Elements of a C Program

- A C development environment includes
  - *System libraries and headers*: a set of standard libraries and their header files. For example see `/usr/include` and `glibc`.
  - *Application Source*: application source and header files
  - *Compiler*: converts source to object code for a specific platform
  - *Linker*: resolves external references and produces the executable module
- User program structure
  - there must be one main function where execution begins when the program is run. This function is called main
    - `int main (void) { ... },`
    - `int main (int argc, char *argv[]) { ... }`
    - UNIX Systems have a 3rd way to define main(), though it is not POSIX.1 compliant
      `int main (int argc, char *argv[], char *envp[])`
  - additional local and external functions and variables

A Simple C Program

- *Create* example file: `try.c`
- *Compile* using gcc:
  `gcc –o try try.c`
- The standard C library *libc* is included automatically
- *Execute* program
  `./try`
- Note, I always specify an absolute path
- Normal termination:
  `void` **`exit`**`(int status);`
  - calls functions registered with `atexit()`
  - flush output streams
  - close all open streams
  - return status value and control to host environment

```
/* you generally want to
 * include stdio.h and
 * stdlib.h
 * */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
   printf("Hello World\n");
   exit(0);
}
```

Source and Header files

- Header files (*.h) export interface definitions
  - function prototypes, data types, macros, inline functions and other common declarations
  - const definitions

- Do not place source code (i.e. definitions) in the header file with a few exceptions.
- *C preprocessor* (cpp) is used to insert common definitions into source files
- There are other cool things you can do with the preprocessor

C Standard Header Files you may want to use

- Standard Headers you should know about:
  - `stdio.h` - file and console (also a file) IO: *`perror`*, *`printf`*, *`open`*, *`close`*, *`read`*, *`write`*, *`scanf`*, etc.
  - `stdlib.h` - common utility functions: *`malloc`*, *`calloc`*, *`strtol`*, *`atoi`*, etc
  - `string.h` - string and byte manipulation: *`strlen`*, *`strcpy`*, *`strcat`*, *`memcpy`*, *`memset`*, etc.
  - `ctype.h` - character types: *`isalnum`*, *`isprint`*, *`isupport`*, *`tolower`*, etc.
  - `errno.h` - defines *`errno`* used for reporting system errors
  - `math.h` - math functions: *`ceil`*, *`exp`*, *`floor`*, *`sqrt`*, etc.
  - `signal.h` - signal handling facility: *`raise`*, *`signal`*, etc
  - `stdint.h` - standard integer: *`intN_t`*, *`uintN_t`*, etc
  - `time.h` - time related facility: *`asctime`*, *`clock`*, *`time_t`*, etc.

The Preprocessor

- The C preprocessor permits you to define simple macros that are evaluated and expanded prior to compilation.

- Commands begin with a '#'. Abbreviated list:
  - `#define` : defines a macro
  - `#undef` : removes a macro definition
  - `#include` : insert text from file
  - `#if` : conditional based on value of expression
  - `#ifdef` : conditional based on whether macro defined
  - `#ifndef` : conditional based on whether macro is not defined
  - `#else` : alternative
  - `#elif` : conditional alternative
  - `defined()` : preprocessor function: 1 if name defined, else 0

```
#if defined(__NetBSD__)
```

Difference between #if and #ifdef

#if checks for the value of the symbol
#ifdef checks the existence of the symbol

```
#define ABC 0
#if ABC
// will not compile
#endif
#ifdef ABC
// will compile
#endif
```

#ifdef ABC is a shortcut for #if defined(ABC)

#if can be used for more complex preprocessor conditions.

```
#if defined(ABC) || defined(XYZ)
```

Another Example C Program

/usr/include/stdio.h

```
/* comments */
#ifndef _STDIO_H
#define _STDIO_H

... definitions and protoypes

#endif
```

/usr/include/stdlib.h

```
/* prevents including file
 * contents multiple
 * times */
#ifndef _STDLIB_H
#define _STDLIB_H

... definitions and protoypes

#endif
```

`#include` directs the preprocessor to "include" the contents of the file at this point in the source file. `#define` directs preprocessor to define macros.

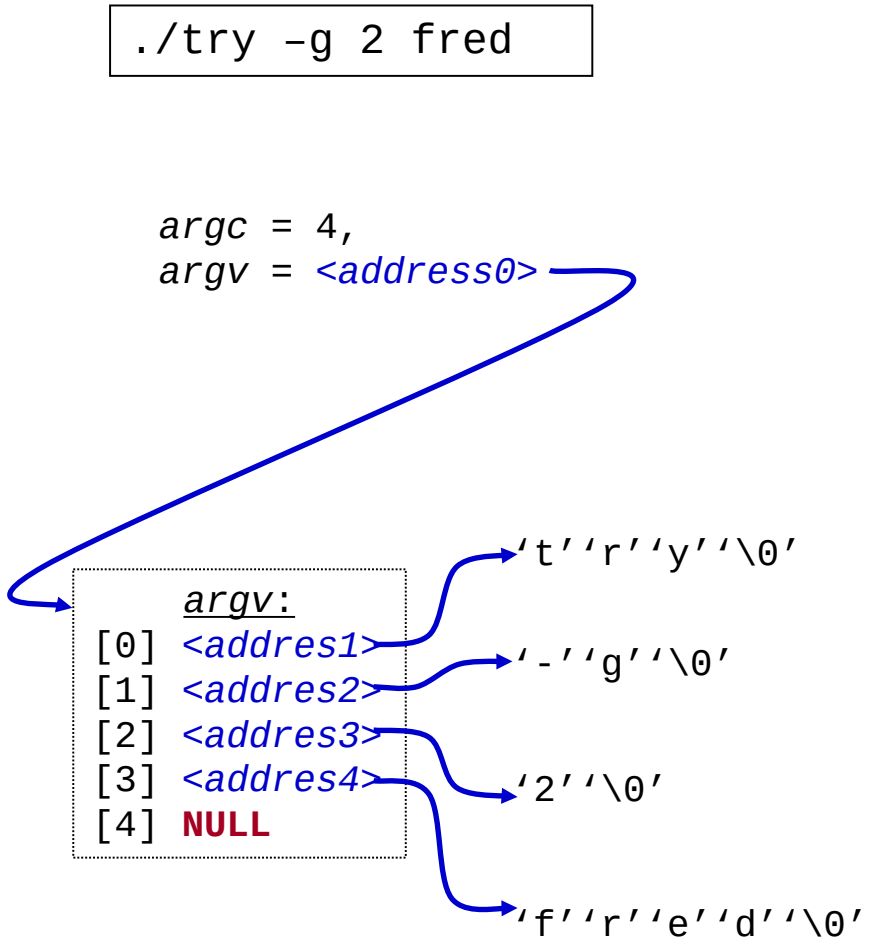
example.c

```
/* You generally want to place
 * all file includes at start of file
 * */
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char **argv)
 {
  // printf prototype in stdio.h
  printf("Hello, Prog name = %s\n",
            argv[0]);
  exit(0);
 }
```

Passing Command Line Arguments

- When you execute a program you can include arguments on the command line.
- The run time environment will create an argument vector.
  - `argv` is the argument vector
  - `argc` is the number of arguments
- Argument vector is an array of pointers to strings.
- a *string* is an array of characters terminated by a binary 0 (NULL or '\0').
- *argv[0]* is always the program name, so *argc* is at least 1.



Another Simple C Program

```
int main (int argc, char **argv) {
  int i;
  printf("There are %d arguments\n", argc);
  for (i = 0; i < argc; i++)
    printf("Arg %d = %s\n", i, argv[i]);

  return 0;
}
```

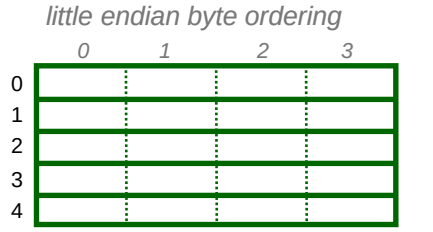
- What's new in the above simple program?
  - Pointers will give you the most trouble

Arrays and Pointers

- A variable declared as an array represents a contiguous region of memory in which the array elements are stored.

```
int x[5]; // an array of 5 4-byte ints.
```

- All arrays begin with an index of 0



memory layout for array x

- An array identifier is equivalent to a pointer that references the first element of the array
  - `int x[5], *ptr;`
    `ptr = &x[0]` is equivalent to `ptr = x;`
- Pointer arithmetic and arrays:
  - `int x[5];`
    `x[2]` is the same as `*(x + 2)`, the compiler will assume you mean 2 objects beyond element x.

Arrays and Pointers

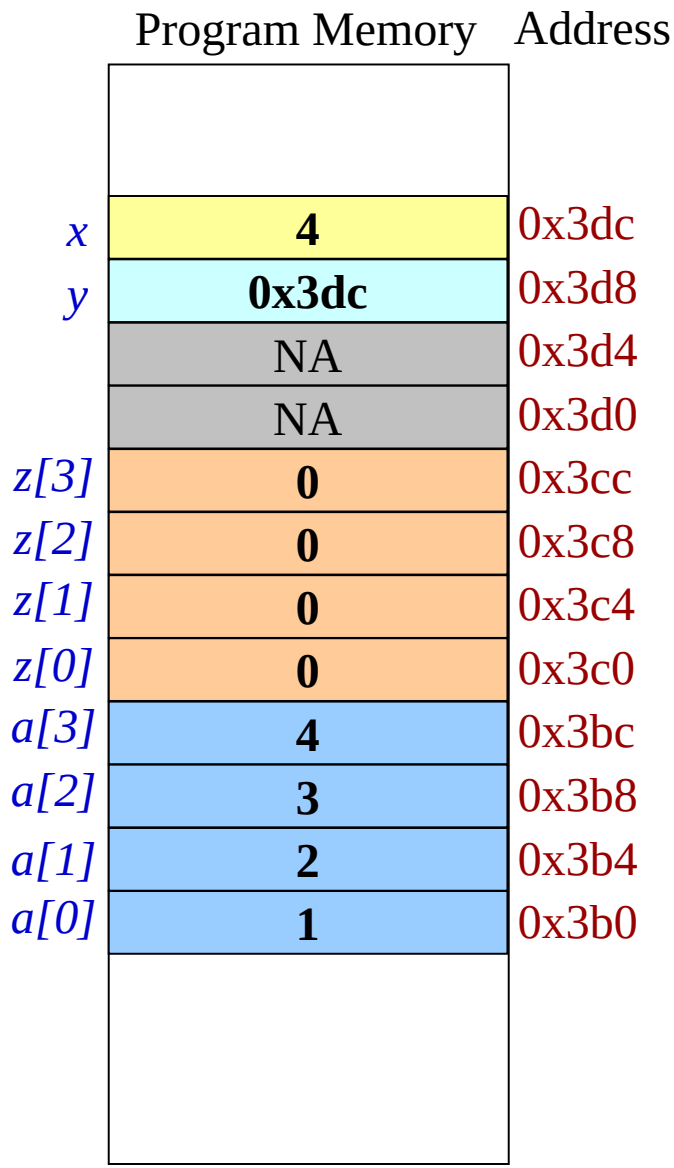
- int A[];
- int* p;
- p = &A[0];
- p++ always advances to the next element of A – it's not p+4 or p+sizeof(int). p++ increments the address by the size of the type it points to automatically, by definition of the operator.

Pointers

- For any type T, you may form a pointer type to T.
  - Pointers may reference a function or an object.
  - The value of a pointer is the address of the corresponding object or function
  - Examples: `int *i; char *x; int (*myfunc)();`
- Pointer operators: ***** dereferences a pointer, *&* creates a pointer (reference to)
  - ```
    int i = 3; int *j = &i;
    *j = 4; printf("i = %d\n", i); // prints i = 4
    ```
  - ```
    int myfunc (int arg);
    int (*fptr)(int) = myfunc;
    i = fptr(4); // same as calling myfunc(4);
    ```
- Generic pointers:
  - Traditional *C* used (char *)
  - Standard *C* uses (void *) – these can not be dereferenced or used in pointer arithmetic. So they help to reduce programming errors
- Null pointers: use ***NULL*** or ***0***. *It is a good idea to always initialize pointers to NULL.*

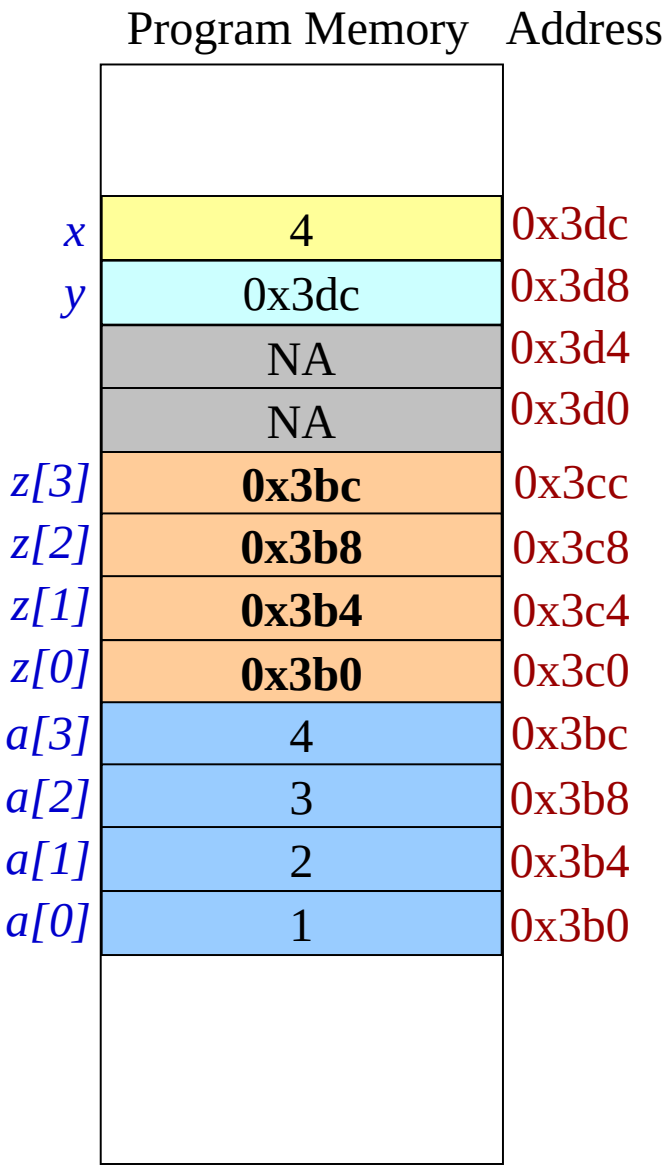
Pointers in C

```
Step 1:
int main (int argc, argv) {
int  x = 4;
int *y = &x;
int *z[4] = {NULL, NULL, NULL, NULL};
int  a[4] = {1, 2, 3, 4};
...
```



Pointers Continued

```
Step 1:
int main (int argc, argv) {
int  x = 4;
int *y = &x;
int *z[4] = {NULL, NULL, NULL, NULL};
int  a[4] = {1, 2, 3, 4};
Step 2: Assign addresses to array Z
z[0] = a; // same as &a[0];
z[1] = a + 1;    // same as &a[1];
z[2] = a + 2;// same as &a[2];
z[3] = a + 3;// same as &a[3];
```



Pointer Validity

A Valid pointer is one that points to memory that your program controls.

Using invalid pointers will cause non-deterministic behavior, and will often cause Linux to kill your process (SEGV or Segmentation Fault).

There are two general causes for these errors:

Program errors that set the pointer value to a strange number

Use of a pointer that was at one time valid, but later became invalid

Pointer Validity

Will ptr be valid or invalid?

```
char * get_pointer()
 {
  char x=0;
  return &x;
 }

main ()
 {
  char * ptr = get_pointer();
  *ptr = 12;  /* valid? */
 }
```


16

C Pointer Declarations

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int

Structures

- structures
  - `struct MyPoint {int x, int y};`
  - `typedef struct MyPoint MyPoint_t;`
  - `MyPoint_t point, *ptr;`
  - `point.x = 0; point.y = 10;`
  - `ptr = &point; ptr->x = 12; ptr->y = 40;`

Functions

- Always use function prototypes

```
int myfunc (char *, int, struct MyStruct *);
int myfunc_noargs (void);
void myfunc_noreturn (int i);
```

- C calls are *call by value*, copy of parameter passed to function
  - if you want to alter the parameter then pass a pointer to it

Lexical Scoping

Every Variable is Defined within some scope. A Variable cannot be referenced by name (a.k.a. Symbol) 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.

```
void p(char x)
{
              /* p,x */
  char y;
              /* p,x,y */
  char z;
              /* p,x,y,z */
}
              /* p */
char z;
              /* p,z */

void q(char a)
{
  char b;
              /* p,z,q,a,b */

  {                                  char b?
    char c;
              /* p,z,q,a,b,c */
  }
                                     legal?
  char d;
  /* p,z,q,a,b,d (not c) */
}

/* p,z,q */
```

Dynamic Memory Allocation

So far all of our examples have allocated variables statically by defining them in our program. This allocates them in the stack.

But, what if we want to allocate variables based on user input or other dynamic inputs, at run-time? This requires dynamic allocation.

sizeof() reports the size of a type in bytes

For details:
$ man calloc

```
int * alloc_ints(size_t requested_count)
{
  int * big_array;
  big_array = (int *)calloc(requested_count, sizeof(int));
  if (big_array == NULL) {
    printf("can't allocate %d ints: %m\n", requested_count);
    return NULL;
  }

  /* now big_array[0] .. big_array[requested_count-1] are
   * valid and zeroed. */
  return big_array;
}
```

calloc() allocates memory for N elements of size k

Returns NULL if can't alloc

%m ?

Emstar tips

It's OK to return this pointer. It will remain valid until it is freed with free()

21

Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:

- Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()'d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory is called a "memory leak".

- Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that has been freed. Whenever you free memory you must be certain that you will not try to use it again. It is safest to erase any pointers to it.

- Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic

22

Malloc

- `#include <stdlib.h>`
- `void *malloc(size_t size)`

Successful: Returns a pointer to a memory block of at least size bytes aligned to 8-byte boundary

Unsuccessful: returns NULL (0) and sets errno

- `void free(void *p)`

Returns the block pointed at by **p** to pool of available memory

p must come from a previous call to **`malloc()`** or **`realloc()`**

- `void *realloc(void *p, size_t size)`

Changes size of block **p** and returns pointer to new block

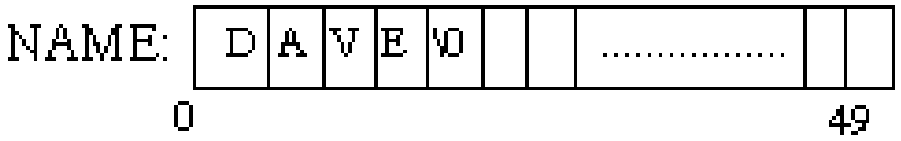
Contents of new block unchanged up to min of old and new size

Old block has been **`free()`**'d (logically, if new != old)

What are strings in C?

- Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the *null character*, which is simply the character with the value 0 (or \0).
- For example:

  char name[50] = "DAVE";



Conventions

- Are the following correct, given char a[6];

```
a[6]='7';
a[5]="7";
a[5]=7;
a++;
```

Declarations and Initializing strings

- Array of char:

```
char str[5] = {'l', 'i', 'n', 'u', 'x'};
char str[6] = {'l', 'i', 'n', 'u', 'x', '\0'};
```

- Declarations

Declarations and Initializing strings

```
#include <stdio.h>
#include <string.h>
int main()
 {
  char string1[ ] = "A string declared as an array.\n";
  char *string2 = "A string declared as a pointer.\n";
  char string3[30];
  strcpy(string3, "A string constant copied in.\n");
  printf (string1);
  printf (string2);
  printf (string3);
  return 0;
 }
```

Declarations and Initializing strings

- char string1[] = "A string declared as an array.\n";

  This is usually the best way to declare and initialize a string. The character array is declared explicitly. There is no size declaration for the array; just enough memory is allocated for the string, because the compiler knows how long the string constant is. The compiler stores the string constant in the character array and adds a null character (\0) to the end.
- Can you do the following?

```
char* foo = (char*) malloc(10 * sizeof(char));
foo = "Hello!";
free(foo);
```

Declarations and Initializing strings

- char *string2 = "A string declared as a pointer.\n";

  The second of these initializations is a pointer to an array of characters. Just as in the last example, the compiler calculates the size of the array from the string constant and adds a null character. The compiler then assigns a pointer to the first character of the character array to the variable string2.

Note: Most string functions will accept strings declared in either of these two ways.

Declarations and Initializing strings

- char string3[30];
- Declaring a string in this way is useful when you don't know what the string variable will contain, but have a general idea of the length of its contents (in this case, the string can be a maximum of 30 characters long). The drawback is that you will either have to use some kind of string function to assign the variable a value, as the next line of code does (strcpy(string3, "A string constant copied in.\n");), or you will have to assign the elements of the array the hard way, character by character.

Memory-Related Perils and Pitfalls

Dereferencing bad pointers
Reading uninitialized memory
Overwriting memory
Referencing nonexistent variables
Freeing blocks multiple times
Referencing freed blocks
Failing to free blocks

Perils and Pitfalls

Explain what is wrong in each of the following slides

Dereferencing Bad Pointers

The classic `scanf` bug

```
int val;

...

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

Reading Uninitialized Memory

Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x)
 {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
 }
```

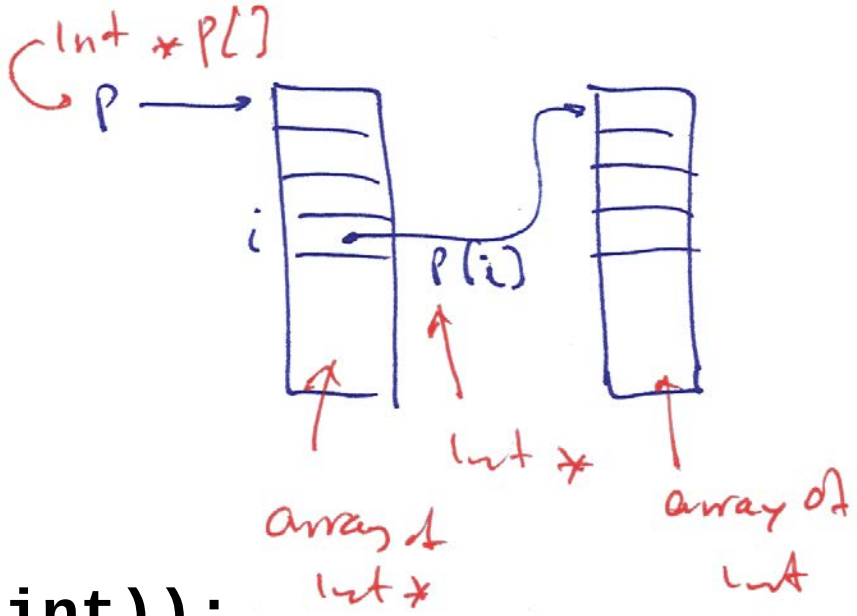
Overwriting Memory

Allocating the (possibly) wrong sized object

```
int **p;

p = malloc(N * sizeof(int));

for (i = 0; i < N; i++)
 {
  p[i] = malloc(M * sizeof(int));
 }
```




```
int **p;
int *p[];
p = malloc(N * sizeof(int));
p = (int **) malloc(N * sizeof(int *));

for (i = 0; i < N; i++)
 {
  p[i] = malloc(M * sizeof(int));
  p[i] = (int *) malloc(M * sizeof(int));
 }
```

Overwriting Memory

Off-by-one error

```
int **p;

p = malloc(N * sizeof(int *));

for (i = 0; i <= N; i++)
 {
  p[i] = malloc(M * sizeof(int));
 }
```

Overwriting Memory

Not checking the max string size

```
char s[8];
int i;

gets(s);  /* reads “123456789” from stdin */
```

Overwriting Memory

Misunderstanding pointer arithmetic

```
int *search(int *p, int val)
 {

   while (*p && *p != val)
    p += sizeof(int);

   return p;
 }
```

Referencing Nonexistent Variables

Forgetting that local variables disappear when a function returns

```
int *foo ()
 {
  int val;

  return &val;
 }
```

Freeing Blocks Multiple Times

```
x = malloc(N*sizeof(int));
/* manipulate x */
free(x);

y = malloc(M*sizeof(int));
/* manipulate y */
free(x);
```

Referencing Freed Blocks

```
x = malloc(N*sizeof(int));
/* manipulate x */
free(x);
   ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Failing to Free Blocks (Memory Leaks)

Slow, long-term killer!

```
foo()
 {
  int *x = malloc(N*sizeof(int));
  ...
  return;
 }
```

Failing to Free Blocks (Memory Leaks)

Freeing only part of a data structure

```
struct list {
   int val;
   struct list *next;
};

foo()
 {
  struct list *head = malloc(sizeof(struct list));
  head->val = 0;
  head->next = NULL;
  /* create and manipulate the rest of the list */
  free(head);
  return;
 }
```