C Programming

Both the Unix and Linux kernels are written in C; consequently, the Unix systems' calling interface is also in C.

C Programming

Both the Unix and Linux kernels are written in C; consequently, the Unix systems' calling interface is also in C.

We need to be able to use the C language.

C Programming

Both the Unix and Linux kernels are written in C; consequently, the Unix systems' calling interface is also in C.

We need to be able to use the C language.

Java, C++, Objective C, and undoubtedly more languages are related (descendants of) to C.

C Programming

Both the Unix and Linux kernels are written in C; consequently, the Unix systems' calling interface is also in C.

We need to be able to use the C language.

Java, C++, Objective C, and undoubtedly more languages are related (descendants of) to C.

There will be problems scattered throughout these slides; I suggest that you try them out.

A Tiny Program in C

#include <stdio.h > *sets up input/output*

A Tiny Program in C

#include <stdio.h > *sets up input/output*

int main() *a C program begins execution by calling main()*

{

A Tiny Program in C

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

sets up input/output

```
int main()
```

a C program begins execution by calling main()

```
{
  printf("this is a small C program\n");
```

is a function provided by the standard C library to display text to the terminal

A Tiny Program in C

#include <stdio.h >	*sets up input/output*
int main()	*a C program begins execution by calling main()*
{	
printf("this is a small C program\n");	*is a function provided by the standard C library to display text to the terminal*
return 0;	*C programs are expected to provide a "status code" on exit; by convention, a "0" means success*
}	

A Tiny Program in C

#include <stdio.h >	*sets up input/output*
int main()	*a C program begins execution by calling main()*
{	
printf("this is a small C program\n");	*is a function provided by the standard C library to display text to the terminal*
return 0;	*C programs are expected to provide a "status code" on exit; by convention, a "0" means success*
}	

Problem 1 *Try compiling, linking, and running this program.*
ie. cc asmallpgm.c -o asmallpgm
asmallpgm

Data Types and Variables

- All variables in C must be *declared* or *defined* prior to use

Data Types and Variables

- All variables in C must be *declared* or *defined* prior to use
- Declaration: gives typing information about a variable (ie. prototypes)

Data Types and Variables

- All variables in C must be *declared* or *defined* prior to use
- Declaration: gives typing information about a variable (ie. prototypes)
- Definition : is a declaration *and* it causes memory to be assigned

Simple Data Types

Type		Min Range		Bytes	
void		to		0	
signed	char	-128 to	127	1	
unsigned	char	0 to	255	1	
enum		-32768 to	32767	2	*(may be 4 bytes)*
signed	short	-32768 to	32767	2	
unsigned	short	0 to	65535	2	
signed	int	-32768 to	32767	2	*(may be 4 bytes)*
unsigned	int	0 to	65535	2	*(may be 4 bytes)*
signed	long	-2,147,483,648 to	2,147,483,647	4	
unsigned	long	0 to	4,294,967,295	4	
	float	3.4e-38 to	3.4e+38	4	
	double	1.7e-308 to	1.7e+308	8	
long	double	3.6e-4951 to	1.19e+4932	16	(64-bit, Intel)
(pointer)		to		4	

(*mostly assuming 32-bit systems*)

Examples

Example 1 *char a,b,c; these allocate three bytes, one byte per var*

Examples

Example 1 *char a,b,c; these allocate three bytes, one byte per var*

Example 2 *char a='b'; initialize a to the character 'b'*

Examples

Example 1 *char a,b,c; these allocate three bytes, one byte per var*

Example 2 *char a='b'; initialize a to the character 'b'*

Problem 2 *Write a set of declarations, one for each variable type given in the previous slide*

Declaring Pointers

datatype *varname;

Declaring Pointers

datatype *varname;

- A pointer variable simply holds an address. In that sense all pointers are the same.

Declaring Pointers

datatype *varname;

- A pointer variable simply holds an address. In that sense all pointers are the same.
- A process’ memory space may be considered to be a list of bytes, numbered from 0 to whatever

Declaring Pointers

datatype *varname;

- A pointer variable simply holds an address. In that sense all pointers are the same.
- A process' memory space may be considered to be a list of bytes, numbered from 0 to whatever
- An address is simply an integer, and is an offset (in bytes) to somewhere in the process' memory space

Declaring Pointers

datatype *varname;

- A pointer variable simply holds an address. In that sense all pointers are the same.
- A process' memory space may be considered to be a list of bytes, numbered from 0 to whatever
- An address is simply an integer, and is an offset (in bytes) to somewhere in the process' memory space
- However, in C, pointers are declared with a type. The pointer, then, is the integer of bytes offset to where some typed item is stored (begins).

Declaring Pointers

datatype *varname;

- A pointer variable simply holds an address. In that sense all pointers are the same.
- A process' memory space may be considered to be a list of bytes, numbered from 0 to whatever
- An address is simply an integer, and is an offset (in bytes) to somewhere in the process' memory space
- However, in C, pointers are declared with a type. The pointer, then, is the integer of bytes offset to where some typed item is stored (begins).
- Doing basic arithmetic to that pointer (adding one, subtracting two) is actually done in increments of the size of the object being pointed to.

Declaring Pointers

datatype *varname;

- A pointer variable simply holds an address. In that sense all pointers are the same.
- A process' memory space may be considered to be a list of bytes, numbered from 0 to whatever
- An address is simply an integer, and is an offset (in bytes) to somewhere in the process' memory space
- However, in C, pointers are declared with a type. The pointer, then, is the integer of bytes offset to where some typed item is stored (begins).
- Doing basic arithmetic to that pointer (adding one, subtracting two) is actually done in increments of the size of the object being pointed to.

Example 3 *Consider* `char *px;`
Note that `sizeof(char)` *is 1*
Assume that px has the value 0xa2340 (that's hexadecimal).
Then px+1 is 0xa2341

Declaring Pointers

datatype *varname;

- A pointer variable simply holds an address. In that sense all pointers are the same.
- A process' memory space may be considered to be a list of bytes, numbered from 0 to whatever
- An address is simply an integer, and is an offset (in bytes) to somewhere in the process' memory space
- However, in C, pointers are declared with a type. The pointer, then, is the integer of bytes offset to where some typed item is stored (begins).
- Doing basic arithmetic to that pointer (adding one, subtracting two) is actually done in increments of the size of the object being pointed to.

Example 3 *Consider* `char *px;`
Note that `sizeof(char)` *is 1*
Assume that px has the value 0xa2340 (that's hexadecimal).
Then px+1 is 0xa2341

Example 4 *Consider* `float *px;`
Note that `sizeof(float)` *is 4*
Assume that px has the value 0xa2340 (that's hexadecimal).
Then px+1 is 0xa2344

Pointers to Functions

Functions are normally written like

```
type functionname(argument-list)
{
}
```

Pointers to Functions

Functions are normally written like

```
type functionname(argument-list)
{
}
```

One may have variables holding pointers to functions. For example,

```
int (*pf)(float);
```

declares a variable pf that points to a function which takes one argument of type float and which returns an int.

Aggregate Data Types: arrays and strings

arrays an array is a block of consecutive data items all of the same type, referenced via a `[subscript]` notation. All arrays in C start with a subscript of zero: `a[0]`.

Example 5 *Two simple string definitions:*

```
double abc[4];
char   def[5];
```

strings are simply a convenient way of expressing an array of `char`. ex. "this is a string." This form cannot be used as a declaration, but can be used as initialization (see pointers), as arguments to functions, and in assignments.

Example 6 *Two simple string definitions:*

```
char *x= "a simple string";
char y[]= "another simple string";
```

In C, strings always end in a null byte, so array will require one additional byte beyond what is visible.

Aggregate Data Types: structures

structures A structure holds a collection of data associated with instance(s) of something.

definition `struct [name] { ... } [varnames];`

Example 7 *A typical structure*

```
struct Abc_str {
    char   x;
    double y;
    struct Abc_str *nxt;
    struct Abc_str *prv;
    };
```

typing One may set up a new "type" for a structure:

`typedef struct Abc_str Abc;`

The `Abc` may now be used where other types are used.

declaration One may declare a variable to be a structure instance:

Example 8 *Declaring a structure variable:*

```
struct Abc_str abc;
Abc abc;              // (assuming that a typedef was used)
```

This defines a collection of data whose leading byte is a single char followed by a double. Note the two pointers, too.

Aggregate Data Types: unions

unions A union syntactically resemble structures, but all the data items begin at the same address in memory. *(ie. they overlap!)*

definition `union [name] { ... } [varnames];`

Example 9 *A typical union:*

```
union CL_un {
    char c[4];
    long l;
    } u;
```

This union gives one access to the individual bytes in a long.

```
u.l= 3; u.c[0]= 2;
```

typing One may set up a new “type” for a union:

```
typedef union CL_un CL;
```

The `CL` may now be used where other types are used.

declaration One may declare a variable to be a union instance:

```
union CL_un u;
CL u;           // (assuming that a typedef was used)
```

Modifiers

bitfields used to access individual groups of bits

Example 10 *a structure using bitfields:*

```
struct Bit_str {
    int b1:1;
    int b2:7;
    int b3:8;
    } ;
```

extern indicates that the variable is only being defined; its allocation has been done in another file.

register a suggestion to the compiler that the associated variable should take over a machine register for improved speed. Usually such variables must be either char or (short) int, and should not be a function parameter.

static the value of the variable should be kept for the duration of the program's run. It also restricts scope to the local function or file.

Modifiers

const used most often for function arguments, this tells the compiler that its value will not change. Allows for some additional error checking and optimization.

volatile tells the compiler that the variable can be changed at any time, not only by the program itself, but via interrupts, etc.

typedef used to define new types.

Example 11 *a structure using bitfields:*

```
typedef struct Animal_str Animal;
Animal animal;
```

The `animal` *could have been allocated via* `struct Animal_str animal;`

Problem 3 *Write a file with the following definitions and declarations:*

1. *define x as an int.*
2. *define px as a pointer to int.*
3. *define a new type called plant using struct* `Plant_str`.
4. *define an array of 5 doubles.*

Problem 4 *Use cdecl to analyze your types from Prob2.*

Casts

- One can make C "think" that a variable of one type is actually of another type.
- This feature is most often used with the memory allocation functions such as `malloc` (see notes06) which allocate memory for your use of type "`void *`".
- The type "`void *`" is something of an anonymous type. (meaning: pointer to void),
- Casts are written like types inside parentheses:

```
(int)
(int *)
(int (*)(int))
```

The last cast is a pointer to function (int) returning int

Scope: global

There are four scopes in C:

global These variables are accessible from anywhere in a program

Scope: global

There are four scopes in C:

global These variables are accessible from anywhere in a program

- These variables are declared outside of functions.

Scope: global

There are four scopes in C:

global These variables are accessible from anywhere in a program

- These variables are declared outside of functions.
- They may be made available by using "extern" in other files.

Scope: global

There are four scopes in C:

global These variables are accessible from anywhere in a program

- These variables are declared outside of functions.
- They may be made available by using "extern" in other files.
- They may be initialized only in the file where they’re declared

Scope: global

There are four scopes in C:

global These variables are accessible from anywhere in a program

- These variables are declared outside of functions.
- They may be made available by using "extern" in other files.
- They may be initialized only in the file where they're declared

Example 12

defined *variable, function, etc is allocated or code provided*

declared *variable, function, etc typing information is provided.*

```
/* file1.c: x is defined and declared this way */
int x= 1;

/* file2.c: x is made available this way (but NOT defined) */
extern int x;
```

Scope: file

file These variables are accessible only from within a file

Scope: file

file These variables are accessible only from within a file

- These variables are declared outside of functions with the prefix "static".

Scope: file

file These variables are accessible only from within a file

- These variables are declared outside of functions with the prefix "static".
- They may be initialized.

Scope: file

file These variables are accessible only from within a file

- These variables are declared outside of functions with the prefix "static".
- They may be initialized.
- They are accessible from the point of declaration to the end-of-file

Scope: file

file These variables are accessible only from within a file

- These variables are declared outside of functions with the prefix "static".
- They may be initialized.
- They are accessible from the point of declaration to the end-of-file

Example 17 *(assumed to be at start of file)*

```
/* file1.c: note the use of "static" */
static int x= 2;
```

Scope: function

function These variables are accessible only from within a function

Scope: function

function These variables are accessible only from within a function

- These variables are declared inside the function body or in their argument lists

Scope: function

function These variables are accessible only from within a function

- These variables are declared inside the function body or in their argument lists
- These variables may be initialized (and initialization occurs each time the function is entered).

Scope: function

function These variables are accessible only from within a function

- These variables are declared inside the function body or in their argument lists
- These variables may be initialized (and initialization occurs each time the function is entered).
- Prefixing a function-scope variable with "static" makes the variable have static lifetime – ie. there's one copy, albeit accessible only while inside that function.

Scope: function

function These variables are accessible only from within a function

- These variables are declared inside the function body or in their argument lists
- These variables may be initialized (and initialization occurs each time the function is entered).
- Prefixing a function-scope variable with “static” makes the variable have static lifetime – ie. there’s one copy, albeit accessible only while inside that function.
- Otherwise every function invocation will have its own copy of function scope variables.

Scope: function

function These variables are accessible only from within a function

- These variables are declared inside the function body or in their argument lists
- These variables may be initialized (and initialization occurs each time the function is entered).
- Prefixing a function-scope variable with "static" makes the variable have static lifetime – ie. there's one copy, albeit accessible only while inside that function.
- Otherwise every function invocation will have its own copy of function scope variables.

Example 23

```
/* file1.c: note the use of "static" */
int f(double x)
{
static double y=0.;
y+= sin(x);
return (int) y+y;
}
```

Scope: local

function These variables are accessible only from within curly brace block

Scope: local

function These variables are accessible only from within curly brace block

- May be initialized

Scope: local

function These variables are accessible only from within curly brace block

- May be initialized
- May use same name as global/file/function scope variables, but will take precedence

Scope: local

function These variables are accessible only from within curly brace block

- May be initialized
- May use same name as global/file/function scope variables, but will take precedence

Example 27

```
int f(double x)
{
double y= 0;
    {
    double y= 1.;
    y= sin(x);
    }
return (int) y+x;
}
```

Scope: local

function These variables are accessible only from within curly brace block

- May be initialized
- May use same name as global/file/function scope variables, but will take precedence

Example 28

```
int f(double x)
{
double y= 0;
    {
    double y= 1.;
    y= sin(x);
    }
return (int) y+x;
}
```

Problem 9 *Determine what the value of* f(.4) *as defined above returns by writing a short program*

Formatted Printing

There is an entire family of `printf()` functions which are used to generate output to the screen, to files, and even to other variables.

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int vfprintf(FIlE *stream, const char *format, va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format, va_list arg);
```

fprintf	print to file
printf	print to display
sprintf	print to variable
vfprintf	print vararg list to file
vprintf	print vararg list to display
vsprintf	print vararg list to variable

Formatted Printing, con't.

- FILE is a type set up by <stdio.h>, the standard i/o header file. One may generate a FILE* by using pre-defined ones such as stdout or stderr, or use fopen() to open a file.

Formatted Printing, con't.

- FILE is a type set up by <stdio.h>, the standard i/o header file. One may generate a FILE* by using pre-defined ones such as stdout or stderr, or use fopen() to open a file.
- The sprintf() and vsprintf() functions both generate formatted strings into variables that the program itself may use later.

Formatted Printing, con't.

- FILE is a type set up by <stdio.h>, the standard i/o header file. One may generate a FILE* by using pre-defined ones such as stdout or stderr, or use fopen() to open a file.
- The sprintf() and vsprintf() functions both generate formatted strings into variables that the program itself may use later.
- The format string is a list of bytes that are interpreted literally, contain format specification substrings, or special characters (\n == newline).

General Form of a Format Specification String

%[flags][width][.precision]type

Format Specifier Types

d,i	signed decimal integers and longs
o	signed octal integers
u	unsigned decimal integers and longs
x,X	unsigned hexadecimal
f	signed floating point
e,E	signed floating point using scientific (exponent) notation
g,G	signed float of either e or f style
c	single character
s	string (an array of characters terminated by a nullbyte)
%%	prints a percent sign
n	number of characters output so far
p	prints pointer address

Formatted Printing Examples

%[flags][width][.precision]type

Example 29 *printf("%f\n",x); – prints x in floating format*

Formatted Printing Examples

%[flags][width][.precision]type

Example 33 *printf("%f\n",x); – prints x in floating format*

Example 34 *printf("%10.3f\n",x); – prints x to take up ten spaces with 3 digits to the right of the decimal point*

Formatted Printing Examples

%[flags][width][.precision]type

Example 37 *printf("%f\n",x); – prints x in floating format*

Example 38 *printf("%10.3f\n",x); – prints x to take up ten spaces with 3 digits to the right of the decimal point*

Example 39 *printf("%10s\n",s); – prints a string from variable s*

Formatted Printing Examples

%[flags][width][.precision]type

Example 41 *printf("%f\n",x); – prints x in floating format*

Example 42 *printf("%10.3f\n",x); – prints x to take up ten spaces with 3 digits to the right of the decimal point*

Example 43 *printf("%10s\n",s); – prints a string from variable s*

Example 44 *printf("%10.3e\n",x); – prints x in exponential format (width=10, 3 digits)*

Format Specification Flags

%[flags][width][.precision]type

-	causes left justification with blank padding

Format Specification Flags

%[flags][width][.precision]type

-	causes left justification with blank padding
0	use zero padding

Format Specification Flags

%[flags][width][.precision]type

-	causes left justification with blank padding
0	use zero padding
+	output will always begin with either + or -

Format Specification Flags

%[flags][width][.precision]type

-	causes left justification with blank padding
0	use zero padding
+	output will always begin with either + or -
blank	positive values begin with blank

Format Specification Flags

%[flags][width][.precision]type

-	causes left justification with blank padding
0	use zero padding
+	output will always begin with either + or -
blank	positive values begin with blank
*	width specifier is supplied by the argument list

Format Specification Flags

%[flags][width][.precision]type

-	causes left justification with blank padding
0	use zero padding
+	output will always begin with either + or -
blank	positive values begin with blank
*	width specifier is supplied by the argument list
l	argument of type long int

Problems

Problem 10 *Given the following declarations:*

```
float  f   = 1.0;
double d   = 2.3e4;
char   s[] = "abcdef";
int    i   = 5;
char   x   = 'z';
```

A basic program that will compile (and not do anything) looks like:

```
#include <stdio.h>
int main()
{
}
```

Change the program above to include the declarations as given and include printf()s to display the information, one per line.

Problem 11 *Write a program to display the string* `"abc"` *twice using the following format:* `"<%10s><%-10s>\n"`.

Expressions

C has a rich complement of operators; see `cprimer.pdf`, page 8, for a list.

Example 45 *Bit shifting and bitwise-or'ing*

```
/* bitrev: this function reverses the bit sequence of the input integer */
unsigned long bitrev(unsigned long x)
{
  /* Reverse the bits of x (32 total). */
  x = ((x & 0xAAAAAAAA) >>  1) | ((x & 0x55555555) <<  1);  //  1 bit pair  swapping
  x = ((x & 0xCCCCCCCC) >>  2) | ((x & 0x33333333) <<  2);  //  2 bits pair swapping
  x = ((x & 0xF0F0F0F0) >>  4) | ((x & 0x0F0F0F0F) <<  4);  //  4 bits pair swapping
  x = ((x & 0xFF00FF00) >>  8) | ((x & 0x00FF00FF) <<  8);  //  8 bits pair swapping
  x = ((x & 0xFFFF0000) >> 16) | ((x & 0x0000FFFF) << 16);  // 16 bits pair swapping
  return x;
}
```

Expressions: Math Library

Example 46 *Computing the secant of x;*

To compile, use `cc -c sec.c`*.*

To link to programs using the math library, use `-lm`*.*

```
/* sec.c: secant(x) = 1/cos(x)) */
#include <stdio.h>
#include <math.h>
double sec(double x)
{
  return 1./cos(x);
}
```

Problems

Problem 12 *Write a short program that opens a file called* <`data`> *with a number on each line, reads it with* `fscanf(fp,...)`*, and computes the average.* *Hints:*

```
FILE *fp;
fp= fopen("filename","r");
...
fscanf(fp,"%lf",&x);
...
```

Problem 13 *Write a short program that computes Einstein's famous equality:* $E = mc^2$.

Hints:

```
c=2.9979246e+08 m / s
m in kg
E in Joules (kg m^2 / s^2)
```

Use it to convert into Joules:

```
1kg
1 electron mass (9.1093897e-31 kg)
1 proton mass   (1.6726231e-27 kg)
1 planckmass    (2.17671e-8 kg)
```

Choices: if-then-else

C supports if-then-else (without a "then" :)...

```
if(expr) {
    ..list-of-statements..
    }
else if(expr) {
    ..list-of-statements..
    }
..repeat-as-needed..
else {
    ..list-of-statements..
    }
```

The `else if(){}` and `else{}` sections are optional. C will evaluate each if-expression to see if any of them evaluate to not-zero; others which follow are skipped.

Choices: switch-case

C supports switch-case

```
switch(expr) {
case 1:
    break;
case '2':
    break;
default:
    break;
    }
```

The switch expression must evaluate to a char/integer/enumerated type, and each of the cases must be one of the same types: char/integer/enumerated type. In other words, `case "abc"`: isn't legal. C will pass control to the matching case. Note that unless the `"break"` statement is included *C will fall through following cases*. If no case matches, the `default:` case will match.

Unlike Java: C's cases are always integers!

Problems

Problem 14 *Write a program using if-else if-else which determines what grade should be assigned to a score, assuming the standard grade-school standard of 90+=A, 80-89=B, etc.*

Hints:

```
int main(int argc,char **argv)
{
int score;
sscanf(argv[1],"%d",&score);
...
}
```

Problem 15 *Modify the program above to use switch-case.*

Hints:

```
int main(int argc,char **argv)
{
char tenscore;
char unitscore;

if(!strcmp("100",argv[1])) ...
tenscore = argv[1][0];
unitscore= argv[1][1];
...
}
```

Loops

```
while(expr) {
  ...
}
```

Executes while the expression evaluates to true (ie. not zero).

```
do {
  ...
} while(expr);
```

Do the following statements and continue doing them while the expression evalutes to true.

```
for(initialize;test;update){
  ...
}
```

The initialize expression can do several things (ex. x=y=0, z=1), and will be executed once.
The test expression is then evaluated; if true, the main body of the loop gets executed.
Each time thereafter, the update expression (which also can do several things) is executed and the test is repeated.

Problems

Problem 16 *Write a program that computes 10! using a while() loop*

Problem 17 *Write a program that computes 10! using a do-while() loop*

Problem 18 *Write a program that computes 10! using a for() loop*

Functions

```
type function_name ( [argument[,argument...]] )
{
..list-of-statements..
}
```

Every function has a type that it returns; if that return type is "void" then the function doesn't really return anything.

To return values use the "return" statement:

```
return [expr];
```

The Main Function

Example 47 *Every C program must have a main() function:*
(it provides a method to get information from the command line)

```
int main(int argc,char **argv)
{
}
```

argc contains a count of arguments on the command line, including the command (program name) itself.

argv is an array of strings

argv[0]	holds first string
argv[1]	holds second string
...	
argv[argc-1]	holds last string
argv[argc]	is NULL (zero)

Example of a Function

Example 48 *Computing* $sin(x) + cos(x)$

```
#include <stdio.h>
#include <math.h>
double sinpcos(double x)
{
  return sin(x) + cos(x);
}
```

Example 49 *Using sinpcos(x):*

```
double z;
z= sinpcos((3.14159265/180.)*30.)
```

Problems

Problem 19 *Write a program which prints out a list of the arguments fed to it, plus a count thereof:*

```
cnt: argument
```

Problem 20 *Write a function which computes the factorial of an integer. Write a main() function which interprets its arguments, one at a time, uses that function to compute the factorial, then prints out each result.*

Memory Handling

C supports recursion; that is, a function can call itself.

Problem 21 *write a function to compute factorials using recursion*

- The memory so used comes from the “stack”
- The instruction pointer where the call is made is first pushed onto the stack,
- Then the call’s arguments,
- And then the function’s local values are “pushed” onto a stack.
- Upon return, the function pops the stack clean of its arguments and local values,
- Pushes any return value, and pops and returns to the instruction pointer location.

Another section of memory holds your `global` and `static` variables.

Memory Allocation: malloc

Finally, yet another section of memory, the "**heap**", holds memory that your program may allocate and de-allocate as its running.

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr,size_t size);
void *calloc(size_t nelem,size_t elsize);
```

malloc allocates a chunk of memory of the specified size

Example 50 *This example allocates ten bytes*

```
char *s;
s= (char *) malloc((size_t) 10);
strcpy(s,"abc");
```

Memory Allocation: free

`free()` will free memory allocated by `malloc`, `realloc`, or `calloc`; just pass a pointer to the memory to the function.

Example 51 `free(s); s= NULL;`

Use Defensive Programming!!!

- Whenever you free a pointer, set it to NULL! This way an invalid access will be reliably caught.
- Put in lots of sanity checks! An errant pointer problem may get caught early with sanity checks. *The earlier caught the better!*

Memory Allocation: calloc

`calloc()` takes “`nelem`", the number of elements, each of size “`size`”, and thus allocates `nelem*size` bytes.

Example 52

```
struct ABC_str {
    double d;
    float  f;
    struct ABC_str *prv;
    struct ABC_str *nxt;
    } *abc;
abc= (struct ABC_str *) calloc((size_t) 10,sizeof(struct ABC_str));
```

Note how, in the example above, the `sizeof()` macro is used to compute the size of a structure so that the programmer need not do so him/herself.

The memory is cleared.

Memory Allocation: realloc

- The `realloc()` function lets the programmer re-allocate memory, retaining as many bytes from the original memory as possible.
- The new memory will be `size` bytes long

Problem

Write a program which will save an arbitrary number of strings and write them out in reverse order. Assume no string is greater than 5000 bytes long.

Hints:

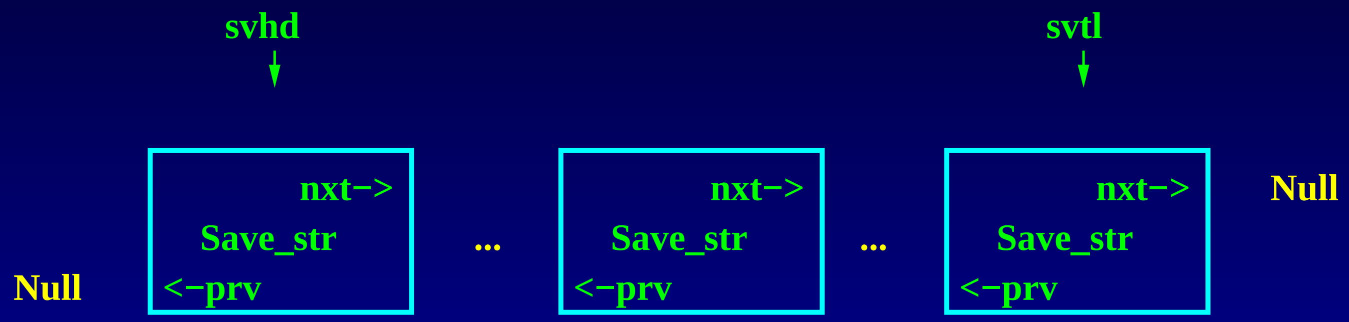
```
#define BUFSIZE 5001
FILE *fp;
char *b;
struct Save_str {
    char *b;
    struct Save_str *nxt;
    struct Save_str *prv;
    } *svhd= NULL, *svtl= NULL;

fp= fopen("filename","r");
while(fgets(buf,BUFSIZE,fp)) {}
save   = (struct Save_str *) malloc(sizeof(struct Save_str));
save->b= calloc(strlen(buf)+1,sizeof(char));
strcpy(save->b,buf);
```

Save strings in a double-linked list. (see the next slide)

Double Linked Lists

The idea here is to have the structure contain a pointer to the next and preceding items in a *doubly linked list*.



Each `Save_str` points to the previous and next `Save_str`.
The first `prv` pointer and the last `nxt` pointer point to NULL, thereby demarcating the beginning and ending of the linked list.

`svhd` points to the first `Save_str` on the list.
`svtl` points to the last `Save_str` on the list.

Debugging

Symbolic Debugger: a program which allows one to debug another program's performance by providing access to its variables and control over instruction execution.

Debugging

Symbolic Debugger: a program which allows one to debug another program's performance by providing access to its variables and control over instruction execution.

core dump: these generally occur because a program attempts to access memory it doesn't own.

Debugging

Symbolic Debugger: a program which allows one to debug another program's performance by providing access to its variables and control over instruction execution.

core dump: these generally occur because a program attempts to access memory it doesn't own.

- A *core dump* is a snapshot of the program's memory and state at program termination.

Debugging

Symbolic Debugger: a program which allows one to debug another program's performance by providing access to its variables and control over instruction execution.

core dump: these generally occur because a program attempts to access memory it doesn't own.

- A *core dump* is a snapshot of the program's memory and state at program termination.
- Frequent cause: a pointer problem. The bad pointer

Debugging

Symbolic Debugger: a program which allows one to debug another program's performance by providing access to its variables and control over instruction execution.

core dump: these generally occur because a program attempts to access memory it doesn't own.

- A *core dump* is a snapshot of the program's memory and state at program termination.
- Frequent cause: a pointer problem. The bad pointer
  - points to memory that hasn't been allocated, often due to not having been initialized *(see ex1.c, ex2.c)*

Debugging

Symbolic Debugger: a program which allows one to debug another program's performance by providing access to its variables and control over instruction execution.

core dump: these generally occur because a program attempts to access memory it doesn't own.

- A *core dump* is a snapshot of the program's memory and state at program termination.
- Frequent cause: a pointer problem. The bad pointer
  - points to memory that hasn't been allocated, often due to not having been initialized *(see ex1.c, ex2.c)*
  - points to memory that has been freed already *(see ex3.c)*

Debugging

Symbolic Debugger: a program which allows one to debug another program's performance by providing access to its variables and control over instruction execution.

core dump: these generally occur because a program attempts to access memory it doesn't own.

- A *core dump* is a snapshot of the program's memory and state at program termination.
- Frequent cause: a pointer problem. The bad pointer
  - points to memory that hasn't been allocated, often due to not having been initialized *(see ex1.c, ex2.c)*
  - points to memory that has been freed already *(see ex3.c)*
  - the pointer has been overwritten (yielding a "pointer" to who-knows-where)

Debugging

Symbolic Debugger: a program which allows one to debug another program's performance by providing access to its variables and control over instruction execution.
core dump: these generally occur because a program attempts to access memory it doesn't own.

- A *core dump* is a snapshot of the program's memory and state at program termination.
- Frequent cause: a pointer problem. The bad pointer
  - points to memory that hasn't been allocated, often due to not having been initialized *(see ex1.c, ex2.c)*
  - points to memory that has been freed already *(see ex3.c)*
  - the pointer has been overwritten (yielding a "pointer" to who-knows-where)
- May I suggest that you use the -Wall option to gcc to help you catch a lot of simple problems.

Beginning gdb

- Compile and run *ex2.c*:

  `gcc -g ex2.c -o ex2` (*-g: retain symbol table*)

  `ex2` will result in a core dump

```
gdb ex2 core
#0 (some hex address) in main(argc=1,argv=0x...) at ex2.c:25
25 printf("pi[0]=%d\n",pi[0]);
(gdb) p pi
$1 = (int *) 0x0
(gdb) quit
```

It is always illegal to dereference a null pointer, and doing so will result in a core dump.

`p [/FMT] expr` prints an expression. `FMT` is a format character:

o=octal	t=binary	i=instruction
x=hex	f=float	c=char
d=decimal	a=address	s=string
u=unsigned decimal		

Core Dumps, reprise

- For debugging, compile with the -g flag. This option tells the compiler to retain symbolic information (such as variable names, argument names, etc)

Core Dumps, reprise

- For debugging, compile with the -g flag. This option tells the compiler to retain symbolic information (such as variable names, argument names, etc)
- Segmentation faults occur when a program attempts to access memory it has no rights to.

Core Dumps, reprise

- For debugging, compile with the -g flag. This option tells the compiler to retain symbolic information (such as variable names, argument names, etc)
- Segmentation faults occur when a program attempts to access memory it has no rights to.
- Memory is obtained from the o/s in “segments”, which are typically 2K-4K. Consequently, your program may have access rights to some memory that it hasn’t allocated yet, and so incorrect accesses to that memory does not cause a core dump. Use efence to catch these problems early. (see later slide)

GDB: list-where-printf-trace

(LFFF=line number, function, filename:linenum, filename:function)

list one may use list to see ten lines around the current address.

list LFFF

specify a place to give a ten-line listing of source

GDB: list-where-printf-trace

(LFFF=line number, function, filename:linenum, filename:function)

list one may use list to see ten lines around the current address.

list LFFF

specify a place to give a ten-line listing of source

where print a backtrace of all stack frames

GDB: list-where-printf-trace

(LFFF=line number, function, filename:linenum, filename:function)

list one may use list to see ten lines around the current address.

list LFFF

specify a place to give a ten-line listing of source

where print a backtrace of all stack frames

More Printing

printf useful for displaying user-defined formatted output

printf "formatstring", arg_1, arg_2, $...$

GDB: list-where-printf-trace

(LFFF=line number, function, filename:linenum, filename:function)

list one may use list to see ten lines around the current address.

list LFFF

specify a place to give a ten-line listing of source

where print a backtrace of all stack frames

More Printing

printf useful for displaying user-defined formatted output

printf "formatstring", arg_1, arg_2, $...$

Stack Examination

backtrace print a trace of stack frames

GDB: list-where-printf-trace

(LFFF=line number, function, filename:linenum, filename:function)

list one may use list to see ten lines around the current address.

list LFFF

specify a place to give a ten-line listing of source

where print a backtrace of all stack frames

More Printing

printf useful for displaying user-defined formatted output

printf "formatstring", arg_1, arg_2, $...$

Stack Examination

backtrace print a trace of stack frames

up/down move current address up and down the call stack

GDB: Running, Breakpoints, Watchpoints

run arg1 arg2 ...	start executing program with given arguments

GDB: Running, Breakpoints, Watchpoints

run arg1 arg2 ...	start executing program with given arguments
b LFFF	breakpoints stop the program at the given point. One may then continue, stop, examine variables, etc.

GDB: Running, Breakpoints, Watchpoints

run arg1 arg2 ...	start executing program with given arguments
b LFFF	breakpoints stop the program at the given point. One may then continue, stop, examine variables, etc.
clear LFFF	clear breakpoint from location

GDB: Running, Breakpoints, Watchpoints

run arg1 arg2 ...	start executing program with given arguments
b LFFF	breakpoints stop the program at the given point. One may then continue, stop, examine variables, etc.
clear LFFF	clear breakpoint from location
delete #	delete #th breakpoint

GDB: Running, Breakpoints, Watchpoints

run arg1 arg2 ...	start executing program with given arguments
b LFFF	breakpoints stop the program at the given point. One may then continue, stop, examine variables, etc.
clear LFFF	clear breakpoint from location
delete #	delete # th breakpoint
info breakpoints	print a table of current breakpoints

GDB: Running, Breakpoints, Watchpoints

run arg1 arg2 ...	start executing program with given arguments
b LFFF	breakpoints stop the program at the given point. One may then continue, stop, examine variables, etc.
clear LFFF	clear breakpoint from location
delete #	delete # th breakpoint
info breakpoints	print a table of current breakpoints
watch EXPR	set a *watchpoint* for an expression. Whenever the expression changes value, the program will be stopped.

GDB: Running, Breakpoints, Watchpoints

run arg1 arg2 ...	start executing program with given arguments
b LFFF	breakpoints stop the program at the given point. One may then continue, stop, examine variables, etc.
clear LFFF	clear breakpoint from location
delete #	delete # th breakpoint
info breakpoints	print a table of current breakpoints
watch EXPR	set a *watchpoint* for an expression. Whenever the expression changes value, the program will be stopped.
rwatch EXPR	stops program when EXPR is read by program

GDB: Running, Breakpoints, Watchpoints

run arg1 arg2 ...	start executing program with given arguments
b LFFF	breakpoints stop the program at the given point. One may then continue, stop, examine variables, etc.
clear LFFF	clear breakpoint from location
delete #	delete # th breakpoint
info breakpoints	print a table of current breakpoints
watch EXPR	set a *watchpoint* for an expression. Whenever the expression changes value, the program will be stopped.
rwatch EXPR	stops program when EXPR is read by program
info watchpoints	print a table on current watchpoints

GDB: Running, Breakpoints, Watchpoints

run arg1 arg2 ...	start executing program with given arguments
b LFFF	breakpoints stop the program at the given point. One may then continue, stop, examine variables, etc.
clear LFFF	clear breakpoint from location
delete #	delete # th breakpoint
info breakpoints	print a table of current breakpoints
watch EXPR	set a *watchpoint* for an expression. Whenever the expression changes value, the program will be stopped.
rwatch EXPR	stops program when EXPR is read by program
info watchpoints	print a table on current watchpoints
c [IGNORECOUNT]	resume execution, optionally ignoring up to IGNORECOUNT breaks at that address

GDB: Stepping, Next, Finish, Until

s [COUNT]	step through instructions to next/COUNT-th source line

GDB: Stepping, Next, Finish, Until

s [COUNT]	step through instructions to next/COUNT-th source line
next [COUNT]	step to next line in same stack frame (ie. step through function calls)

GDB: Stepping, Next, Finish, Until

s [COUNT]	step through instructions to next/COUNT-th source line
next [COUNT]	step to next line in same stack frame (ie. step through function calls)
finish	continue running until the current frame returns

GDB: Stepping, Next, Finish, Until

s [COUNT]	step through instructions to next/COUNT-th source line
next [COUNT]	step to next line in same stack frame (ie. step through function calls)
finish	continue running until the current frame returns
until	after one has stepped to the bottom of a loop, continue until an address > current address is reached

Efence

- Core dumps are extremely useful - they enable you to find where invalid memory accesses occur.

Efence

- Core dumps are extremely useful - they enable you to find where invalid memory accesses occur.
- Unfortunately, not all improper memory accesses are invalid if they occur in a memory segment that the program holds.

Efence

- Core dumps are extremely useful - they enable you to find where invalid memory accesses occur.
- Unfortunately, not all improper memory accesses are invalid if they occur in a memory segment that the program holds.
- Bruce Perens of Pixar wrote a library called “efence” which he has graciously made freely re-distributable and available. It helps with detection of common and difficult to find memory related bugs at the price of being slow and a memory hog.

Efence

- Core dumps are extremely useful - they enable you to find where invalid memory accesses occur.
- Unfortunately, not all improper memory accesses are invalid if they occur in a memory segment that the program holds.
- Bruce Perens of Pixar wrote a library called “efence” which he has graciously made freely re-distributable and available. It helps with detection of common and difficult to find memory related bugs at the price of being slow and a memory hog.
- Efence uses the virtual memory hardware to place an inaccessible memory page immediately after every memory allocation; reading or writing outside memory allocations will cause a seg-fault.

Efence

- Core dumps are extremely useful - they enable you to find where invalid memory accesses occur.
- Unfortunately, not all improper memory accesses are invalid if they occur in a memory segment that the program holds.
- Bruce Perens of Pixar wrote a library called “efence” which he has graciously made freely re-distributable and available. It helps with detection of common and difficult to find memory related bugs at the price of being slow and a memory hog.
- Efence uses the virtual memory hardware to place an inaccessible memory page immediately after every memory allocation; reading or writing outside memory allocations will cause a seg-fault.
- Overrunning a memory allocation and attempting to use free’d memory also cause immediate core dumps

Efence

- Core dumps are extremely useful - they enable you to find where invalid memory accesses occur.
- Unfortunately, not all improper memory accesses are invalid if they occur in a memory segment that the program holds.
- Bruce Perens of Pixar wrote a library called “efence” which he has graciously made freely re-distributable and available. It helps with detection of common and difficult to find memory related bugs at the price of being slow and a memory hog.
- Efence uses the virtual memory hardware to place an inaccessible memory page immediately after every memory allocation; reading or writing outside memory allocations will cause a seg-fault.
- Overrunning a memory allocation and attempting to use free’d memory also cause immediate core dumps
- All you need to do is link to the library: `cc -g ... -lefence -o ...`

Instrumenting Your Code

Method 1 put in printf()s as needed, and delete them later

Instrumenting Your Code

Method 1 put in printf()s as needed, and delete them later

Method 2 use the preprocessor

```
#define DEBUG
...
#ifdef DEBUG
...printf()...
#endif
```

Instrumenting Your Code

Method 1 put in printf()s as needed, and delete them later

Method 2 use the preprocessor

```
#define DEBUG
...
#ifdef DEBUG
...printf()...
#endif
```

(one may also use cc -DDEBUG rather than #define DEBUG)

Instrumenting Your Code

Method 1 put in printf()s as needed, and delete them later

Method 2 use the preprocessor

```
#define DEBUG
...
#ifdef DEBUG
...printf()...
#endif
```

(one may also use cc -DDEBUG rather than #define DEBUG)

Example 53 *of using #ifdef DEBUG...#endif*

```
#ifdef DEBUG
printf("x=%5.2lf\n",x);
#endif
```

Instrumenting Your Code

Method 3 Printf(()) macros

```
#ifdef DEBUG
# define Printf(x) printf x
#else
# define Printf(x)
#endif
```

Then Printf((x)); is conditionally compiled in when DEBUG is defined.

Instrumenting Your Code

Method 3 Printf(()) macros

```
#ifdef DEBUG
# define Printf(x) printf x
#else
# define Printf(x)
#endif
```

Then `Printf((x));` is conditionally compiled in when `DEBUG` is defined.

Example 54 *Printf(("x=%5.2lf",x));*

Instrumenting Your Code

Method 4 Use a function-tracking internal debugger

(see ex6.c)

Instrumenting Your Code

Method 4 Use a function-tracking internal debugger

- You may get it at http://www.drchip.org/drchipdbg

  Then gunzip drchipdbg.tar.gz

  tar -oxvf drchipdbg.tar

  make

  This produces a library, xtdio.a.

(see ex6.c)

Instrumenting Your Code

Method 4 Use a function-tracking internal debugger

- You may get it at http://www.drchip.org/drchipdbg

  Then gunzip drchipdbg.tar.gz

  tar -oxvf drchipdbg.tar

  make

  This produces a library, xtdio.a.

- Functions provided:

(see ex6.c)

Instrumenting Your Code

Method 4 Use a function-tracking internal debugger

- You may get it at http://www.drchip.org/drchipdbg

  Then gunzip drchipdbg.tar.gz

  tar -oxvf drchipdbg.tar

  make

  This produces a library, xtdio.a.

- Functions provided:

  Initdbg((&argc,argv)); initialize the debugger in your main(int argc,char **argv) function

(see ex6.c)

Instrumenting Your Code

Method 4 Use a function-tracking internal debugger

- You may get it at http://www.drchip.org/drchipdbg

  Then gunzip drchipdbg.tar.gz

  tar -oxvf drchipdbg.tar

  make

  This produces a library, xtdio.a.

- Functions provided:

  Initdbg((&argc,argv)); initialize the debugger in your main(int argc,char **argv) function

  Edbg((”funcname(fmt)”,args)); at the start of every function, put in an Edbg call.

(see ex6.c)

Instrumenting Your Code

Method 4 Use a function-tracking internal debugger

- You may get it at http://www.drchip.org/drchipdbg

  Then gunzip drchipdbg.tar.gz

  tar -oxvf drchipdbg.tar

  make

  This produces a library, xtdio.a.

- Functions provided:

  Initdbg((&argc,argv)); initialize the debugger in your main(int argc,char **argv) function

  Edbg((”funcname(fmt)”,args)); at the start of every function, put in an Edbg call.

  Rdbg((”funcname retval”)); just before every return, put in a Rdbg call

(see ex6.c)

Instrumenting Your Code

Method 4 Use a function-tracking internal debugger

- You may get it at `http://www.drchip.org/drchipdbg`

  Then `gunzip drchipdbg.tar.gz`

  `tar -oxvf drchipdbg.tar`

  `make`

  This produces a library, `xtdio.a`.

- Functions provided:

  Initdbg((&argc,argv)); initialize the debugger in your `main(int argc,char **argv)` function

  Edbg((”funcname(fmt)”,args)); at the start of every function, put in an `Edbg` call.

  Rdbg((”funcname retval”)); just before every return, put in a `Rdbg` call

  Dprintf((detaillevel,’fmt’,args)); print output at given detail level or greater

(see ex6.c)

Instrumenting Your Code

- The internal debugger supports custom instrumentation of your code

Instrumenting Your Code

- The internal debugger supports custom instrumentation of your code
- Your C/C++ files should `#include "dbg.h"`

Instrumenting Your Code

- The internal debugger supports custom instrumentation of your code
- Your C/C++ files should `#include "dbg.h"`
- Use `-DDEBUG` on the compile command (ex. `cc -DDEBUG -c myfile.c`) to enable debugging; when you compile without the `-DDEBUG`, its compiles as if you had no debugging statements

Instrumenting Your Code

- The internal debugger supports custom instrumentation of your code
- Your C/C++ files should `#include "dbg.h"`
- Use `-DDEBUG` on the compile command (ex. `cc -DDEBUG -c myfile.c`) to enable debugging; when you compile without the `-DDEBUG`, its compiles as if you had no debugging statements
- Typically,

```
#include "dbg.h"
int main(int argc,char **argv)
{
Initdbg((&argc,argv));
...
Rdbg(("main 0"));
return 0;
}
```

Using the Internal Debugger

The internal debugger takes commands following -@:

```
yourpgm anyargs -@ dbgcmd
yourpgm anyargs -@ filename_containing_dbgcmds
```

Using the Internal Debugger

The internal debugger takes commands following -@:

```
yourpgm anyargs -@ dbgcmd
yourpgm anyargs -@ filename_containing_dbgcmds
```

After Initdbg((&argc,argv)), argc is adjusted so that the rest of the program doesn't know about the debugging arguments.

Using the Internal Debugger

The internal debugger takes commands following -@:

```
yourpgm anyargs -@ dbgcmd
yourpgm anyargs -@ filename_containing_dbgcmds
```

After Initdbg((&argc,argv)), argc is adjusted so that the rest of the program doesn't know about the debugging arguments.

General format for an internal debugger command:

```
{on|off} funcname [*] [detaillvl] [:dtst1 [:dtst2]] [preqlist] [>file]
```

More Debugging Tools

valgrind This program works with executables. With it you can find uninitialized variable issues, memory leaks, lost/unreachable blocks, and more

strace Displays system calls used by a process and signals received by a process.

gprof displays a call graph profile; use this with the `-pg` option `gcc` provides.

tcpdump a network packet analyzer

vmstat virtual memory statistics

ps process status program (use `ps -f -u username`, for example)

free display memory usage (free, used, shared, etc)

pmap display the memory map of a running process (`pmap pid`; use `ps` to find the process id you wish to check upon)

lsof list open files on your system (ex. `lsof -u userid`)

Make

- Its a nuisance to build projects, especially multiple file projects, with shell scripts.
  Problem: scripts tend to re-build every object file every time. Even when only one file was updated and needs re-compiling.
  Problem: a header file change may require a number of files to be recompiled
- Solution: the `make` program. The `make` program uses a file, `makefile`, to tell `make` which files depend on which files, and how to build files based on that dependency.
- typical `makefile`:

```
abc : abc.o
      cc abc.o -o abc
```

This `makefile` says: `abc` depends upon the file `abc.o`; if `abc.o` is newer than `abc`, then re-build `abc` by executing `cc abc.o -o abc`.

Make, con't.

- Multiple lines of commands (all preceded by a <tab>) may follow the target : dependency rule.
- `make` has built-in general rules; these tell `make` how to build `.o` files from `.c`, `.f`, etc files, how to build `.c` files from `.l`, `.y` (lex, yacc) files, etc.
- Assuming that `abc.c` exists, but `abc.o` and `abc` don't, our "typical makefile" will result in:

```
cc -c abc.c
cc abc.o -o abc
```

The first command is due to the builtin rule handling `.c`→`.o`.

Structure of a Makefile

```
# Comments may go anywhere
[VARIABLE=VALUE] # and apply to end-of-line
[VARIABLE=VALUE]
    ...
target : pre-requisites [; command]
<tab>command
<tab>command
    ...
```

- Timestamps of targets are compared with those of their pre-requisites
- If any pre-requisite is newer than its target, then those pre-requisites are themselves checked to see if any rules apply.
- After this recursive check, the rule's commands are executed.
- Without any arguments, the first target in a `makefile` is made

Makefile Variables

- *make* allows one to define variables (akin to C's *#define*)

  Example 55 *ABC=/path/to/somewhere*

Makefile Variables

- *make* allows one to define variables (akin to C's *#define*)

  Example 56 *ABC=/path/to/somewhere*

- These variables are string variables, and they may be used in commands, targets, and pre-requisites via any of the following formats:
  $ABC $ABC $(ABC)

Makefile Variables

- *make* allows one to define variables (akin to C's *#define*)

  Example 57 *ABC=/path/to/somewhere*

- These variables are string variables, and they may be used in commands, targets, and pre-requisites via any of the following formats:
  $ABC $ABC $(ABC)
- Special Variables and Targets:

Makefile Variables

- *make* allows one to define variables (akin to C's *#define*)

  Example 58 *ABC=/path/to/somewhere*

- These variables are string variables, and they may be used in commands, targets, and pre-requisites via any of the following formats:
  $ABC $ABC $(ABC)

- Special Variables and Targets:

  VPATH =...colon-delimited-paths...

  Pre-requisites may reside other than in the current directory; they will be searched for with the *VPATH*.

Makefile Variables

- *make* allows one to define variables (akin to C's *#define*)

  Example 59 *ABC=/path/to/somewhere*

- These variables are string variables, and they may be used in commands, targets, and pre-requisites via any of the following formats:
  $ABC $ABC $(ABC)

- Special Variables and Targets:

  VPATH =...colon-delimited-paths...
  Pre-requisites may reside other than in the current directory; they will be searched for with the *VPATH*.

  .SILENT same as using *make -s*; normally commands are echo'd as executed, but the presence of this "variable" will silence that. Use ".SILENT :" *(ie. as a target with no dependencies)*
  One may also silence a specific command by preceding it with an "@"

Makefile Variables

- *make* allows one to define variables (akin to C's *#define*)

  Example 60 *ABC=/path/to/somewhere*

- These variables are string variables, and they may be used in commands, targets, and pre-requisites via any of the following formats:
  $ABC $ABC $(ABC)

- Special Variables and Targets:

  VPATH =...colon-delimited-paths...
  Pre-requisites may reside other than in the current directory; they will be searched for with the *VPATH*.

  .SILENT same as using *make -s*; normally commands are echo'd as executed, but the presence of this "variable" will silence that. Use ".SILENT :" *(ie. as a target with no dependencies)*
  One may also silence a specific command by preceding it with an "@"

  .IGNORE same as using *make -e*; errors will be ignored and execution will continue. Normally *make* will terminate upon error. Use ".IGNORE :" *(ie. as a target with no dependencies)*
  One may ignore errors from a specific command by preceding it with a "-"

Libraries

When one has a group of re-usable functions to link to, it is often preferable to build a *library*. Object files in a library are linked to as needed.

Libraries under Unix are generated by the ar command. (short for "archive")

Libraries

When one has a group of re-usable functions to link to, it is often preferable to build a *library*. Object files in a library are linked to as needed.

Libraries under Unix are generated by the ar command. (short for “archive”)

ar q libname.a *list-of-files* “quick” add files

Libraries

When one has a group of re-usable functions to link to, it is often preferable to build a *library*. Object files in a library are linked to as needed.

Libraries under Unix are generated by the ar command. (short for "archive")

ar q libname.a *list-of-files*	"quick" add files
ar r libname.a *list-of-files*	add files to libname.a; replace any already in libname.a

Libraries

When one has a group of re-usable functions to link to, it is often preferable to build a *library*. Object files in a library are linked to as needed.

Libraries under Unix are generated by the ar command. (short for "archive")

ar q libname.a *list-of-files*	"quick" add files
ar r libname.a *list-of-files*	add files to libname.a; replace any already in libname.a
ar d libname.a *list-of-files*	delete files from libname.a

Libraries

When one has a group of re-usable functions to link to, it is often preferable to build a *library*. Object files in a library are linked to as needed.

Libraries under Unix are generated by the ar command. (short for “archive”)

ar q libname.a *list-of-files*	“quick” add files
ar r libname.a *list-of-files*	add files to libname.a; replace any already in libname.a
ar d libname.a *list-of-files*	delete files from libname.a
ar x libname.a *list-of-files*	extract copy of listed files from libname.a

Libraries

When one has a group of re-usable functions to link to, it is often preferable to build a *library*. Object files in a library are linked to as needed.

Libraries under Unix are generated by the ar command. (short for "archive")

ar q libname.a *list-of-files*	"quick" add files
ar r libname.a *list-of-files*	add files to libname.a; replace any already in libname.a
ar d libname.a *list-of-files*	delete files from libname.a
ar x libname.a *list-of-files*	extract copy of listed files from libname.a

some Unix systems need to follow the archiver ar with ranlib:

Libraries

When one has a group of re-usable functions to link to, it is often preferable to build a *library*. Object files in a library are linked to as needed.

Libraries under Unix are generated by the ar command. (short for "archive")

ar q libname.a *list-of-files*	"quick" add files
ar r libname.a *list-of-files*	add files to libname.a; replace any already in libname.a
ar d libname.a *list-of-files*	delete files from libname.a
ar x libname.a *list-of-files*	extract copy of listed files from libname.a

some Unix systems need to follow the archiver ar with ranlib:

ranlib libname.a – this builds a special table-of-contents file for quicker linking

Libraries

When one has a group of re-usable functions to link to, it is often preferable to build a *library*. Object files in a library are linked to as needed.

Libraries under Unix are generated by the ar command. (short for “archive”)

ar q libname.a *list-of-files*	“quick” add files
ar r libname.a *list-of-files*	add files to libname.a; replace any already in libname.a
ar d libname.a *list-of-files*	delete files from libname.a
ar x libname.a *list-of-files*	extract copy of listed files from libname.a

some Unix systems need to follow the archiver ar with ranlib:

ranlib libname.a – this builds a special table-of-contents file for quicker linking

These libraries are *static libraries*; the linker will build the executable by linking them physically into the executable.

Shared Libraries

Shared libraries: libraries which are dynamically linked in as the program executes.

Loadable shared libraries...

Shared Libraries

Shared libraries: libraries which are dynamically linked in as the program executes.

Loadable shared libraries...

- reduce the need for copies of their contents

Shared Libraries

Shared libraries: libraries which are dynamically linked in as the program executes. Loadable shared libraries...

- reduce the need for copies of their contents
- makes updated versions immediately available to programs without requiring them to be re-linked.

Shared Libraries

Shared libraries: libraries which are dynamically linked in as the program executes.

Loadable shared libraries...

- reduce the need for copies of their contents
- makes updated versions immediately available to programs without requiring them to be re-linked.

How to compile source `gcc -fPIC -rdynamic -O2 -c file.c`

creates a "Position Independent Code" *(at optimization level 2)*

Shared Libraries

Shared libraries: libraries which are dynamically linked in as the program executes. Loadable shared libraries...

- reduce the need for copies of their contents
- makes updated versions immediately available to programs without requiring them to be re-linked.

How to compile source `gcc -fPIC -rdynamic -O2 -c file.c`

creates a "Position Independent Code" *(at optimization level 2)*

Creating the Shared Library `gcc -fPIC -shared -Wl,-soname,libname.so.# -o libname.so.# *.o`

One may need to also include `-Wl,export-dynamic` ("#" is a version number)

Shared Libraries

Shared libraries: libraries which are dynamically linked in as the program executes. Loadable shared libraries...

- reduce the need for copies of their contents
- makes updated versions immediately available to programs without requiring them to be re-linked.

How to compile source `gcc -fPIC -rdynamic -O2 -c file.c`
creates a "Position Independent Code" *(at optimization level 2)*

Creating the Shared Library `gcc -fPIC -shared -Wl,-soname,libname.so.# -o libname.so.# *.o`
One may need to also include `-Wl,export-dynamic` ("#" is a version number)

ldd To display which shared libraries are used by a program, use
`ldd progname`