pointers

We need a **pointer-to-pointer**, whose declaration looks like

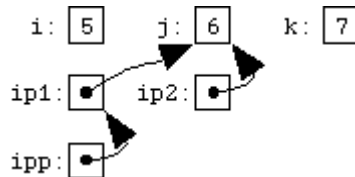
```
int **ipp;
```

Example:

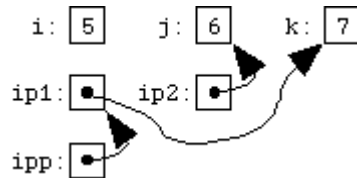
```
int **ipp;  
int i = 5, j = 6; k = 7;  
int *ip1 = &i, *ip2 = &j;  
ipp = &ip1;
```



```
*ipp = ip2;
```



```
*ipp = &k;
```



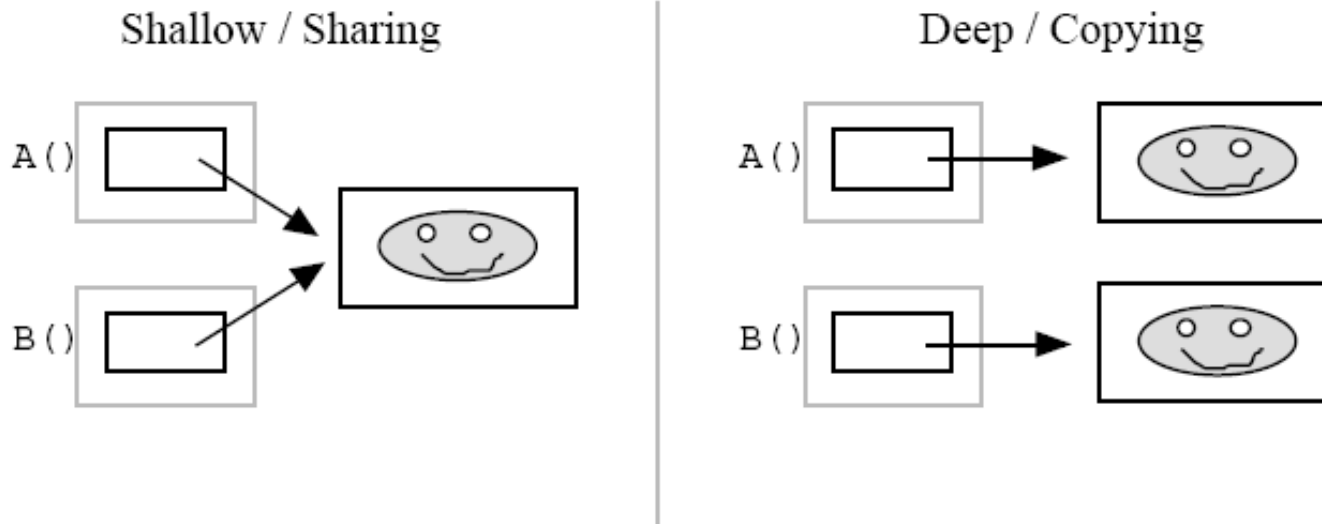
Pointers Summary

- A pointer stores a **reference to its pointee**. The pointee, in turn, **stores something useful**.
- The **dereference** (*) operation on a pointer accesses its pointee. A pointer may only be dereferenced **after** it has been assigned to refer to a pointee. Most pointer bugs involve violating this rule.
- **Allocating** a pointer does not automatically assign it to refer to a pointee. Assigning the pointer to refer to a specific pointee is a separate operation which is easy to forget.
- Assignment between two pointers makes them refer to the same pointee: introduces the concept of **memory sharing** (extremely important!!!)

Memory Sharing

- One of the key advantages of pointers is memory sharing: two pointers can refer to the same pointee

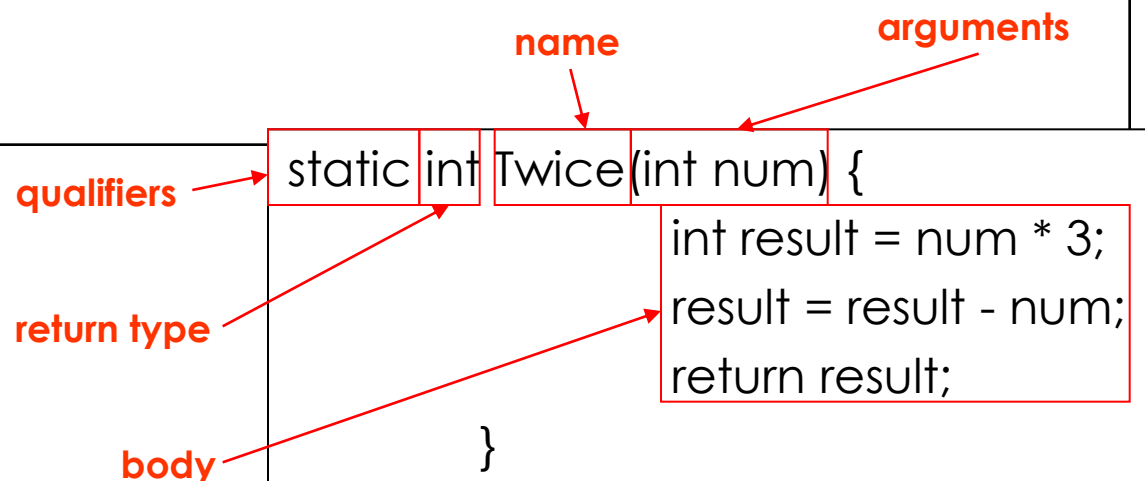
This can be particularly useful for functions that access the same values (Shallow Copying). The alternative is Deep Copying



FUNCTIONS

Functions

- All languages have a construct to **separate and package** blocks of code.
- C uses **the "function"** to package blocks of code.
- A function has a **name**, a **type**, a **list of arguments** which it takes when called, and the **block of code** it executes when called.
- C functions are defined in a text file and the names of all the functions in a C program are lumped together in a **single, flat namespace**.
- The special function called "**main**" is where program execution begins.
- The keyword "**static**" defines that the function will only be available to callers in the file where it is declared.



Functions (cont.ed)

- The expression passed to a function by its caller is called the "**actual parameter**"
- The parameter storage local to the function is called the "**formal parameter**"
- Parameters are passed "**by value**" that means there is a single **copying** assignment. The actual parameter is evaluated in the caller's context, and then the value is copied into the function's formal parameter just before the function begins executing. The alternative parameter mechanism is "by reference" which C **does not implement directly**
- The variables local to `Twice()`, `num` and `result`, only exist **temporarily** while `Twice()` is executing. This is the standard definition for "**local**" storage for functions.
- The return at the end of `Twice()` computes the return value and exits the function.

void type

void is a type formalized in ANSI C which means "nothing". To indicate that a function does not return anything, use void as the return type.

```
void function(int a);    // returns nothing
float function(void);    // takes no parameters
void function(void);    // returns nothing and takes no params
float function();        // Pay attention to ancient oddities!
```

Lexical Scoping

Every Variable is Defined within some scope. A Variable cannot be referenced by name from outside of that scope.

Lexical scopes are defined with curly braces { }.

→ The scope of Function Arguments is the complete body of the function.

→ The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block

→ The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called “Global” Vars.

The "Stack"

Recall lexical scoping. If a variable is valid "within the scope of a function", what happens when you call that function recursively? Is there more than one "exp"?

Yes. Each function call allocates a "stack frame" where Variables within that function's scope will reside.

float x	5.0	
int exp	0	Return 1.0
float x	5.0	
int exp	1	Return 5.0
int argc	1	
char **argv	0x2342	
float p	5.0	

```
#include <stdio.h>
#include <inttypes.h>

float pow(float x, int exp)
{
    /* base case */
    if (exp == 0) {
        return 1.0;
    }

    /* "recursive" case */
    return x*pow(x, exp - 1);
}

int main(int argc, char **argv)
{
    float p;
    p = pow(5.0, 1);
    printf("p = %f\n", p);
    return 0;
}
```

Call by value and by reference

- C passes parameters **"by value"** which means that the actual parameter values are **copied into local storage**.
- The caller and callee functions **do not share any memory** -- they each have their own copy.
- Two main disadvantages:
 1. Because the callee has its own copy, modifications to that memory are **not communicated back** to the caller (return is often not enough...)
 2. Sometimes it is undesirable to copy the value from the caller to the callee because the **value is large** and so copying it is expensive, or because at a conceptual level copying the value is undesirable.

The alternative is to pass the arguments **"by reference"**. Instead of passing a copy of a value from the caller to the callee, **pass a pointer to the value**.

Passing by reference

- Have a **single copy** of the value of interest. The single "master" copy.
- Pass pointers to that value to any function which wants to see or **change** the value.
- Functions can dereference their pointer to see or change the value of interest.
- Functions must remember that they do not have their own local copies. If they dereference their pointer and change the value, they really are **changing the master value**. If a function wants a local copy to change safely, the function must explicitly allocate and initialize such a local copy.

Example 1

```
void B(int worth) {  
    worth = worth + 1;  
    // T2  
}  
void A() {  
    int netWorth;  
    netWorth = 55; // T1  
  
    B(netWorth);  
    // T3 -- B() did not change netWorth  
}
```

T1 -- The value of interest netWorth is local to A().	T2 -- netWorth is copied to B()'s local worth. B() changes its local worth from 55 to 56.	T3 -- B() exits and its local worth is deallocated. The value of interest has not been changed.
A() netWorth 55	B() worth 55 56 A() netWorth 55	A() netWorth 55

Example 2

```
// B() now uses a reference parameter -- a pointer to
// the value of interest. B() uses a dereference (*) on the
// reference parameter to get at the value of interest.
void B(int* worthRef) {          // reference parameter
    *worthRef = *worthRef + 1; // use * to get at value of interest
    // T2
}

void A() {
    int netWorth;
    netWorth = 55;           // T1 -- the value of interest is local to A()

    B(&netWorth); // Pass a pointer to the value of interest.
                  // In this case using &.

    // T3 -- B() has used its pointer to change the value of interest
}
```

T1 -- The value of interest, netWorth, is local to A() as before.	T2 -- Instead of a copy, B() receives a pointer to netWorth. B() dereferences its pointer to access and change the real netWorth.	T3 -- B() exits, and netWorth has been changed.
A() netWorth 55	B() worth [] ↓ A() netWorth 55 56	A() netWorth 56

Example 3

```
// Takes the value of interest by reference and adds 2.
void C(int* worthRef) {
    *worthRef = *worthRef + 2;
}

// Adds 1 to the value of interest, and calls C().
void B(int* worthRef) {
    *worthRef = *worthRef + 1; // add 1 to value of interest as before

    C(worthRef);             // NOTE no & required. We already have
                             // a pointer to the value of interest, so
                             // it can be passed through directly.
}
```

Example 4

```
void Swap(int x, int y) {           // NO does not work
    int temp;

    temp = x;
    x = y;           // these operations just change the local x,y,temp
    y = temp;       // -- nothing connects them back to the caller's a,b
}
```

```
// Some caller code which calls Swap()...
int a = 1;
int b = 2;
Swap(a, b);
```

```
static void Swap(int* x, int* y) {   // params are int* instead of int
    int temp;

    temp = *x;           // use * to follow the pointer back to the caller's memory
    *x = *y;
    *y = temp;
}
```

```
// Some caller code which calls Swap()...
int a = 1;
int b = 2;

Swap(&a, &b);
```

Const variable

The qualifier `const` can be added to the left of a variable or parameter type to declare that the code using the variable **will not change its value**.

```
void foo(const struct fraction* fract);
```

Note that using qualifiers is not mandatory nor optional.

- They clarify the code (why not to use them?)
- Enforce correctness
- All qualifiers are a huge help to the compiler

Static and External variables

Static variables have a lifetime over the **entire program**, however the **scope is limited**. Static variables continue to exist even after the block in which they are defined terminates. Thus, the value of a static variable in a function is retained between repeated function calls to the same function.

```
static float sum;
```

External variables have global scope across the entire program (provided extern declarations are used in files other than where the variable is defined), and a lifetime over the the entire program run.

```
extern int index;
```

Extern variables example

```
#include <stdio.h>
#define MAXLINE 1000 /* maximum input line size */
int max;          /* maximum length seen so far */
char line[MAXLINE]; /* current input line */
char longest[MAXLINE]; /* longest line saved here */
int getline(void);
void copy(void);

/* print longest input line */
int main(void)
{
    int len; /* current line length */
    extern int max;
    extern char longest[]; // NOT necessary

    copy();

    if (max > 0) /* there was a line */
        printf("%s", longest);
    return 0;
}
```

```
/*getline: specialized version */
int getline(void)
{
    int c,i;
    extern char line[];
    ...
}
```

```
/* copy: specialized version */
void copy(void)
{
    int i;
    extern char line[],
    longest[];
    ...
}
```

Notice that:
array size is not needed because
no storage is being allocated

DYNAMIC MEMORY MANAGEMENT

Allocation and Deallocation

- Variables represent storage space in the computer's memory.
- C can assign memory to a variable **only when this is necessary**
- a variable is **allocated** when it is given an area of memory to store its value.
- A variable is **deallocated** when the system reclaims the memory from the variable, so it no longer has an area to store its value.
- For a variable, the period of time from its allocation until its deallocation is called its **lifetime**

Local Memory

The variables are called "**local**" to capture the idea that their lifetime is tied to the function where they are declared.

Whenever the function runs, its local variables are allocated. When the function exits, its locals are **deallocated**.

In any case, Once the flow of control leaves that body, there is no way to refer to the locals **even if they were allocated**. That locals are available ("scoped") only within their owning function is known as "lexical scoping" and pretty much all languages do it that way now.

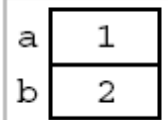
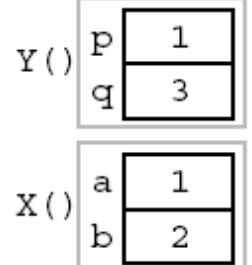
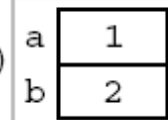
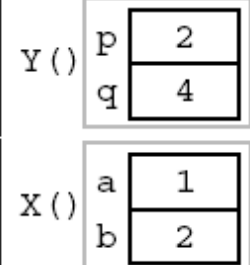
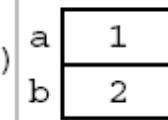
Example

```
void X() {
    int a = 1;
    int b = 2;
    // T1

    Y(a);
    // T3
    Y(b);

    // T5
}

void Y(int p) {
    int q;
    q = p + 2;
    // T2 (first time through), T4 (second time through)
}
```

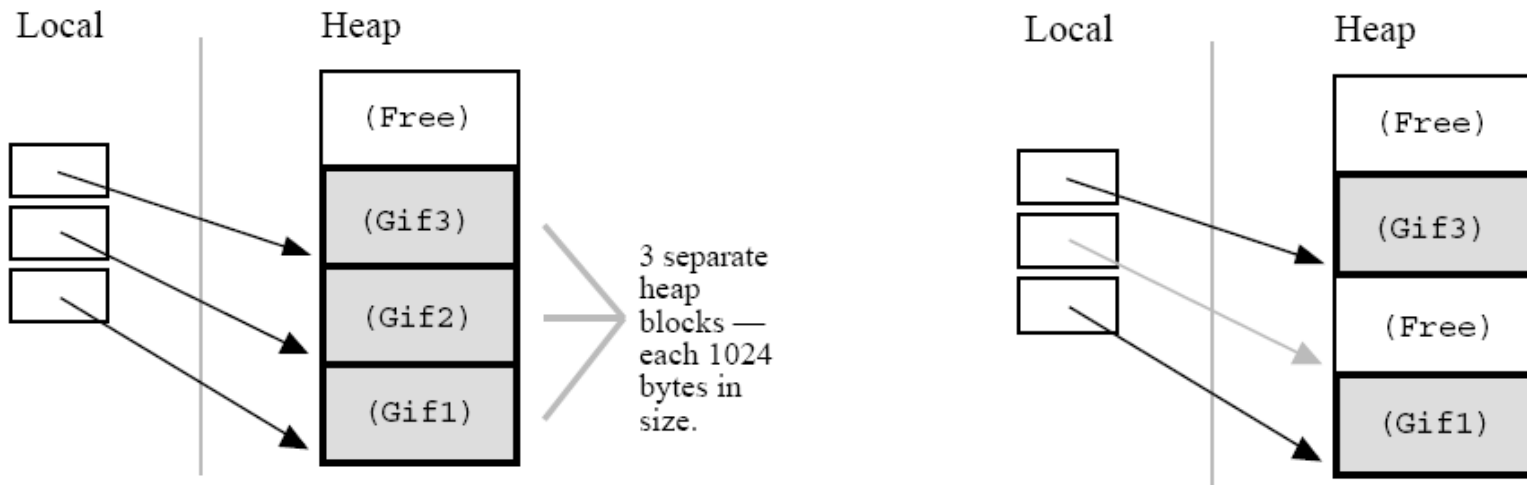
T1 - X()'s locals have been allocated and given values..	T2 - Y() is called with p=1, and its locals are allocated. X()'s locals continue to be allocated.	T3 - Y() exits and its locals are deallocated. We are left only with X()'s locals.	T4 - Y() is called again with p=2, and its locals are allocated a second time.	T5 - Y() exits and its locals are deallocated. X()'s locals will be deallocated when it exits.
				

Local Variables Advantages & Disadvantages

- **Convenient.** Local (automatic-stack) variables provide temporary, independent memory.
 - **Efficient.** Allocating and deallocating them is time efficient (fast) and they are space efficient in the way they use and recycle memory.
 - **Secure.** Local parameters are basically local copies of the information from the caller. This has the advantage that the callee can change its local copy without affecting the caller.
-
- **Short Lifetime.** Their allocation and deallocation schedule (their "lifetime") is very strict. Sometimes a program needs memory which continues to be allocated even after the function which originally allocated it, has exited.
 - **Restricted Communication.** Since locals are copies of the caller parameters, they do not provide a means of communication from the callee back to the caller.

Heap Memory

- "Heap" (Dynamic) memory is an alternative to local stack memory.
- The programmer explicitly requests the allocation of a memory "block" of a particular size, and the block continues to be allocated until the programmer explicitly requests that it is deallocated.
- Nothing happens automatically. So the programmer has much greater control of memory, but with greater responsibility



Heap Variables Advantages & Disadvantages

- **Lifetime.** Because the programmer now controls exactly when memory is allocated and deallocated, it is possible to build a data structure in memory, and return that data structure to the caller.
 - **Size.** The size of allocated memory can be controlled with more detail
-
- **More Work.** Heap allocation needs to be arranged explicitly in the code which is just more work.
 - **More Bugs.** Because it's now done explicitly in the code, realistically on occasion the allocation will be done incorrectly leading to memory bugs. Local memory is constrained but it's **less bug-prone** since it's managed by the runtime.

Syntax (include stdlib.h)

```
void* malloc(size_t size);
```

The malloc() function takes an unsigned integer which is the requested size of the block **measured in bytes**.

Malloc() **returns a pointer to a new heap block** if the allocation is successful, and **NULL** if the request cannot be satisfied because the heap is full.

The C operator

```
sizeof()
```

is a convenient way to compute the size in bytes of a type —sizeof(int) for an int pointer, sizeof(struct fraction) for a struct fraction pointer

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

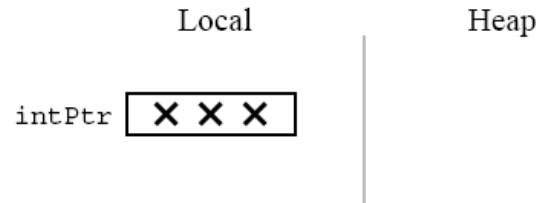
The free() function takes a pointer to a heap block and returns it to the free pool for later reuse. The pointer passed to free() **must be** the pointer returned earlier by malloc()

```
void *realloc(void * heapBlockPointer, unsigned long  
size);
```

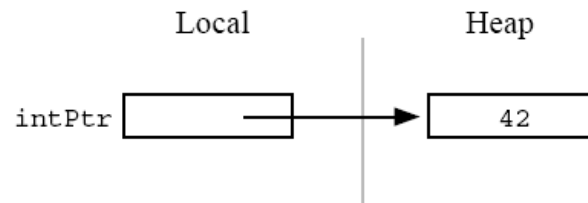
changes (increase or decrease) the size of the memory block pointed to by heapBlockPointer to size bytes.

Example

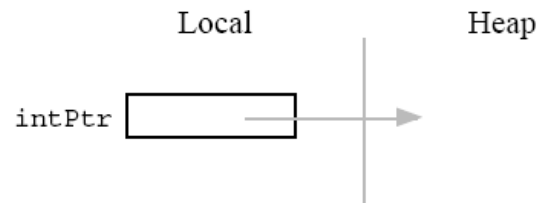
```
void Heap1() {  
    int* intPtr;  
    // Allocates local pointer local variable (but not its pointee)  
    // T1
```



```
    // Allocates heap block and stores its pointer in local variable.  
    // Dereferences the pointer to set the pointee to 42.  
    intPtr = malloc(sizeof(int));  
    *intPtr = 42;  
    // T2
```



```
    // Deallocates heap block making the pointer bad.  
    // The programmer must remember not to use the pointer  
    // after the pointee has been deallocated (this is  
    // why the pointer is shown in gray).  
    free(intPtr);  
    // T3
```



```
}
```


Memory Leaks

A program which forgets to deallocate a block is said to have a "memory leak" which is a really perfidious problem. The result will be that the heap gradually **fill up** as there continue to be allocation requests, but no deallocation requests to return blocks for re-use.

C does not have any garbage collector, a form of automatic memory management which attempts to reclaim memory used by objects that will never again be accessed or mutated by the application.

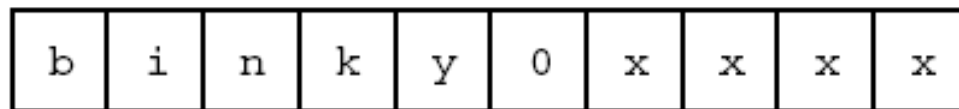
We need external tools!

STRINGS

C Strings

- C has **minimal support of character strings**.
- For the most part, strings operate as ordinary **arrays of characters**.
- Their maintenance is up to the programmer using the standard facilities available for **arrays and pointers**.
- The "null" character (**'\0'**) is stored after the last real character in the array to mark the **end of the string**
- A **"string" library** is available to manipulate strings (see later)

localString



0 1 2 ...

Copying strings

How to copy strings?

```
s = t;?
```

No: simply makes s and t point to same chars

Want: copy actual chars from source t to dest s

For now: assume s points to enough free space

Copy strings as arrays:

```
void strcpy(char s[], char t[]) {  
    int i = 0;  
    while ((s[i] = t[i]) != '\\0')  
        i++;  
}
```

Each time around, 1) Assign s char to t char; 2) check for \\0

Copying strings

Copy strings as pointers:

```
void strcpy(char *s, char *t) {
    while ((*s = *t) != '\\0') {
        s++;
        t++;
    }
}
```

Essentially the same as previous

No index needed, but now increment both pointers

Copying strings

Mix & match:

```
void strcpy(char s[], char *t) {
    /* s arr, t ptr */
    int i = 0;
    while ((*s = t[i]) != '\0') {
        s++; /* s ptr */
        i++; /* t arr */
    }
}
```

Copying strings

Shorter versions:

```
void strcpy(char *s, char *t) {  
    while ((*s++ = *t++) != '\0');  
}
```

Since the *value* of the null char '\0 is 0 (FALSE):

```
void strcpy(char *s, char *t) {  
    while ((*s++ = *t++));  
}
```

Dynamic Strings

The convention with C strings is that the owner of the string is responsible for allocating array space which is "large enough" to store whatever the string will need to store.

Two problems:

1. Strings could be larger than expected
2. Space could be wasted

The dynamic allocation of arrays works very well for allocating strings in the heap. The advantage of heap allocating a string is that the heap block can be just big enough to store the actual number of characters in the string.

```
#include <string.h>

/*
 * Takes a c string as input, and makes a copy of that string
 * in the heap. The caller takes over ownership of the new string
 * and is responsible for freeing it.
 */
char* MakeStringInHeap(const char* source) {
    char* newString;

    newString = (char*) malloc(strlen(source) + 1); // +1 for the '\0'
    assert(newString != NULL);
    strcpy(newString, source);
    return(newString);
}
```


THE STANDARD LIBRARY

C Standard Library Functions

Many basic housekeeping functions are available to a C program in form of standard library functions. To call these, a program **must #include the appropriate .h file**. Most compilers link in the standard library code by default. Most common libraries:

- `stdio.h` file input and output
- `cctype.h` character tests
- `string.h` string operations
- `math.h` mathematical functions such as `sin()` and `cos()`
- `stdlib.h` utility functions such as `malloc()` and `rand()`
- `assert.h` the `assert()` debugging macro
- `stdarg.h` support for functions with variable numbers of arguments
- `setjmp.h` support for non-local flow control jumps
- `signal.h` support for exceptional condition signals
- `time.h` date and time
- `limits.h`, `float.h` constants which define type range values such as `INT_MAX`

SEE <http://www.cppreference.com/>

I/O

Arguments to Main

Arguments to main provide a useful way to give parameters to programs.

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

When a program starts, the arguments to main will have been initialized to meet the following conditions:

- argc is larger than zero.
- argv[argc] is a null pointer.
- argv[0] through to argv[argc-1] are pointers to **strings** whose meaning will be determined by the program.
- argv[0] will be a string containing the program's name or a null string if that is not available. Remaining elements of argv represent the arguments supplied to the program.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    while(argc-->0) printf("%s\n", *argv++);
    exit(EXIT_SUCCESS);
}
```

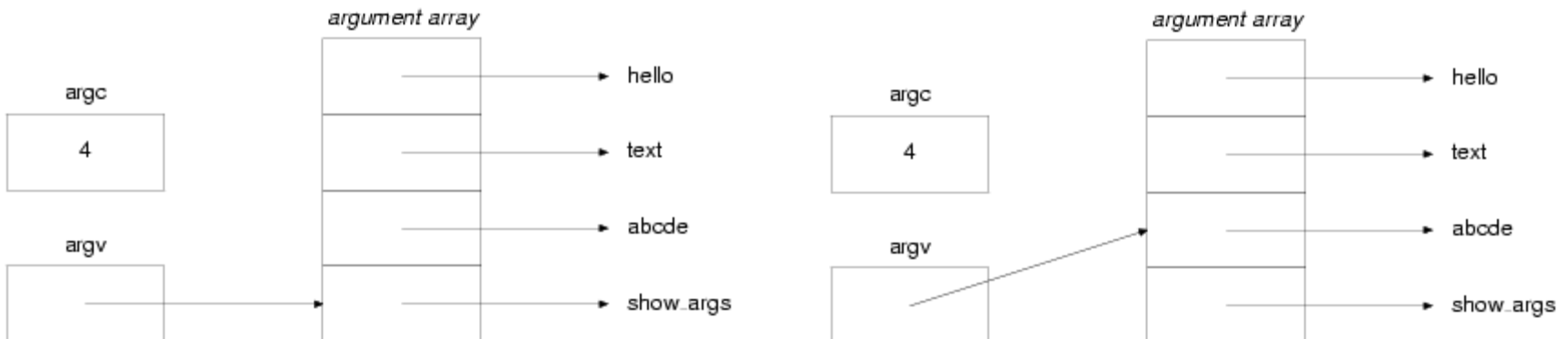
>./show_args abcde text hello

show_args

abcde

text

hello



Read/Write Data

Typical I/O functions are available in the **stdio** library.

Require inclusion of `stdio.h`

Predefined Types and Values:

- **FILE** is a datatype which holds information about an open file.
- **EOF** is a value returned to indicate end-of-file and is required by ANSI C to be a negative constant integer expression and is traditionally set to -1.
- **size_t** is an unsigned integer type which is large enough to hold any value returned by `sizeof`

Stdio: stdin, stdout, stderr

Three file streams are predefined and preopened:

- FILE *stdin
 - stdin is associated with a user's standard input stream.
- FILE *stdout
 - stdout is associated with an output stream used for normal program output.
- FILE *stderr
 - stderr is associated with an output stream used for error messages.

Useful functions

- printf
- scanf

Stdio: Read/Write Data from/in Files

File stream must be specified, opened and closed

- `fopen`
- `fclose`
- `fread`
- `fwrite`
- `fprintf`
- `fscanf`
- `fflush`

The format string

Every I/O function that requires a “format” string (*print*, *scan*) expects a description of the formatted stream:

```
printf("I'm iteration %5d, my value is %.4lf \n", i, val);  
> I'm iteration      1, my value is 3.1002
```

The placeholder characters contained in the format string will be replaced with the actual values of parameters.

The format string: placeholders

%[flags][min field width][precision][length]conversion specifier

\	#, *	.#, .*	/	\
\	\		/	\
#, 0, -, +, , ', I		hh, h, l, ll, j, z, L		c, d, u, x, X, e, f, g, s, p, %
# Alternate,		hh char,		c unsigned char,
0 zero pad,		h short,		d signed int,
- left align,		l long,		u unsigned int,
+ explicit + - sign,		ll long long,		x unsigned hex int,
space for + sign,		j [u]intmax_t,		X unsigned HEX int,
' locale thousands grouping,		z size_t,		e [-]d.ddde±dd double,
I Use locale's alt digits		t ptrdiff_t,		E [-]d.dddE±dd double,
		L long double,		-----
if no precision => 6 decimal places			/	f [-]d.ddd double,
if precision = 0 => 0 decimal places			_____ /	g e f as appropriate,
if precision = # => # decimal places				G E F as appropriate,
if flag = # => always show decimal point				s string,
			-----
			/	p pointer,
if precision => max field width			/	% %

Example

```
#include <stdio.h>
main()
{
    int i1;
    int i2;
    printf("Enter two integers: ");
    scanf("%d %d",&i1,&i2);
    printf("\nThe product of %d and %d is %d.\n",i1,i2,i1*i2);
    return 0; /* no error */
}
```

Example

```
#include <stdio.h>

int main(int argc, char **argv) {
    char * input_filename = "data_input";
    FILE * input_file;
    char * output_filename = "data_output";
    FILE * output_file;
    int number;

    input_file = fopen(input_filename, "r");
    if(input_file == NULL) {
        fprintf(stderr, "Can't open %s.\n", input_filename);
        return 1;
    }

    output_file = fopen(output_filename, "w");
    if (output_file == NULL) {
        fprintf(stderr, "Can't open %s.\n", output_filename);
        return 1;
    }

    fscanf(input_file, "%d", &number);

    fprintf(output_file, "The square of %d is %d.\n", number, number*number);
    fclose(input_file);
    fclose(output_file);
    return 0;
}
```

Notes about mixing FORTRAN and C

Overview

- **Total absence of standard coverage**
- **Extremely unportable** (dependent on compilers, operating systems, libraries, etc...)
- No facilities provided, needs to be totally **hand-coded**



Problems arising everywhere

Strings

```
void foo_c_function(int a, char *s)
{
    struct coords c;
    ...
}
```

Objects alignment

Base types associations

Symbols decoration

Array memory ordering

Array bounds

```
subroutine foo_fort_subroutine(a, b)
    integer*4, intent(inout) :: a
    real*8, intent(inout) :: b
    integer*4, dimension(10) :: v
    ...
end subroutine foo_fort_subroutine
```

Base types

- Every **implementation** (compiler+operating system+machine) has its own association.
- There is no straightforward solution.
- **You should take total control of portability: define your own types for interfaces!**

FORTRAN (x86_64 linux, GCC 4.1)	C/C++ (x86_64 linux, GCC 4.1)
byte	unsigned char
integer*2	short int
integer	int
integer matrix(2,3)	int matrix[3][2];
logical	int
logical*1	unsigned char (C), bool (C++)
real*4	float
real*8	double
real*16	long double
complex	struct{float r; float i;}
double complex	struct{double r; double i;}
character*6 string	char string[6];
character*6 strings(4)	char strings[4][6];
parameter	#define <i>PARAMETER value</i>

Linkage

- Some compilers “decorate” the symbols of FORTRAN routines to avoid clashes.
- A generic subroutine named `test` becomes:
 - `test_` on Linux (Intel/GCC/PGI)
 - `test` on AIX (VisualAge/GCC)
 - `test__` on some exotic systems
- If you want to invoke that subroutine from C (or vice-versa!), **you need to know how the symbols are internally modified by the compiler.**
- Compiler flags exist, but **the only reliable solution is to take total control of the situation (exploit the preprocessor).**

Function/routine parameters

- Regardless of the specified INTENT, **every FORTRAN subroutine takes reference only parameters.**
- Remember this rules:

C calling FORTRAN: pass pointers as actual parameters

```
subroutine foo(a, b)
  integer*4, intent(in) :: a
  real*8, intent(out) :: b
  ...
```

```
int c_a = 100;
double c_b;

foo_(&c_a, &c_b);
```

FORTRAN calling C: the C function must accept pointers only

```
void foo_(int *a, double *b)
{
  ...
```

```
integer*4 :: f_a
real*8 :: f_b

call foo(f_a, f_b)
```

- **Pay extreme attention or you'll end up with a corrupted stack**

Strings

- C strings are just arrays of integers.
- In FORTRAN, they are **opaque objects** managed by the runtime.
- For example, how can we retrieve a string `len()` ?

```
subroutine foo(s)
  character, dimension(:), intent(in), :: s

  write(*,*) len(s)
  ...
```

The compiler manages internally the opaque object's properties

```
subroutine foo(s, _s_len)
  character, dimension(:), intent(in), :: s
  integer, intent(in), :: _s_len

  write(*,*) len(s) ! write(*,*) _s_len
  ...
```

Strings

- The fortran string internal implementation is **unspecified**

Passed by value!!!

Linux (GNU/Intel)

```
void foo_(char *a, int _a_len, char *b, int _b_len);
```

AIX (VisualAge)

```
void foo(char *a, char *b, int _a_len, int _b_len);
```

SUN (SunStudio)

```
typedef struct
{
    char *string;
    int length;
} fort_string;

void foo_(fort_string a, fort_string b);
```

- Investigate your system's implementation and port your interfaces from scratch!**

Strings

- Remember that C strings are **null terminated!**

```
character, dimension(32) :: f_string = "I am FORT"//CHAR(0)  
call c_function(f_string)
```

- **Convert your FORTRAN strings prior the invocation of a C function!**
- **When reading back, remember the extra character CHAR(0)!**

Arrays

- For single dimension arrays, the only thing to remember is the **bounds difference** (0,N-1 for C and 1,N for FORTRAN)
- For multi-dimensional arrays, the things get tricky:

```
integer, dimension(2,3), f
```

COLUMN MAJOR

```
! Memory snapshot:
```

```
! f(1,1) f(2,1) f(1,2) f(2,2) f(1,3) f(2,3)
```

```
int c[2][3];
```

ROW MAJOR

```
// Memory snapshot:
```

```
// c[0,0] c[0,1] c[0,2] c[1,0] c[1,1] c[1,2]
```

- **Pay extreme attention (reorder loops) or use tricks (transpose in C, FORTRAN90 RESHAPE)**

Common blocks/Extern structures

- Mapping FORTRAN common blocks to C structures (and vice-versa) is possible but **extremely dangerous and unsafe**

```
real*8 r
integer*4 i_a, i_b, i_c
common/fortdata/ r, i_a, i_b, i_c
```

```
extern struct {
    double r;
    int i_a, i_b, i_c;
} fortdata_;
```

Pay attention to:

- Fields alignment
 - First access (FORTRAN side must access first!)
 - Use named commons only
-
- **You can control the situation using compiler flags and pragmas only!**

USE THIS TECHNIQUE AT YOUR OWN RISK

CASE STUDY: LINKED LISTS

Linked Lists

A method of organizing stored data in a computer's memory or on a storage medium based on the **logical order** of the data and **not the physical order**.

It is a list in which each data element has information for locating the **next**. The data elements may be in **noncontiguous** storage locations.

Concepts involved:

- Pointers
- Structures
- Dynamic memory management

Arrays Reminder

An array is a collection of elements organized such that they have **contiguous** storage locations

Array access is always implemented using **fast** address arithmetic: the address of an element is computed as an offset from the start of the array which only requires one multiplication and one addition.

```
int scores[100];
```



Arrays Disadvantages

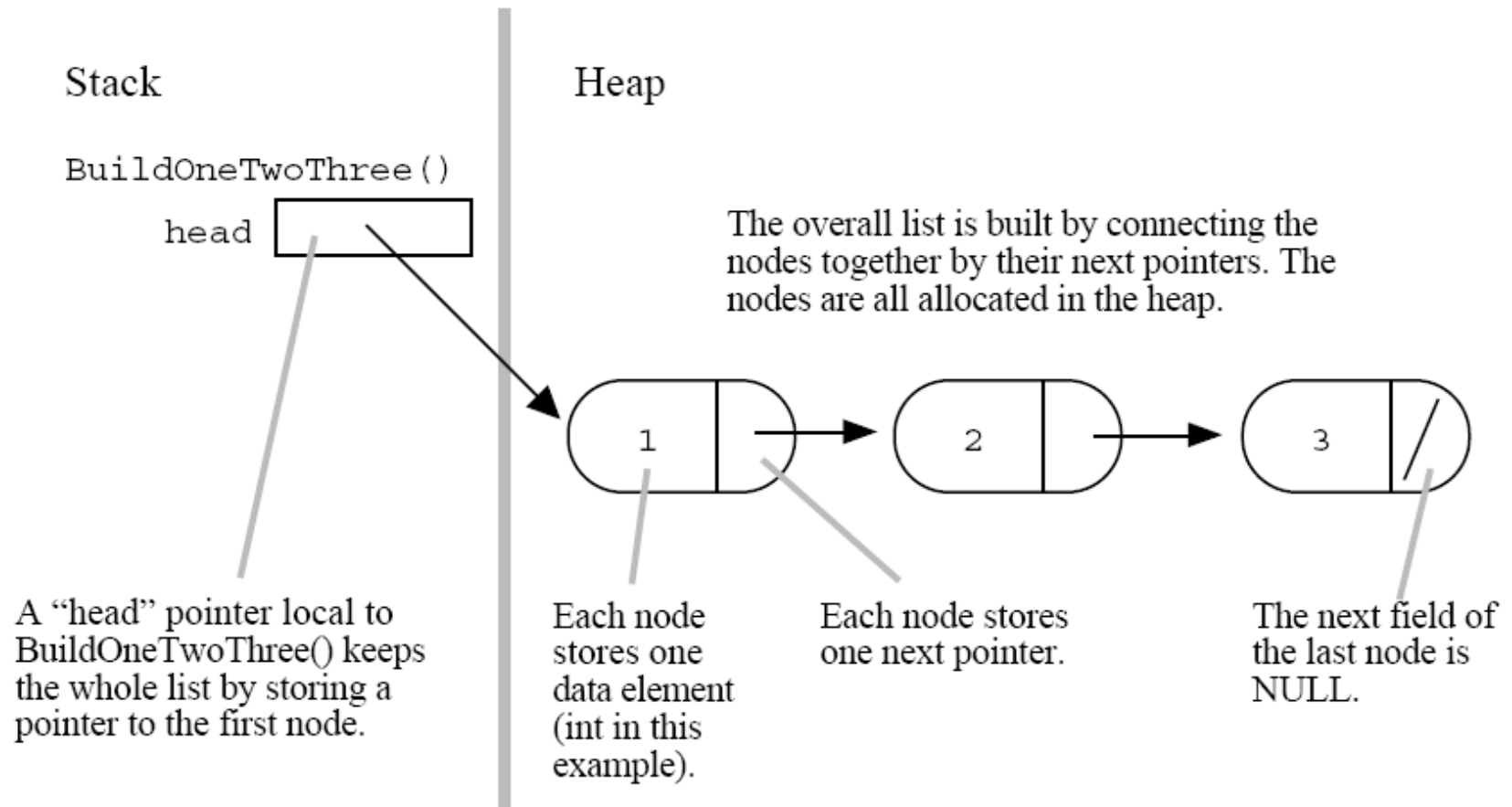
1. The size is fixed. Needs reallocation of memory to change
2. Often waste of space (oversized arrays)
3. Difficult to add new elements in front of or between elements

Linked Lists are strong where Arrays are weak (and vice-versa...)

Linked Lists Basics

- An array allocates memory for all its elements as a contiguous block of memory.
- In contrast, a linked list allocates space for **each element separately** in its own block of memory called a "linked list element" or "**node**".
- The list gets its overall structure by using pointers to connect all its nodes together like the links in a **chain**.
- Each node contains two fields: a "**data**" field and a "**next**" field which is a pointer used to link one node to the next node.
- Each node is allocated in the heap with a call to `malloc()`, so the node memory continues to exist until it is **explicitly deallocated** with a call to `free()`.
- The **front** of the list is a pointer to the first node.

A Simple Linked List



The “node” Data Type

- Define the type for the nodes which will make up the body of the list.
- These are allocated in the heap. Each node contains a single client data
- element and a pointer to the next node in the list:

```
struct node {  
    int data;  
    struct node* next;  
};
```

Extremely IMPORTANT: next is a pointer, its value is the ADDRESS of next node. All the linked list will be based on this feature!!!!

The BuildOneTwoThree() function

```
/*  
  Build the list {1, 2, 3} in the heap and store  
  its head pointer in a local stack variable.  
  Returns the head pointer to the caller.  
*/  
struct node* BuildOneTwoThree() {  
    struct node* head = NULL;  
    struct node* second = NULL;  
    struct node* third = NULL;  
  
    head = malloc(sizeof(struct node));    // allocate 3 nodes in the heap  
    second = malloc(sizeof(struct node));  
    third = malloc(sizeof(struct node));  
  
    head->data = 1;    This is an address!!! // setup first node  
    head->next = second;    // note: pointer assignment rule  
  
    second->data = 2;    // setup second node  
    second->next = third;  
  
    third->data = 3;    // setup third link  
    third->next = NULL; NULL is the NEXT one!!!  
  
    // At this point, the linked list referenced by "head"  
    // matches the list in the drawing.  
    return head;  
}
```

The Length() function

```
/*
  Given a linked list head pointer, compute
  and return the number of nodes in the list.
*/
int Length(struct node* head) {
    struct node* current = head;
    int count = 0;

    while (current != NULL) {
        count++;
        current = current->next; This is an address!!!
    }

    return count;
}
```


Put Together

```
void LengthTest() {  
    struct node* myList = BuildOneTwoThree();  
  
    int len = Length(myList); // results in len == 3  
}
```

This is a pointer

Generalization

- The linked list can be expanded “indefinitely”
- However the access to an element requires going through the list. Slower and slower for “last elements”. Fast for first elements
- Deleting a list means deleting all the single nodes
- Adding a node is generally done at the beginning of the list. It is a three steps procedure:
 1. Allocate the new node in the heap
 2. Set the next pointer of the new node to point to the current first node of the list
 3. Change the head pointer to point to the new node

3 Steps Link Code

```
void LinkTest() {
    struct node* head = BuildTwoThree(); // suppose this builds the {2, 3} list
    struct node* newNode;

    newNode= malloc(sizeof(struct node)); // allocate
    newNode->data = 1;

    newNode->next = head;                // link next
    head = newNode;                      // link head

    // now head points to the list {1, 2, 3}
}
```

Now we can think of writing a general function which adds a single node to head end of any list. Historically, this function is called "Push()" since we're adding the link to the head end

The Wrong "Push" function

```
void WrongPush(struct node* head, int data) {
    struct node* newNode = malloc(sizeof(struct node));

    newNode->data = data;
    newNode->next = head;
    head = newNode;          // NO this line does not work!
}
```

```
void WrongPushTest() {
    List head = BuildTwoThree();

    WrongPush(head, 1);    // try to push a 1 on front -- doesn't work
}
```

- head **IS A POINTER**
- We are passing it **by VALUE** (don't be confused by the * in the argument. *** IS PART of the head type**)
- head inside Push is a **LOCAL** variable. Its value is modified but this has **NO feedback OUTSIDE**

The Right "Push" function

```
/*
  Takes a list and a data value.
  Creates a new link with the given data and pushes
  it onto the front of the list.
  The list is not passed in by its head pointer.
  Instead the list is passed in as a "reference" pointer
  to the head pointer -- this allows us
  to modify the caller's memory.
*/
void Push(struct node** headRef, int data) {
    struct node* newNode = malloc(sizeof(struct node));

    newNode->data = data;
    newNode->next = *headRef; // The '*' to dereferences back to the real head
    *headRef = newNode;      // ditto
}

void PushTest() {
    struct node* head = BuildTwoThree(); // suppose this returns the list {2, 3}

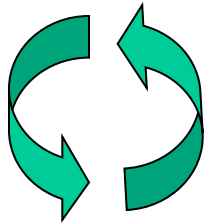
    Push(&head, 1); // note the &
    Push(&head, 13);

    // head is now the list {13, 1, 2, 3}
}
```

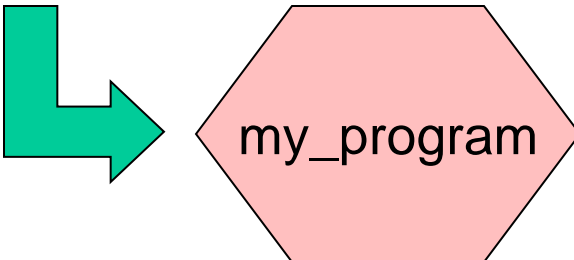
A BIT OF “HANDS-ON”

Writing and Running Programs

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```



```
$ gcc -Wall -g my_program.c -o my_program
tt.c: In function `main':
tt.c:6: parse error before `x'
tt.c:5: parm types given both in parmlist and separately
tt.c:8: `x' undeclared (first use in this function)
tt.c:8: (Each undeclared identifier is reported only once
tt.c:8: for each function it appears in.)
tt.c:10: warning: control reaches end of non-void function
tt.c: At top level:
tt.c:11: parse error before `return'
```



1. Write text of program (source code) using an editor such as emacs, save as file e.g. my_program.c

2. Precompile the code

2. Run the compiler to convert program from source to an “executable” or “binary”:

```
$ gcc -Wall -g my_program.c -o my_program
```

3-N. Compiler gives errors and warnings; edit source file, fix it, and re-compile

N. Run it and see if it works 😊

```
$ ./my_program
Hello World
$
```

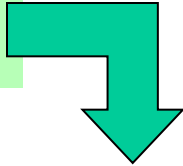
```
./?
```

What if it doesn't work?

A Quick Digression About the Compiler

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```

Preprocess



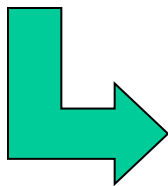
Compilation occurs in two steps:
“Preprocessing” and “Compiling”

Why ?

```
__extension__ typedef unsigned long long int __dev_t;
__extension__ typedef unsigned int __uid_t;
__extension__ typedef unsigned int __gid_t;
__extension__ typedef unsigned long int __ino_t;
__extension__ typedef unsigned long long int __ino64_t;
__extension__ typedef unsigned int __nlink_t;
__extension__ typedef long int __off_t;
__extension__ typedef long long int __off64_t;
extern void flockfile (FILE *__stream) ;
extern int ftrylockfile (FILE *__stream) ;
extern void funlockfile (FILE *__stream) ;
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```

In Preprocessing, source code is “expanded” into a larger form that is simpler for the compiler to understand. Any line that starts with ‘#’ is a line that is interpreted by the Preprocessor.

- Include files are “pasted in” (#include)
- Macros are “expanded” (#define)
- Comments are stripped out (/* */ , //)
- Continued lines are joined



Compile

my_program

The compiler then converts the resulting text into binary code the CPU can run directly

Final step: the loader (linker)

- The loader merge together all the different functions that compose the final program:

```
gcc -o my_exec.out -L/usr/local/lib -l lib1 -L/my_local_lib/lib -l lib2
```

The makefile utility

If a code is large and/or it shares subroutines with other codes, it is useful to split the source in many files that could be placed in different directories.

To avoid compiling by hands the sources in the proper order, the **make** command could be used

The **make** command can be programmed to do the job for you using a file containing instructions and directives.

By default the make command looks in the present directory for a file called Makefile or makefile


A simple makefile

```
# this is a comment within the makefile
```

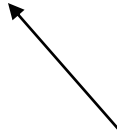
```
myprog.x : main.o  
    gcc -o myprog.x main.o
```

```
main.o : main.c  
    gcc -c main.c
```

this tell to the make command
that myprog.x depend from
main.o



make execute the command only
when main.o
have been built



to compile the code, from the console the programmer issue
the command:

```
> make
```