Outline

Part III

The C Language

Thinking in C vs. Java

- There are a number of differences that require a new way of thinking.
- C is sometimes known as a “portable, high-level assembly language” and is often used to write operating systems, database management systems, graphics systems because C allows you to have:
  1. **power:** manage and manipulate memory down to a single bit, or access I/O devices
  2. **performance:** have a simple mapping from C construct to execution model
  3. **understandability:** does not have (many) hidden overheads and side effects
  4. **functionality:** many libraries and good support on Unix

Thinking in C vs. Java (2)

- But, with power comes more responsibility for the programmer:
  1. C does not have automatic initialization/finalization via constructors/destructors
  2. C does not have automatic array bounds checking
  3. C does not have garbage collection
  4. C will let you bypass type checking, both explicitly and implicitly
  5. C will let you modify **any** part of the user's address space
- The programmer must learn the model, the "best practices", and the standard toolset (e.g., debuggers)
- C can be used for any kind of programming. But, it is not the best choice for all kinds of applications.
- We all need to know C because of all the C code "out there", including Linux, compilers (e.g., gcc), etc.

Thinking in C vs. Java (3)

1. Instead of Java's objects, think of C's structures and abstract data types (ADT)
2. Instead of Java's class methods, think of C's functions with similar parameter types (e.g., `FILE *` for `fprintf(), fscanf()`)
3. Instead of Java's method overloading, think of C's functions with different but similar names (e.g., `scanf(), fscanf(), sscanf()`)
4. Instead of Java's string class, think of C's arrays of memory and buffers used to hold characters and other data
5. Instead of Java's garbage collection, think of C's `free(), fclose(), pclose()`, etc.

Thinking in C vs. Java (4)

6. Instead of Java's object references, think of C's pointers to anything
7. Instead of Java's exceptions, think of a C function's return value and the global variable `errno`
8. Instead of Java's `import` feature for packages, think of C's header files (i.e., `#include`) and linking to libraries
9. Instead of Java's virtual machine, think of C's execution model and address space organization
10. Instead of Java's libraries, think of C's libraries and Unix system calls

Thinking in C vs. Java (5)

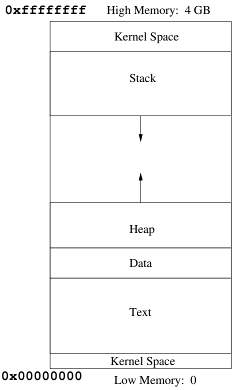
11. Instead of Java's boolean `FALSE` and `TRUE`, think of C's interpretation of zero and non-zero values. (NOTE: C99 does have a `bool` type.)
12. Instead of Java's `new` and garbage collection, think of C's library functions `malloc()`, `calloc()`, and `free()`
13. Instead of Java's <u>inheritance</u>, think of C's ???? Wrappers (e.g., `myprintf()`)??
14. Instead of Java's <u>polymorphism</u>, think of C's ???? Function pointers ??

Actually, one can write in an OO style (including any design pattern) in C. But, it is often non-obvious or much more complicated.

A Process's Address Space

- A process is a program being executed.
- The von Neumann model includes the notion that code and data are stored in a common memory. We call that an address space.
- Each process (usually) has a separate address space.
- The key parts of an address space include:
  1. Program "text" segment: The executable code written by the user and library code (e.g., `main()` and `printf()`)
  2. Global "data" segment: Global variables and constants, including string constants (e.g, `char * str = "Hello"`)
  3. "Heap" segment: For dynamic memory management, such as `malloc()`
  4. "Stack" segment: For local variables, function parameters, and tracking nested functions calls and recursion

A Process's Address Space (2)



- "Stack" segment: Local variables, function parameters, support for function calls
  - Grows from high to low memory
- "Heap" segment: Dynamic memory, `malloc()`
  - Grows from low to high memory
- Global "data" segment: Global variables and constants
- Program "text" segment: Code

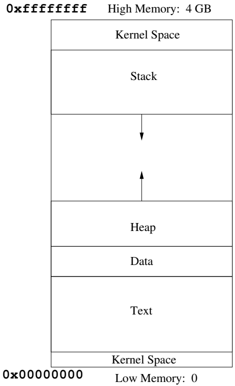
A Process's Address Space (3)

- We also distinguish between:
  1. User space: parts of memory that can be accessed by normal programs
     1. The user's code and data
     2. Library code and data
  2. Kernel or OS space: parts of memory that can only be accessed by the operating system
- A "segmentation fault" refers to an inappropriate access to a part of memory, usually due to a wrong pointer or overflowing a buffer.
- And, we distinguish between:
  1. Physical memory: RAM chips
  2. Logical memory: An illusion of memory provided by the OS (a big topic in CMPUT 379)

A Process's Address Space (4)

- When it comes to variables and pointers, we can ask 3 questions:
  1. Where is the storage, if any, for this variable or pointer? If it has storage, then it is an L-value. If it does not have storage, then it is an R-value or expression
  2. What is the value for this variable or pointer?
  3. What does this pointer point at? Where does this pointer point to?

A Process's Address Space (4)

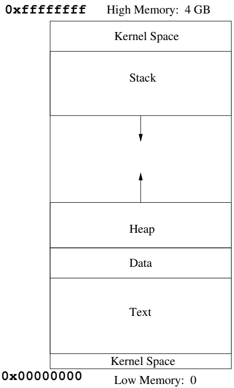


NOTE: Numbers may vary.

```
drafiei@ug20:~/201>cat t1.c
#include <stdio.h>

char Buffer[ 8 ];
int Global1;

int main( int argc, char * argv[] )
{
        int     local1;

        printf( "Buffer  = %p / %u\n",
                Buffer, (int)Buffer );
        printf( "Global1 = %p / %u\n",
                &Global1, (int)&Global1 );
        printf( "local1  = %p / %u\n",
                &local1, (int)&local1 );
        printf( "&(Buffer[ 7 ])  = %p / %u\n",
                &(Buffer[7]), (int)&(Buffer[7]) );
        return( 0 );
} /* main */
drafiei@ug20:~/201>gcc -Wall -std=c99 t1.c
drafiei@ug20:~/201>./a.out
Buffer  = 0x8049600 / 134518272
Global1 = 0x8049608 / 134518280
local1  = 0xbfd47640 / 3218372160
&(Buffer[ 7 ])  = 0x8049607 / 134518279
```

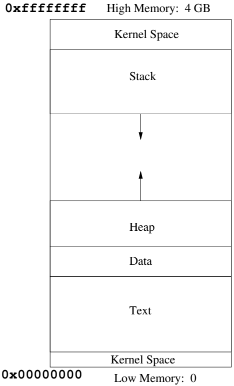
A Process's Address Space (5)

The C Language (1)
Thinking in C
Memory Model
Arrays and Strings in Memory
The C Language (2)
Linked Lists
Multidimensional Arrays
The C Language (3)
Storage and Scope
Function Pointers
Initialization




```
#include <stdio.h>
int Global1;
void sub()
{
        int local1;
        printf( "sub (local1)  = %p / %u\n",
                &local1, (int)&local1 );
} /* sub */

int main( int argc, char * argv[] )
{
        int     local1;

        printf( "main(local1)  = %p / %u\n",
                &local1, (int)&local1 );
        sub();
        printf( "Global1       = %p  / %u\n",
                &Global1, (int)&Global1 );
        printf( "main()        = %p  / %u\n",
                main, (int)main );
        printf( "sub()         = %p  / %u\n",
                sub, (int)sub );
        return( 0 );
} /* main */
drafiei@ug20:~/201>./a.out
main(local1)  = 0xbff72070 / 3220643952
sub (local1)  = 0xbff72054 / 3220643924
Global1       = 0x804964c  / 134518348
main()        = 0x8048374  / 134513524
sub()         = 0x8048354  / 134513492
```

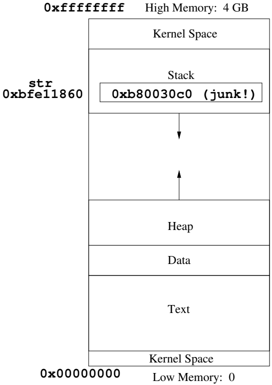
A Process's Address Space (6)




```
drafiei@ug20:~/201>./a.out
&str  = 0xbffe48dc / 3221113052
 str  = 0x8048489 / 134513801
 str  = 0xa00f008  / 167833608
 str2 = 0x8048543  / 134513987
```

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

void sub( char * str )
{
        printf( "&str  = %p / %u\n",
                &str, (int)&str );
        printf( " str  = %p  / %u\n",
                str, (int)str );
} /* sub */

int main( int argc, char * argv[] )
{
        char * str;
        char * str2 = "Hello";

        printf( "&str  = %p / %u\n",
                &str, (int)&str );
        printf( " str  = %p / %u\n",
                str, (int)str );
        str = malloc( 16 );
        printf( " str  = %p  / %u\n",
                str, (int)str );
        printf( " str2 = %p  / %u\n",
                str2, (int)str2 );
        sub( str );
        return( 0 );
} /* main */
```

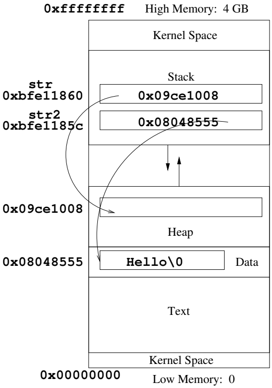
Arrays and Strings in Memory



Focusing on `str`

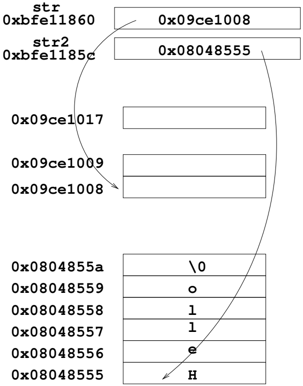
```
drafiei@ug20:~/201>cat t4.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>      /* New */

int main( int argc, char * argv[] )
{
        char * str;
        char * str2 = "Hello";

        printf( "&str   = %p / %u\n",
                &str, (int)&str );
        printf( " str   = %p / %u\n",
--->            str, (int)str );
        str = malloc( 16 );
        assert( str != NULL );    /* New */
        printf( " str   = %p  / %u\n",
                str, (int)str );
        printf( " &str2  = %p  / %u\n",
                &str2, (int)&str2 );
        printf( " str2  = %p  / %u\n",
                str2, (int)str2 );
        return( 0 );
} /* main */
drafiei@ug20:~/201>gcc -Wall -std=c99 t4.c
drafiei@ug20:~/201>./a.out
&str   = 0xbfe11860 / 3219200096
 str   = 0xb80030c0 / 3087020224
 str   = 0x9ce1008  / 164499464
 &str2  = 0xbfe1185c  / 3219200092
```

Arrays and Strings in Memory (2)



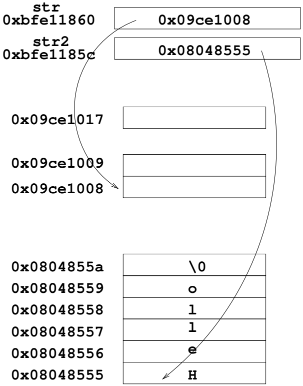

```
drafiei@ug20:~/201>cat t4.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>     /* New */

int main( int argc, char * argv[] )
{
        char * str;
        char * str2 = "Hello";

        printf( "&str   = %p / %u\n",
                &str, (int)&str );
        printf( " str   = %p / %u\n",
                str, (int)str );
        str = malloc( 16 );
        assert( str != NULL );    /* New */
        printf( " str   = %p  / %u\n",
                str, (int)str );
printf( " &str2  = %p  / %u\n",
                &str2, (int)&str2 );
        printf( " str2  = %p  / %u\n",
                str2, (int)str2 );
        return( 0 );
} /* main */
drafiei@ug20:~/201>gcc -Wall -std=c99 t4.c
drafiei@ug20:~/201>./a.out
&str   = 0xbfe11860 / 3219200096
 str   = 0xb80030c0 / 3087020224
 str   = 0x9ce1008  / 164499464
 &str2  = 0xbfe1185c  / 3219200092
```

Arrays and Strings in Memory (3)




```
drafiei@ug20:~/201>cat t4.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>     /* New */

int main( int argc, char * argv[] )
{
        char * str;
        char * str2 = "Hello";

        printf( "&str   = %p / %u\n",
                &str, (int)&str );
        printf( " str   = %p / %u\n",
                str, (int)str );
        str = malloc( 16 );
        assert( str != NULL );    /* New */
        printf( " str   = %p  / %u\n",
                str, (int)str );
printf( " &str2  = %p  / %u\n",
                &str2, (int)&str2 );
        printf( " str2  = %p  / %u\n",
                str2, (int)str2 );
        return( 0 );
} /* main */
drafiei@ug20:~/201>gcc -Wall -std=c99 t4.c
drafiei@ug20:~/201>./a.out
&str   = 0xbfe11860 / 3219200096
 str   = 0xb80030c0 / 3087020224
 str   = 0x9ce1008  / 164499464
 &str2  = 0xbfe1185c  / 3219200092
```

Arrays and Strings in Memory (4)




```
&( str[  0 ] ) is 0x09ce1008
&( str[  1 ] ) is 0x09ce1009
&( str[  2 ] ) is 0x09ce100a
&( str[  3 ] ) is 0x09ce100b
&( str[  4 ] ) is 0x09ce100c
&( str[  5 ] ) is 0x09ce100d
&( str[  6 ] ) is 0x09ce100e
&( str[  7 ] ) is 0x09ce100f
&( str[  8 ] ) is 0x09ce1010
...
&( str[ 15 ] ) is 0x09ce1017
&( str[ 16 ] ) is 0x09ce1018 (overflow!)

&( str2[ 0 ] ) is 0x08048555, str2[ 0 ] == 'H'
&( str2[ 1 ] ) is 0x08048556, str2[ 1 ] == 'e'
&( str2[ 2 ] ) is 0x08048557, str2[ 2 ] == 'l'
&( str2[ 3 ] ) is 0x08048558, str2[ 3 ] == 'l'
&( str2[ 4 ] ) is 0x08048559, str2[ 4 ] == 'o'
&( str2[ 5 ] ) is 0x0804855a, str2[ 5 ] == '\0'
```

Arrays and Strings in Memory (5)


```
drafiei@ug20:~/201>cat t5.c
#include <stdio.h>

int main( int argc, char * argv[] )
{
    int m[ 4 ];

    printf( "&( m[ 0 ] ) = %p\n", &( m[ 0 ] ) );
    printf( "&( m[ 1 ] ) = %p\n", &( m[ 1 ] ) );
    printf( "&( m[ 2 ] ) = %p\n", &( m[ 2 ] ) );
    printf( "&( m[ 3 ] ) = %p\n", &( m[ 3 ] ) );
    /* No valid index 4, but C doesn't care */
    printf( "&( m[ 4 ] ) = %p\n", &( m[ 4 ] ) );
    return( 0 );
} /* main */
drafiei@ug20:~/201>gcc -Wall -std=c99 t5.c
drafiei@ug20:~/201>./a.out
&( m[ 0 ] ) = 0xbff661a4
&( m[ 1 ] ) = 0xbff661a8
&( m[ 2 ] ) = 0xbff661ac
&( m[ 3 ] ) = 0xbff661b0
&( m[ 4 ] ) = 0xbff661b4

&( m[ 2 ] ) = 0xbff661a4 + ( 2 * sizeof(int) )
```

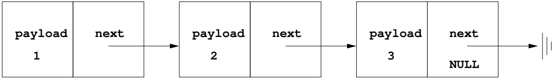
Arrays and Strings in Memory (6)

There is a “clear” mapping of indexing to assembly code.

```
drafiei@ug20:~/201>cat t6.c
#include <stdio.h>

int M[ 4 ];

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

        M[ 2 ] = 1;
        M[ i ] = 2;
        return( 0 );
} /* main */
```

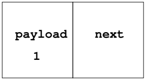
```
drafiei@ug20:~/201>gcc -S -Wall -std=c99 t6.c
drafiei@ug20:~/201>cat t6.s
        .file   "t6.c"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $16, %esp
        movl    $1, M+8
        movl    -8(%ebp), %eax
        movl    $2, M(,%eax,4)
        movl    $0, %eax
        addl    $16, %esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .comm   M,16,4
        .ident  "GCC: (GNU) 4.2.4"
        .section        .note.GNU-stack,"",@progbits
```

Linked Lists



- A linked list uses dynamic memory allocation and pointers to implement abstract, sequential arrays that can be of arbitrary size.
- Although C has pointers and Java has object references, the basic ideas behind linked lists are the same.
- The main issues are:
    1. How to declare the basic structure?
    2. How to allocate a node?
    3. How to use a node?
    4. How to insert a node into the list?
    5. How to find a node?
    6. How to delete a node?

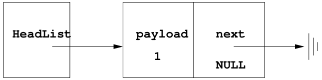
How to declare the basic structure?



- Typically, one uses a `struct`
- You can have a `struct Node *` inside the declaration of the structure itself
- The payload (or contents) can be anything, including other structures, arrays, other pointers (careful!), etc.

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

struct Node
{
  int payload;
  struct Node * next;
};

struct Node * HeadList = NULL;

int main( int argc, char * argv[] )
{
  struct Node * newNode;

  newNode = malloc( sizeof( struct Node ) );
  assert( newNode != NULL );

  newNode->payload = 1;

  /* Insert at head of list */
  newNode->next = HeadList;
  HeadList = newNode;

  return( 0 );
} /* main */
```

How to allocate/use a node?

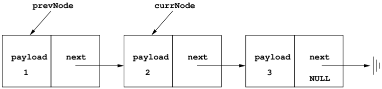


- When using `malloc()`, be careful to use the right expression.
  - `sizeof( struct Node ) == 8`
  - `sizeof( struct Node * ) == 4`
- The arrow operator (`->`)
  - `newNode->payload = 1` is the same as
  - `(*newNode).payload = 1`
- Use `malloc()` for dynamic allocation

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

struct Node
{
  int payload;
  struct Node * next;
};

struct Node * HeadList = NULL;

int main( int argc, char * argv[] )
{
  struct Node * newNode;

  newNode = malloc( sizeof( struct Node ) );
  assert( newNode != NULL );

  newNode->payload = 1;

  /* Insert at head of list */
  newNode->next = HeadList;
  HeadList = newNode;

  return( 0 );
} /* main */
```

The C Language (1)
Thinking in C
Memory Model
Arrays and Strings in Memory
The C Language (2)
Linked Lists
Multidimensional Arrays
The C Language (3)
Storage and Scope
Function Pointers
Initialization

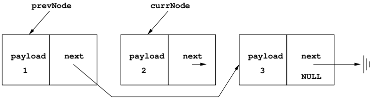
How to find a node?



- Never dereference a pointer that might be `NULL`.
  - `while(currNode != NULL)`
- Either fix your control flow, or use an assertion.
- It is OK to <u>assign</u> a `NULL`.

```
int main( int argc, char * argv[] )
{
  struct Node * currNode, * prevNode;

  /* Find */
  currNode = HeadList;
  prevNode = NULL;
  while( currNode != NULL )
  {
      if( currNode->payload == 2 )
            break;
      prevNode = currNode;
      currNode = currNode->next;
  } /* while */

  /* Delete */
  if( currNode != NULL )
  {
      if( currNode == HeadList )
      {
          HeadList = currNode->next;
      }
      else if( prevNode != NULL )
      {
          prevNode->next = currNode->next;
      }
      free( currNode );
  } /* if */

} /* main */
```

How to delete a node?



- Never dereference a pointer that might be `NULL`.
  - `if(currNode != NULL)`
  - `if(prevNode != NULL)`
- Deleting the first node of a list is a special case
- It is OK to <u>assign</u> a `NULL`.

```
int main( int argc, char * argv[] )
{
  struct Node * currNode, * prevNode;

  /* Find */
  currNode = HeadList;
  prevNode = NULL;
  while( currNode != NULL )
  {
      if( currNode->payload == 2 )
            break;
      prevNode = currNode;
      currNode = currNode->next;
  } /* while */

  /* Delete */
  if( currNode != NULL )
  {
      if( currNode == HeadList )
      {
          HeadList = currNode->next;
      }
      else if( prevNode != NULL )
      {
          prevNode->next = currNode->next;
      }
      free( currNode );
  } /* if */

} /* main */
```

Linked List: Miscellaneous

- It is a good idiom and best practice to immediately initialize a new node.

```
newNode = malloc( sizeof( struct Node ) );
assert( newNode != NULL );
memset( newNode, 0, sizeof( struct Node ) );
```

or

```
newNode = calloc( 1, sizeof( struct Node ) );
assert( newNode != NULL );
```

- See the C textbook(s) for more information and a different perspective.
- Was the check `if( prevNode != NULL )` actually necessary?

Linked List: Testing

- The best programmers are also good debuggers and good testers.
- Good testing is another example of paying “attention to detail”.
  - Imagine how silly it would be to build a car, check that the radio works, and then declare it ready for use.
  - Imagine how silly it would be to build a car, drive it down the alley–once, and then declare it ready for use.
- In Real World programming projects (e.g., Mozilla, Firefox), a Nightly Build and testing is an **expected** practice. See `http://www.mozilla.org/developer/#builds`
- Testing can find bugs. But, testing cannot prove that a program is bug free. But, you must still test, test, test.

Linked List: Testing (2)

- Some testing should be done by someone who did not write the code.
  - Video game companies employ lots of testers, who are not developers.
  - You are allowed to (encouraged to!) post your test cases in the course forum. Post your tests, not your code.
- The field of software engineering has a lot of ideas and theories about testing.
  1. White box vs. black box tests
  2. Unit vs. integration tests
  3. Coverage
  4. Pre-conditions and post-conditions
  5. Formal methods

Linked List: Testing (3)

Some key, practical rules of thumb are:

1. Testing should be done often.
   - Testing is part of the programming process. Testing is NOT something done just before a deadline. (Another example of "find problems early".)
   - Make it easy by creating (and updating) a `make test` target in your Makefile.
   - Consider a `make fulltest` too.
   - Use it just before submitting your assignment. Use it, on a fresh untar'ed copy of your code, after you submit.
   - The "extreme programming" school argues that a good, automatic testing infrastructure can actually speed up software development by giving the programmer confidence about code changes.

Linked List: Testing (4)

2. When you find a bug via testing, fix it, but also
   - Look for other places in the code for similar bugs.
   - Add relevant `assert()`s
   - Add a new test case.
   - Re-write the code if the bug was caused by bad style or bad design.
3. When you add a new feature or a new abstract data type (ADT), make sure you add new tests as well.
4. Design tests for the "corner cases". Be paranoid. Understand why things can go wrong.

Linked List: Testing (5)

Let's use the linked list as a case study:

1. Consider a function `testLinkedList()`. More details shortly.
2. Corner cases to unit test:
   1. Adding the first node to an empty list.
   2. Delete the first node, immediately.
   3. Try to delete a node, but linked list is empty.
   4. Add several nodes.
   5. Search for a node that exists.
   6. Search for a node that does not exist.
   7. Search for a node that exists and is at the head.
   8. Search for a node that exists and is at the tail.
   9. Delete a node. Repeat the search tests.
   10. Delete all nodes, start with the head node.
   11. Delete all nodes, start with the tail node.
   12. Delete all nodes, start with the middle node.

Linked List: Separate compilation

- You can use separate compilation to help with testing.
- You can also use separate compilation to create ADTs.
- Separate compilation allows you modularize your code into different `.c` files.
  1. The separate `.c` files can be compiled independently (and linked together).
  2. A `.c` file can implement one ADT.
  3. Debugging and testing code can be factored into a separate `.c` file.

Linked List: Separate compilation (2)

Suppose you wanted to create a Linked List ADT.

1. First, use a header file to declare the data structures, functionality, interface, or behaviour.

```
% cat linkedlist.h
#ifndef _LINKEDLIST_H_
#define _LINKEDLIST_H_

#define MAX_SOMETHING   128

struct Node
{
  int payload;
  struct Node * next;
};

extern struct Node * HeadList; /* Declaration */

extern void initializeLinkedList( struct Node * head );
extern void addToList( struct Node * head, struct Node * node );
extern void deleteFromListByKey( struct Node * head, int key );
extern struct Node * searchListByKey( struct Node * head, int key );

#endif /* _LINKEDLIST_H_ */
```

Do NOT put code or variable definitions in the header file.

Linked List: Separate compilation (3)

2. Second, use a source file (aka `.c` file) to define variables and functions.

```
% cat linkedlist.c
#include "memwatch.h"
#include "linkedlist.h"

struct Node * HeadList = NULL; /* Definition */

void initializeLinkedList( struct Node * head )
{
    ...code here...
} /* initializeLinkedList */

void addToList( struct Node * head, struct Node * node )
{
    ...code here...
} /* addToList */

void deleteFromListByKey( struct Node * head, int key );
{
    ...code here...
} /* deleteFromListByKey */

struct Node * searchListByKey( struct Node * head, int key );
{
    ...code here...
} /* searchListByKey */
```

Linked List: Separate compilation (4)

3. Third, compile the source file into an object file (aka `.o` file).

```
% gcc -Wall -std=c99 -DMEMWATCH -DMW_STDIO -c linkedlist.c
% ls *.o
linkedlist.o
```

4. Fourth, link the object files together to create full-fledged executables.

```
% cat testlinkedlist.c
#include "memwatch.h"
#include "linkedlist.h"

void testLinkedList()
{
    ...code here...
} /* testLinkedList */

int main( int argc, char * argv[])
{
        testLinkedList();
} /* main */
% gcc -Wall -std=c99 -DMEMWATCH -DMW_STDIO -c testlinkedlist.c
% gcc -Wall -std=c99 testlinkedlist.o linkedlist.o -o mytest
```

or

```
% gcc -Wall -std=c99 main.o linkedlist.o -o myexecutable
```

Linked List: Separate compilation (4)

5. Fifth, automate everything with your Makefile.

```
% cat Makefile
myexecutable: main.o linkedlist.o
        gcc -Wall -std=c99 main.o linkedlist.o -o myexecutable

mytest: testlinkedlist.o linkedlist.o
        gcc -Wall -std=c99 testlinkedlist.o linkedlist.o -o mytest

linkedlist.o:  linkedlist.c linkedlist.h
        gcc -Wall -std=c99 -DMEMWATCH -DMW_STDIO -c linkedlist.c

main.o:  main.c linkedlist.h
        gcc -Wall -std=c99 -DMEMWATCH -DMW_STDIO -c main.c

testlinkedlist.o:  testlinkedlist.c linkedlist.h
        gcc -Wall -std=c99 -DMEMWATCH -DMW_STDIO -c testlinkedlist.c

test:
        make myexecutable
        ./myexecutable test.in.1
        ./myexecutable test.in.2

testunit:
        make mytest
        ./mytest
```

Of course, you can use variables and constants (e.g., `CFLAGS`) in your Makefile

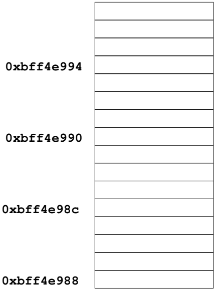
Multidimensional Arrays

- Scientific, math-oriented programs can require multidimensional arrays.
- Non-math-oriented (aka integer) programs tend to use structures, arrays of structures, and pointer-based data structures.
- For Assignment #1, you didn't need multidimensional arrays. An array of structs would have looked cleaner.
- For Assignment #2, consider using an ADT (or array of struct) instead of multidimensional arrays.
- **Case 1:** If the dimensions of the array are known at compile-time (i.e., are static), then multidimensional arrays are easy. (See K or KR textbooks)

Multidimensional Arrays (2)


```
#include <stdio.h>
#define SIZE 2
int main( int argc, char ** argv )
{
  int m[ SIZE ][ SIZE ], * mAlt, i, j;

  /* Access like multidimensional array */
  for( i = 0; i < SIZE; i++ )
  {
    printf( "%2d:  ", i );
    for( j = 0; j < SIZE; j++ )
    {
      printf( "%p  ", &( m[ i ][ j ] ) );
    }
    printf( "\n" );
  }
  printf( "\n" );
  /* Access by indexing manually */
  mAlt = &( m[ 0 ][ 0 ] );
  for( i = 0; i < SIZE; i++ )
  {
    printf( "%2d:  ", i );
    for( j = 0; j < SIZE; j++ )
    {
      printf( "%p  ", &( mAlt[ i * SIZE + j ] ) );
    }
    printf( "\n" );
  }
  return( 0 );
} /* main */
```

```
drafiei@ug20:~/201>make
gcc -Wall -std=c99 marray.c -o marray
drafiei@ug20:~/201>./marray
 0:  0xbff4e988  0xbff4e98c
 1:  0xbff4e990  0xbff4e994

 0:  0xbff4e988  0xbff4e98c
 1:  0xbff4e990  0xbff4e994
```

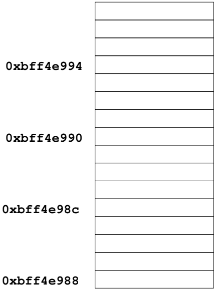


Multidimensional Arrays (3)

```c
#include <stdio.h>
#define SIZE 2
int main( int argc, char ** argv )
{
  int m[ SIZE ][ SIZE ], * mAlt, i, j;

  /* Access like multidimensional array */
  for( i = 0; i < SIZE; i++ )
  {
    printf( "%2d:  ", i );
    for( j = 0; j < SIZE; j++ )
    {
      printf( "%p  ", &( m[ i ][ j ] ) );
    }
    printf( "\n" );
  }
  printf( "\n" );
...snip...
```

```
drafiei@ug20:~/201>make
gcc -Wall -std=c99 marray.c -o marray
drafiei@ug20:~/201>./marray
 0:  0xbff4e988  0xbff4e98c
 1:  0xbff4e990  0xbff4e994

 0:  0xbff4e988  0xbff4e98c
 1:  0xbff4e990  0xbff4e994
```

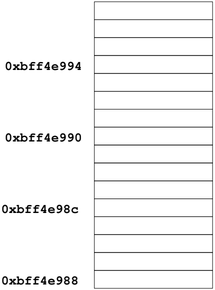
- C lays out an array in row-major format: one complete row, then another row.
- FORTRAN is the best known language that uses column-major format



Multidimensional Arrays (4)


```c
#include <stdio.h>
#define SIZE 2
int main( int argc, char ** argv )
{
  int m[ SIZE ][ SIZE ], * mAlt, i, j;

...snip...

  /* Access by indexing manually */
  mAlt = &( m[ 0 ][ 0 ] );
  for( i = 0; i < SIZE; i++ )
  {
    printf( "%2d:  ", i );
    for( j = 0; j < SIZE; j++ )
    {
      printf( "%p  ", &( mAlt[ i * SIZE + j ] ) );
    }
    printf( "\n" );
  }
  return( 0 );
} /* main */
```

```
drafiei@ug20:~/201>make
gcc -Wall -std=c99 marray.c -o marray
drafiei@ug20:~/201>./marray
 0:  0xbff4e988  0xbff4e98c
 1:  0xbff4e990  0xbff4e994

 0:  0xbff4e988  0xbff4e98c
 1:  0xbff4e990  0xbff4e994
```



- If you understand the layout of arrays, you can do the indexing manually.

Multidimensional Arrays (5)


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

int main( int argc, char ** argv )
{
  int  * mAlt, i, j, size;

  size = atoi( argv[ 1 ] );
  assert( size >= 0 && size < 1024 );

  mAlt = malloc( size*size*sizeof(int) );
  assert( mAlt != NULL );

  /* Access by indexing manually */
  for( i = 0; i < size; i++ )
  {
    printf( "%2d:  ", i );
    for( j = 0; j < size; j++ )
    {
      printf( "%p  ",
             &( mAlt[ i * size + j ] ) );
    }
    printf( "\n" );
  }
  return( 0 );
} /* main */
```

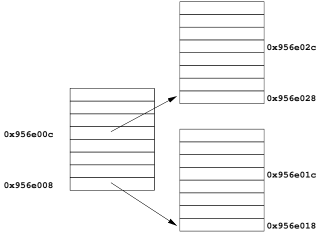
```
drafiei@ug20:~/201>make
gcc -Wall -std=c99 marray.2.c -o marray.2
drafiei@ug20:~/201>./marray.2 2
 0:  0x9181008  0x918100c
 1:  0x9181010  0x9181014

drafiei@ug20:~/201>./marray.2 3
 0:  0x8e76008  0x8e7600c  0x8e76010
 1:  0x8e76014  0x8e76018  0x8e7601c
 2:  0x8e76020  0x8e76024  0x8e76028

drafiei@ug20:~/201>./marray.2 -1
marray.2: marray.2.c:10: main:
Assertion `size >= 0 && size < 1024'
failed.
Abort (core dumped)
```

- **Case 2:** If the dimensions are not known until run-time, you can use `malloc()` and manual indexing

Multidimensional Arrays (6)


```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
int main( int argc, char ** argv )
{
  int  ** mAltPtr, i, j, size;

  size = atoi( argv[ 1 ] );
  assert( size >= 0 && size < 1024 );
  mAltPtr = malloc( size * sizeof( int * ) );
  assert( mAltPtr != NULL );

  for( i = 0; i < size; i++ )
  {
    mAltPtr[ i ] = malloc( size * sizeof(int) );
    assert( mAltPtr[ i ] != NULL );
  }

  /* Access by indexing manually */
  for( i = 0; i < size; i++ )
  {
    printf( "%2d:  ", i );
    printf( "%p -> ", &( mAltPtr[ i ] ) );
    for( j = 0; j < size; j++ )
    {
      printf( "%p  ", &( mAltPtr[ i ][ j ] ) );
    }
    printf( "\n" );
  }
  return( 0 );
} /* main */
```

```
drafiei@ug20:~/201>make
gcc -Wall -std=c99 marray.3.c -o marray.3
drafiei@ug20:~/201>./marray.3 2
 0:  0x956e008 -> 0x956e018  0x956e01c
 1:  0x956e00c -> 0x956e028  0x956e02c
```



- **Case 3:** You can use arrays-of-arrays and use the same notation as for static multidimensional arrays.
- There are storage overheads.

Multidimensional Arrays (7)


```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
int main( int argc, char ** argv )
{
  char  ** mAltPtr;
  int i, j, size;

  size = atoi( argv[ 1 ] );
  assert( size >= 0 && size < 1024 );
  mAltPtr = malloc( size * sizeof( char * ) );
  assert( mAltPtr != NULL );

  for( i = 0; i < size; i++ )
  {
    mAltPtr[ i ] = malloc( size * sizeof(char) );
    assert( mAltPtr[ i ] != NULL );
  }
  /* Access by indexing manually */
  for( i = 0; i < size; i++ )
  {
    printf( "%2d:  ", i );
    printf( "%p -> ", &( mAltPtr[ i ] ) );
    for( j = 0; j < size; j++ )
    {
      printf( "%p  ", &( mAltPtr[ i ][ j ] ) );
    }
    printf( "\n" );
  }
  return( 0 );
```

```
drafiei@ug20:~/201>make
gcc -Wall -std=c99 marray.4.c -o marray.4
drafiei@ug20:~/201>./marray.4 2
 0:  0x9463008 -> 0x9463018  0x9463019
 1:  0x946300c -> 0x9463028  0x9463029
```

- Of course, these techniques work for types other than integers
- Compare this to `char ** argv`.
- I recommend the Case 2 technique.

Matrix Multiplication


```
int a[SIZE][SIZE];
int b[SIZE][SIZE];
int c[SIZE][SIZE];
int i, j, k;
float csum;

/* c = a x b */
for( i = 0; i < size; i++ )
{
  for( j = 0; j < size; j++ )
  {
    csum = 0;
    for( k = 0; k < size; k++ )
    {
      csum += a[i][k] * b[k][j];
    }
    c[i][j] = csum;
  }
}
```

```
/* Assume a, b, c are matrix ADTs */
/* a is m x n.  b is n x p.  c is m x p */

int i, j, k;
float csum;

/* c = a x b */
for( i = 0; i < m; i++ )
{
  for( j = 0; j < p; j++ )
  {
    csum = 0;
    for( k = 0; k < n; k++ )
    {
      csum += getElem(a,i,k) * getElem(b,k,j);
    }
    setElem(c,i,j,csum);
  }
}
```

- For Assignment #2, you are likely using linked-list ADTs.
- You can abstract the details of how to find a particular row and column of a matrix via `getElem()` and `setElem()`

Matrix Addition

```
int a[SIZE][SIZE];
int b[SIZE][SIZE];
int c[SIZE][SIZE];
int i, j;
float csum;

/* c = a + b */
for( i = 0; i < size; i++ )
{
  for( j = 0; j < size; j++ )
  {
    csum = a[i][j] + b[i][j];
    c[i][j] = csum;
  }
}
```

```
/* Assume a, b, c are matrix ADTs */
/* a is m x n.  b is m x n.  c is m x n */

int i, j;
float csum;

/* c = a + b */
for( i = 0; i < m; i++ )
{
  for( j = 0; j < n; j++ )
  {
    csum = getElem(a,i,j) + getElem(b,i,j);
    setElem(c,i,j,csum);
  }
}
```

- Matrix addition only has 2 loops.
- The matrices are all the same size (i.e., m x n)
- Consider abstracting `m`,`n`, `p` with `getNumRows()` and `getNumCols()`

Storage and Scope


```
drafiei@ug20:~/201>cat t8.c
#include <stdio.h>
#include <stdlib.h>

void sub()
{
    static int i = 2;
    int j = 1;
    printf( "&i     = %p  / %u\n",
            &i, (int)&i );
    printf( " i     = %d\n", i );
    printf( "&j     = %p / %u\n",
            &j, (int)&j );
    printf( " j     = %d\n", j );
    i++;
    j++;
} /* sub */

int main( int argc, char * argv[] )
{
    sub();
    sub();
    sub();
    return( 0 );
} /* main */
drafiei@ug20:~/201>make
gcc -Wall -std=c99 t8.c -o t8
```

```
drafiei@ug20:~/201>./t8
&i      = 0x8049608  / 134518280
 i      = 2
&j      = 0xbfffba34 / 3221207604
 j      = 1
&i      = 0x8049608  / 134518280
 i      = 3
&j      = 0xbfffba34 / 3221207604
 j      = 1
&i      = 0x8049608  / 134518280
 i      = 4
&j      = 0xbfffba34 / 3221207604
 j      = 1
```

- The `static` declaration specifier (aka modifier) changes a local variable on the stack into a “global” variable in the data segment.
- But, only function `sub()` can “see” the variable `i`

Storage and Scope (2)


```
drafiei@ug20:~/201>cat t8.c
#include <stdio.h>
#include <stdlib.h>

void sub()
{
    static int i = 2;
    int j = 1;
    printf( "&i     = %p  / %u\n",
            &i, (int)&i );
    printf( " i     = %d\n", i );
    printf( "&j     = %p / %u\n",
            &j, (int)&j );
    printf( " j     = %d\n", j );
    i++;
    j++;
} /* sub */

int main( int argc, char * argv[] )
{
    sub();
    sub();
    sub();
    return( 0 );
} /* main */
```

- And, since `i` is now on the data segment, its values (and changes to its value) persist across multiple function calls.
- Notice how the value of `j` does not persist.
- Is it possible for other functions to change the value of `i`?
- Is it theoretically possible for local variables to keep their values from call-to-call, but are still on the stack?

Storage and Scope (3)

```
drafiei@ug20:~/201>cat t8.c
#include <stdio.h>
#include <stdlib.h>

void sub()
{
        static int i = 2;
#ifdef BONZO
        int j = 1;
#endif /* BONZO */
        int j;
        printf( "&i      = %p  / %u\n",
                &i, (int)&i );
        printf( " i      = %d\n", i );
        printf( "&j      = %p / %u\n",
                &j, (int)&j );
        printf( " j      = %d\n", j );
        i++;
        j++;
} /* sub */

int main( int argc, char * argv[] )
{
        sub();
        sub();
        sub();
        return( 0 );
} /* main */
drafiei@ug20:~/201>make
gcc -Wall -std=c99 t8.c -o t8
```

```
drafiei@ug20:~/201>./t8
&i      = 0x8049608  / 134518280
 i      = 2
&j      = 0xbfb52d94 / 3216321940
 j      = 134518244
&i      = 0x8049608  / 134518280
 i      = 3
&j      = 0xbfb52d94 / 3216321940
 j      = 134518245
&i      = 0x8049608  / 134518280
 i      = 4
&j      = 0xbfb52d94 / 3216321940
 j      = 134518246
```

- What happens if we remove the `= 1` part of `int j = 1;`?

Storage and Scope (4)


```
#include <stdio.h>

static struct Node * GlobalHeadOfList = NULL;

int main( int artc, char * argv[] )
{
    ...snip...
}
```

- The `static` declaration specifier (aka modifier) can also be used with global variables
- Such variables are exactly like other global variables, but they cannot be seen from other source code files, as per separate compilation.

Storage and Scope (5)

- Therefore, `static` helps implement a form of information hiding
  1. variables that are only visible within a function:
     King (pg. 461 2E) describes this as static storage duration, block scope, no linkage
  2. variables that are only visible within a separately compiled file:
     King (pg. 461 2E) describes this as static storage duration, file scope, internal linkage
- Within an ADT, which is typically within a `.c` file, consider making as many (previously) global variables `static`, as possible.

Storage and Scope (6)


```
#include <stdio.h>

static struct Node * GlobalHeadOfList = NULL;

static int FreeLinkedList( struct Node * head );

int main( int artc, char * argv[] )
{
    ...snip...
}
```

- The `static` declaration specifier (aka modifier) can also be used with functions.
- These functions cannot be seen (or called) from other source code files, as per separate compilation.
- Compare these concepts to Java's (and C++'s) much-more sophisticated use of `public`, `private`, `protected`, etc.

Function Pointers


```
% man qsort
QSORT(3)             Linux Programmer's Manual             QSORT(3)

NAME
       qsort - sorts an array

SYNOPSIS
       #include <stdlib.h>

       void qsort(void *base, size_t nmemb, size_t size,
                  int(*compar)(const void *, const void *));

DESCRIPTION
       The qsort() function sorts an array with nmemb elements of
       size size.  The base argument points to the start  of  the
       array.
...
       The comparison function must return an integer less  than,
       equal  to,  or  greater than zero if the first argument is
       considered to be respectively  less  than,  equal  to,  or
       greater than the second.  If two members compare as equal,
       their order in the sorted array is undefined.
```

- `qsort()` is a library function that implements the quicksort algorithm.

Function Pointers (2)


```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

- `int(*compar)(const void *, const void *)` is a function pointer: a pointer to a function.
- But, `compar` is not just any function.
- It must return type `int` and take a `const void *` and `const void *` as its two parameters
- It should do a comparison between two list elements, or else the algorithm will not work.

Function Pointers (3)


```c
#include <stdlib.h>
#define SIZE  128

/*
       void qsort(void *base, size_t nmemb, size_t size,
                    int(*compar)(const void *, const void *));
*/

int compare_ints( const void * p, const void * q )
{
    return( *((int *)p) - *((int *)q) );
}

int main( int argc, char * argv[] )
{
    int a[ SIZE ];

    qsort( a, SIZE, sizeof( a[ 0 ] ), compare_ints );

    return( 0 );
} /* main */
```

- Why is parameter `size` required?

Function Pointers (4)


```
#define SIZE 128

struct ClassSortedList {
    int list[ SIZE ];
    int(*compar)(const void *, const void *);
    int(*sortList)();
    int(*printList)();
};

extern int compareInts(const void *, const void *);
extern int sortIntList();
extern int printIntList();

int main( int argc, char * argv[] )
{
    struct ClassSortedList * obj;

    obj = (struct ClassSortedList *)calloc(1,sizeof( struct ClassSortedList));
    assert( obj != NULL );
    obj->compar = compareInts;
    obj->sortList = sortIntList;
    obj->printList = printIntList;
    obj->printList();    /* Call via function pointer */
    (*obj->printList)(); /* Call via function pointer */
} /* main */
```

- The concept of function pointers can be used to build an object.

Function Pointers (5)

From King, pg. 443, 2E

```
void (*file_cmd[])(void) = { new_cmd,   /* All _cmd identifiers */
                              open_cmd,  /*  refer to actual functions */
                              close_cmd,
                              close_all_cmd,
                              save_cmd,
                              save_as_cmd,
                              save_all_cmd,
                              print_cmd,
                              exit_cmd
                            };
```

- If the user enters an integer `n` between 0 and 8 (or selects a menu item between 0 and 8), then we can do this:

```
assert( 0 <= n && n <= 8 );
(*file_cmd[ n ])();   /* or file_cmd[n]() */
```

- How could this have been handled with a `switch` statement?

Initialization

We have already seen statements like the following:

```
int GlobalInt = 1;
char * GlobalString = "hello";

int main( int argc, char * argv[] )
{
    int localInt = 2;
    char * localString = "hello there";
} /* main */
```

- However, we can also provide initial values to structures, arrays, and arrays of structures.

Initialization (2)


```
#include <stdio.h>

struct pair
{
    int num;
    int square;
};

struct pair S10 = { 10, 100 };

/* Look-up table */
struct pair Squares[] = { {0, 0}, {1,1},
            {2,4}, {3,9}, {-1,-1} };

int main( int argc, char * argv[] )
{
    int i;
    for( i = 0; Squares[ i ].num != -1; i++ )
    {
        printf( "%2d ^ 2 = %2d\n",
                i, Squares[ i ].square );
    }

    return 0;
} /* main */
drafiei@ug20:~/201>make
gcc -Wall -std=c99 t10.c -o t10
```

```
drafiei@ug20:~/201>./t10
 0 ^ 2 =  0
 1 ^ 2 =  1
 2 ^ 2 =  4
 3 ^ 2 =  9
```

Initialization (3)


```
drafiei@ug20:~/201>cat t11.c
#include <stdio.h>

/* Look-up table */
int Squares[] = { 0, 1, 4, 9, -1 };

int main( int argc, char * argv[] )
{
    int i;
    for( i = 0; Squares[ i ] != -1; i++ )
    {
        printf( "%2d ^ 2 = %2d\n", i, Squares[ i ] );
    }

    return 0;
} /* main */
drafiei@ug20:~/201>make
gcc -Wall -std=c99 t11.c -o t11
drafiei@ug20:~/201>./t11
 0 ^ 2 =  0
 1 ^ 2 =  1
 2 ^ 2 =  4
 3 ^ 2 =  9
```

Initialization (4)

```
drafiei@ug20:~/201>cat t12.c
#include <stdio.h>

struct pair
{
    int num;
    char * s;
};

/* Look-up table */
struct pair Num[] = { {0, "zero"}, {1, "one"},
     {2, "two"}, {3, "three"}, {-1, "none"} };

int main( int argc, char * argv[] )
{
    int i;
    for( i = 0; Num[ i ].num != -1; i++ )
    {
        printf( "%2d = %s\n", i, Num[ i ].s );
    }

    return 0;
} /* main */
drafiei@ug20:~/201>make
gcc -Wall -std=c99 t12.c -o t12
drafiei@ug20:~/201>./t12
 0 = zero
 1 = one
 2 = two
 3 = three
```