Dynamic memory allocation and other programming tools

Introduction

- Fixed-size data structures
- Have to declare the size of arrays, and may end up going for too much
- Contradicts the savings of bytes we have been talking about
- Solution: Get only as much memory as needed – no more and no less – at run time

Self-referential structures

- Points to another structure that is of the same type as the original structure
- Exemplified by many data structures in computer science, including stacks, queues, linked lists, and trees
- Can be implemented by an array of structures with two elements

```
struct queue_node {
    info_type      info;
    int            ptr;
} queue[100];
```

  - In this queue, the second field contains an index within the array to "point to" the next element in order
  - May lead to problems if the index is not carefully controlled
  - Also, we still had to allocate space for 100 elements whether we need that much space or not

Dynamic memory allocation

- Enables a program to obtain more memory space at execution time and to release memory when it is no longer needed
- Limit for dynamic memory allocation can be as large as the amount of virtual memory available on the system
- The function `malloc` allocates storage for a variable and returns a pointer to it
- ANSI version is prototyped as

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

  - `void *` is used to return a generic pointer

- Assume the declarations

```
char *p;
int n = ...;
```

- The assignment statement

```
p = malloc(n);
```

  requests a block of memory space consisting of `n` bytes

  - If `n` consecutive bytes are available, the request is granted
  - The address of the first byte is assigned to `p`
  - If the call is unsuccessful, `p` is assigned `NULL`
  - It is a good idea to follow the `malloc()` by

```
if ( p == NULL ) ...
```

  - The allocated memory block can be treated the same as if it had been declared by using the statement

    ```
    char p[n]
    ```

    if that were possible (`n` is not a constant)

- Any integer expression can be used as an argument to `malloc` instead of only a constant-expression in array declaration
- `malloc` gives us more freedom with respect to when and where we reserve the memory
- Recommended use of `malloc`

```
person_info_type * person;
person = ( person_info_type * ) malloc ( sizeof ( person_info_type ) );
```

- Memory is deallocated by using the `free` function
  - Memory is returned to the system to be reallocated in future
  - The memory allocated to the variable `person` above can be freed by using the statement

    ```
    free ( person );
    ```

  - Since the pointer is passed using call-by-value, it is not set to `NULL`
  - `free ( NULL );` is valid and results in no action
- Caution
  1. *If* `malloc` *is used to allocate anything other than a character string, it should be typecast to avoid the type-conflict errors*
  2. *A structure's size is not necessarily the sum of the sizes of its members; this is so because of various machine-dependent boundary alignment requirements;* `sizeof` *operator resolves this problem*
  3. *Not returning dynamically allocated memory when it is no longer needed can cause the system to run out of memory prematurely – a phenomenon known as "memory leak"*
  4. *Do not attempt to* `free` *memory not allocated through* `malloc`
  5. *Do not attempt to refer to memory that has been* `free`*d*
- The function `realloc(ptr, size)`
  - Used to increase or decrease the allocated space
  - Changes the size of the block referenced by `ptr` to `size` bytes and returns a pointer to the [possibly moved] block
  - The values returned by the function is `null` if the memory allocation fails
  - If successful, the function returns a pointer to the first byte
  - The contents of the block are left unchanged up to the lesser of the new and old sizes; contents may be copied if the block needs to be moved
  - The function `free(p);` can also be written as `realloc ( p, 0 );`

Programming methodology and software engineering

- Transition from "trivial programs" to "complex programs"
  - Must develop and standardize on techniques to create sophisticated programs
  - As much art as science
  - Hacking vs development
- "Software life cycle" comprises of various activities

  - Problem analysis
  - Requirements definition
  - High-level design, or “big picture”
  - Low-level design, or “pieces of code to perform smaller tasks”
  - Implementation
  - Testing and verification
  - Delivery
  - Production
  - Maintenance

- Software engineering is the disciplined approach to different steps in the software life cycle of computer programs, to facilitate their development within the time and cost estimates, with certain tools to manage their size and complexity
- Goals of quality software

  1. It should work
     - Must work correctly (do things right) and completely (do everything)
     - The correctness and completeness is measured against the *requirements definition*
     - It should have a set of *software specifications*, including the function, input and output (values and formats)
     - It should be as efficient as possible
       * No wastage of CPU time, memory, or peripheral devices
  2. It should be easily read and understood
     - Someone other than the programmer should be able to read it and make sense out of it
     - Should not be like an entry from the “Obfuscated C Contest”
     - Take a look at the following winning entry from 1984 Obfuscated C Contest – All it does is to print “hello, world!”

       ```
       int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
       o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
       ```
  3. It should be modifiable, without spending too much time and effort
     - Software may need to be modified in every phase of its life cycle
     - The end user may change requirements even after the software has been put into production
     - Bugs always appear when you never expect them to, such as the end of a century (Year 2000 problem, Change of DST)
     - For easy modification, the software must meet the previous requirement
  4. It should be completed on time and within budget
     - You cannot get away with excuses like “computer was down” or “my car won’t start” in real world
     - Just imagine the amount of money lost by the Denver airport due to software bugs, when it was scheduled to open in the 90s

- Software development process
  - Write detailed specifications
    * A formal and complete definition of problem
    * What are the inputs? What is the format?
    * What are the outputs? Format?
    * What sort of processing is required?
    * What interactions take place to solve the problem (getting inputs and showing intermediate results/outputs)
    * What can be reasonably assumed? Will the customer object to any of the assumptions? Shall we tell the customer that we are assuming this (like two digits for the year, instead of four)?

- Solving the problem
  - I have to go from campus to Chesterfield. How do I go?

1. Florissant, I-70W, I-270S, I-40W, Clarkson
2. Florissant, I-70W, I-270S, Olive, Clarkson
3. Florissant, I-70W, I-170S, I-40W, Clarkson
4. Natural Bridge, I-170S, I-40W, Clarkson
5. Natural Bridge, Hanley, Olive, Clarkson

- There are always multiple ways (*algorithms*) to solve a problem
- Comparing algorithms
  - Amount of work done by the computer vs amount of work done by the programmer
  - Must define some *objective measures* to compare two algorithms
  - Computer science devotes a complete branch of study to the *analysis of algorithms*
  - Given two algorithms, the one with the shortest running time is definitely better *as long as the two algorithms run on the same computer, are coded in the same language, use the same compiler with the same flags to generate executables, and use the same data sets as inputs*
  - We could also count the number of instructions executed by each of the algorithms
  - The job is made simpler by counting the steps in some critical loop that seems to use the most time
  - To solve the comparison problem, find the operations that seem to cause the bottleneck (take maximum time)
- Big-Oh notation
  - O stands for *order of magnitude*
  - Generally identified with the term that increases the fastest relative to the size of the problem
  - Define a function $f(n) = n^4 + 100n^2 + 10n + 50$
  - $f(n)$ is said to be of order n^4, or $O(n^4)$
  - Reading and printing n elements from a file after opening it
  - Searching a number in a sorted array
  - Comparing rates of growth

n	$\lg n$	$n \lg n$	n^2	n^3	2^n
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4,096	65,536
32	5	160	1,024	32,768	4,294,967,296
64	6	384	4,096	262,144	∞
128	7	896	16,384	2,097,152	∞
256	8	2,048	65,536	16,777,216	∞

- Top-down design, or stepwise refinement
  - Divide and conquer approach
  - Break the problem into several subtasks, dividing each subtask into smaller subsubtasks, dividing each subsubtask into . . ., until the final subsub. . .subtask is readily solvable
  - Details are deferred as long as possible as we go from a general to a specific solution
  - The smaller units of tasks that can be readily solvable are called *modules*
  - The main module of the program becomes `main()` while the smaller modules become the functions
  - This methodology leads to *modular design*
  - Almost any program is divided into following modules

```
int main()
{
    open_files(...);
    read_data(...);
    process_data(...);
    print_output(...);
    return ( 0 );
}
```

The entire program is written easily if we can write the corresponding functions and fill in the parameter lists

- Information hiding
  - Practice of hiding the details of a function or structure, making them inaccessible to other parts of the program
  - Makes the high-level design of a program independent of the low-level design, for example, the `qsort` routine
  - Encouraged by the top-down design – do not need to know *how* a certain function is performed, as long as we know exactly *what* it does
  - Ignorance is bliss (look at `printf`)
- Abstraction
  - Separation of logical properties of data or function from their implementation in a computer program
- Designing data structures
  - Top-down approach in data structure design
    - Go from more abstract to a more concrete view of the data, deferring the details as long as possible
  - Object-oriented approach
    - Treat the logical data entity as an "object," and define a set of operations to manipulate the object
    - The data object and the operations on it form a single module of the program, with all operations being dependent on the implementation of data type
    - The combination of data type and operations will be called an *abstract data type* or ADT
    - See the way we read something using `fgets`
- Implementation
  - Coding starts at the end of design
  - Coding should be done to conform to the goals 2 and 3 of the software development
  - Well-documented programs
    - External documentation
      - Written information outside the body of the source code
      - Includes detailed specifications, history of program development and modification, top-down design, architectural diagrams, and user manuals
    - Internal documentation
      - Comments; but do not add excessive comments
      - Self-documenting code; with meaningful names for identifiers
      - Program formatting, including indentation and white space, commonly called *prettyprinting*
    - Standards for program formatting

      Capitalization. · C makes a distinction between lower case and upper case letters
      - Make all variable names and function names as lowercase
      - Make all constants all upper case
      - Mix the upper case and lower case only in comments

      Blank space. · Leave a blank space before and after each operator
      - Separate parentheses from other things by one blank space
      - Do not use more than one blank space to separate the above
      - Use blank lines liberally to break the programs into functional units

      Statements. · Each statement must be on a separate line
      - If a long statement must be broken across lines, try to find a logical place to make the break

      ```
      final_grade = ( 0.20 * mid_term_grade ) +
                    ( 0.30 * homework ) +
                    ( 0.50 * final_exam );
      ```

      Declarations and definitions. · Declarations and definitions should be one per line
      - Try to declare all variables alphabetically, or logically as per their function in the program

· Line up the variables nicely and provide a small comment explaining their use

```
char    last_name[20];       /* Last name of the customer        */
date    date_of_birth;       /* Customer's date of birth         */
float   salary;              /* Employee's base salary           */
```

Indentation. · `main`, function statements, `#include` and `#define` statements must begin in the leftmost column

· A nested function definition statement may be indented

· Each nested level must be indented by at least two spaces

· The number of spaces used for indentation must be constant

· `if` statement

```
if ( condition )
{
    ....
}
else
{
    ....
}
```

· `case` statement

```
switch ( expression )
{
case 0 :
case 1 :
    ....
default:
    ....
}
```

· `while` loop

```
while ( condition )
{
    ....
}
```

· `for` loop

```
for ( init; test; step )
{
    ....
}
```

· `do while` loop

```
do
    ....
while ( condition )
```

Testing and debugging

- Debugging
  - The process of locating the errors in a program and fixing them
  - If no bugs can be found, we proclaim the program to be correct
    - *Are we right in doing that?*
- Program verification

  - The process of determining the degree to which a software product fulfills its specifications

- Sources of bugs
  - Generally akin to looking for the criminal in a murder case
  - Look at the available evidence and deduce the location of the criminal/bug
  - May not always work in C but from experience, you can learn to look for the clues
  - Better thing – prevent as many bugs as possible through good design
  - Errors in specification and design
    - What if the details in the program description as you received them are incorrect?
    - Writing a program to wrong specifications?
    - Bugs generally found after the program has been developed
    - One of the most expensive types of bugs to be fixed
      - Much of the conceptual design and coding effort is wasted
    - Sooner you find the bug in the software development cycle, the less expensive it is to fix it
  - Compile-time errors
    - Syntax errors, or grammatical errors
    - Easily prevented
    - Try to get the syntax right the very first time
    - If you get syntactically correct but semantically wrong program, that can cause a lot of headache
  - Run-time errors
    - May make the program to crash or abnormally terminate
    - Best exemplified by ‘division by zero’ or using up unallocated areas of memory
    - Also possible because of unanticipated input values
    - Can be avoided by ‘input validation’
    - Graceful exit
      - The program should terminate with a meaningful error message
      - Exemplified by the file open being used in this class
    - Robustness
      - Ability of a program to recover from error, and continue to operate within its environment
    - Check for only those inputs that can be wrong; not applicable to input from files created by our program
    - Also be careful with parameter passing mechanism (value *v.* reference)

- Designing for correctness
  - Design the programs based on verification techniques
  - Assertions and program design
    - Assertion – *A logical proposition that can be true or false*
    - Can make assertions about the state of the program

```
sum = part + 1;     /* sum and part are integers */
```

      - We could also assert that the value of `sum` is always greater than that of `part`
    - We can make assertions about what the program can do (think of it in relation to the top-down approach)
    - Based on the assertions, prove through a logical argument that the program indeed achieves its intended goal
  - Preconditions and postconditions
    - Clarify what happens at the boundaries of a module
      - Conditions at the time of entrance into the module
      - Conditions at the time of exit from the module
    - Exact interface to the module – name and parameter list to indicate input and output
    - Any assumptions that must be true for the operation to function correctly
    - Preconditions

      · Assertions that must be true on entry into an operation for the postconditions to be guaranteed
      · Could be exemplified by

```
/**********************************************************************/
/*                                                                    */
/* qsort : Apply quick sort to an array of integers                   */
/* Inputs: array -- An array of integers                              */
/*         num_elements -- Number of elements in the array            */
/*         lower_bound -- Lower index in the array                    */
/*         upper_bound -- Upper index in the array                    */
/* Preconditions:                                                     */
/*     num_elements must be greater than zero                         */
/*     lower_bound must be greater than or equal to zero              */
/*     upper_bound must be less than or equal to num_elements         */
/*     lower_bound must be less than or equal to upper-bound          */
/* Postconditions:                                                    */
/*     array is sorted between the indices lower_bound and upper_bound  */
/*                                                                    */
/**********************************************************************/

void qsort ( int * array, int num_elements, int lower_bound, int upper_bound )
{
    ...
}
```

    ∗ Postconditions
      · Assertions that state what results are to be expected at the exit of an operation, assuming that the preconditions are true

  – Loop invariants
    ∗ Loops are known troublemakers
      · Infinite loops – if the loop control variable does not change
      · Segmentation faults – Loop goes one step too far
    ∗ Loop invariants
      · Assertion of what conditions must be true on entry into an iteration of the loop body and on exit from the loop
      · Must always be true
      · Must say something about the purpose and semantics of the loop

- Deskchecking, walk-throughs, and inspections
  – Manual verification of the design of the program
  – Write down essential data (variables, input values, function parameters, etc.)
  – Walk through the design, making changes in the data on paper
  – Look for potential trouble spots
  – Checklist for deskchecking
    1. Does each module in the design have a clear function or purpose?
    2. Can large modules be broken down into smaller pieces?
       ∗ Rule of thumb. *Each function as well as the main program should fit in one or two screenfuls.*
    3. Are all the assumptions valid? Are they well documented?
    4. Are the preconditions and postconditions accurate assertions about what should be happening in the module they specify?
    5. Is the design correct and complete, as measured against the program specifications? Are there any missing cases? Is there faulty logic?
    6. Has the design been clearly and correctly implemented? Are features of the programming language used appropriately?
    7. Is the program designed well for understandability and maintainability?
    8. Are all output parameters of functions (passed by reference) assigned values?
    9. Are parameters that return values passed as pointers?
    10. Are functions coded to be consistent with the interfaces (input, output, and intended function) shown in the design?
    11. Are the actual parameters of function calls consistent with the parameters declared in the function headings?

12. Is each data object to be initialized set correctly at the proper time? Is each data object set before its value is used?
13. Do all the loops terminate?
14. Is the design free of identifiers/constants that are not readily apparent to the reader?
15. Does each constant, type, and variable name have a meaningful name? Are comments included with the declarations to clarify the use of these data objects?

- Talking it out
  - If you get stuck with an error, it may be a good idea to show your program and read it aloud to a friend, who may or may not understand the problem
  - Talking it out generally helps the author see bugs that are hiding in there
- Walk through
  - Manual simulation of the program with sample test inputs, keeping track of the program's data by hand on paper
  - Not intended to simulate all possible test cases
- Inspection
  - A reader goes through the code line-by-line, recording errors on an inspection report

• Program testing

- Final step to execute the code with the intention of finding any errors that may still remain
- Try to "break" the code in as many different ways as possible
- Design a set of test cases such that each instance in the set tests the correctness of a certain program feature
- For each instance of a test case
  1. Determine inputs that will demonstrate the goal of the test case
  2. Determine the *expected* behavior of the program for the given input
  3. Run the program and observe the resulting behavior
  4. Compare the expected behavior and the actual behavior of the program

• Debugging with a plan

- Plan for debugging before you even run the program
- Identify potential trouble spots and insert temporary "debugging print statements" into the identified spots
- Well-placed debugging lines can help you locate the bugs during execution
- Easiest to do with C preprocessor directives (to be discussed later)
- You can use sample code as follows

```
#define DEBUG 0      /* Debugging on if 1 */

int main()
{
    ....
    if ( DEBUG )
        printf ( ... );
    ....
}
```

• Developing a testing goal

- What kind of test cases are appropriate? How many are needed?
- Should we try to test every possible input?
- Data coverage
  - *Functional domain* – Set of valid inputs
  - If the functional domain is small, a function can be verified by testing it against all possible inputs

        - This is known as *exhaustive testing*
        - Example – Linear search in an array when the element is known to exist in the array
        - What if the element may or may not exist in the array? How about the function that takes a number as input and determines whether it is an even number?
        - We may have to employ some other criterion to test such functions
        - Random testing
            - Enter different values and see if the function crashes on any one of them
            - May uncover some bugs but wastes a lot of time
            - There is no guarantee that a new input will not crash the function
        - Goal-oriented approach
            - Cover general classes of data
            - At least one example of each class of inputs, as well as boundaries and special cases
        - Other cases of data coverage
            - If the input consists of commands, each command must be tested
            - If the input is a fixed-sized array containing variable number of records, test for the maximum number of records, as well as no records (boundary conditions); also try for one more than the maximum number of records
            - Such testing is also known as *black box testing*
    - Code coverage
        - Also known as *white box testing*
        - Check inside the module to see the code being tested
        - Make sure that the test cases cuase every statement to be executed, including every *branch*
        - You could also check for every *path* or combination of branches that might be traveled
        - You can trace execution by putting `printf`s at the beginning of every branch
    - Metric-based testing
        - Testing goals are based on some measurable factors
        - Count the number of paths in the program, and count how many have been covered using the test cases
        - May not be possible to cover 100% of the paths

- Test plans
    - A document showing the test cases planned for a program or module, their purposes, inputs, expected outputs, and criteria for success
    - Required or desired level of testing should be determined and the general strategy and test cases must be planned before the testing phase begins

- Structured integrating testing
    - Aimed at integration of separately tested pieces of code
    - Top-down testing
        - Testing begins at the top levels
        - Test whether the overall logical design works and whether the interfaces between the modules are correct
        - The lower level modules are *assumed* to work correctly and are replaced by a special function (called *stub* that can be used to stand in for the lower level function
        - The stub itself may contain only a single `printf` statement to show that the module is properly reached
        - At the next level of testing, the actual modules replace the stub
        - Top-down testing gives the user the ability to control the modules to be tested in any run of the program
    - Bottom-up testing
        - The lowest level functions are tested first, using *driver* programs
        - The driver program sets up the testing environment by declaring and assigning initial values to variables, and calls the function to be tested
        - Effective in group programming environment

  - Mixed testing approaches
    * Combines the top-down and bottom-up methods

- Practical considerations
  - Level of verification depends on the program's requirement specifications as set by the customer
  - It is possible to specify higher level of verification for parts of the program

Data design

- Data from the top down
  - We always start with a logical picture of the data, for example, a list of addresses
  - The logical picture, or *data abstraction*, hides the details of how the data will be presented in the basic available types
  - We also need to define the set of operations that are needed to manipulate the logical data (printing an address)
- What is data?
  - Functions and other operations (addition or subtraction among them) are *verbs of the programming language*
  - Data are the *n*ouns of the programming language
  - *Data abstraction* is used to separate the computer's view of data (bits/bytes) from the human view (integers, floats, structures)
- Data abstraction
  - Different internal representations of integers – BCD, sign-and-magnitude binary, unsigned binary, 1's complement, 2's complement
  - *Concept* of multiplication
  - Data encapsulation
    * The user of the data does not see the implementation, but deals with it only in terms of its logical picture or abstraction
    * Separation of the representation of data from the applications that use the data at a logical level
    * Operations for encapsulated data type `int` in C (+, −, *, /, %, =, +=, -=, *=, /=, %=, &, `|`, &&, `||`, ++, `--`)
    * The implementation details, or lower levels, are hidden from the user
  - Goal of top-down design
    * Reduce complexity through abstraction
    * Protect data abstraction through encapsulation
  - Abstract data type
    * Logical picture of a data type, plus the specifications of the operations required to create and manipulate objects of the data type
- Data structures
  - A collection of data elements whose organization is characterizd by accessing operations used to store and retrieve the individual data elements
  - May be decomposed into component elements
  - Arrangement of elements in the structure is a feature of the structure that affect how each element is to be accessed
  - Both arrangement of the elements and the way they are accessed can be encapsulated
  - Exemplified by a library where user accesses books through librarian only (books may be ordered by ISBN, Dewey Decimal number, subject, last name of author, or title)
  - Data structure design can be based on three considerations

1. Application – Modeling real-life data in a specific context
2. Abstract or logical – Abstract collection of elements with corresponding access operations
3. Implementation – Specific representation and corresponding access operations in a specific programming language

- Object-oriented programming
  - Programming methodology to make computer programs more closely model the real world
  - Properties of object-oriented languages

    Encapsulation. Create an "object" by combining a data entity with the operations that manipulate it
    Inheritance. Build a hierarchy of objects, with each descendant inheriting access to all its ancestors' operations
    Polymorphism. Ability to define an operation that is shared within an object hierarchy, with an appropriate implementation of the operation that is specific to each object in the hierarchy

- Built-in structured data types
  - One-dimensional arrays

    Abstract level Abstract collection and corresponding operations
    - Structured data type made up of a finite, fixed sise collection of ordered homogeneous elements
    - Accessing mechanism is *direct access*, based on an index
    - Operations on an array
      ```
      create_array ( array, num_elements, element_type )
      access_element ( array, index )
      ```
    - How to do both the things in C?

    Application level Modeling in specific context
    - Lists of like data elements
    - Accounts in a bank

    Implementation level Specific representation and operations
    - Reserve memory area
    - Encode accessing operations
    - *Base address* – Location of the first memory cell in the array
    - *Dope vector* – Table containing information about array characteristics
    - Accessing function
      ```
      address[i] = base_address + i * sizeof ( element_type )
      ```
  - Two-dimensional arrays – Do as homework exercise
  - Accessing a particular element in an array is an $O(1)$ operation
  - Structures – Do as homework exercise
  - Unions – Do as homework exercise