Programming in C

UVic SEng 265

Daniel M. German
Department of Computer Science
University of Victoria

October 23, 2002 Version: 1.00

Parameter passing

- C implements call-by-value parameter passing

```c
int a = 5;
int b = 10;
int c;
c = maxint(a, b); /* actual parameters */
...

int maxint(int m, int n) /* formal parameters */
{
    if (m > n) {
        return m;
    } else {
        return n;
    }
}
```

Parameter passing ...

- Call-by-value semantics copies actual parameters into formal parameters.
- Fact: assignments to formal parameters are not propagated to the actual parameters
- Example:
  ```c
  int power2(float f)
  {
      if (f > /* some maximum number */) {
          return -1; /* cannot square without overflow */
      } else {
          f = f*f; return 1;
      }
  }
  
  float g = 4.0; power2(g); printf("%f\n", g);
  ```

Important

- Remember that:
  - all variables are data
  - all data resides in memory
  - every memory location has an address
- Pointer variables:
  - is a ”pointer” to a memory location
  - contains the address of another variable (other variable possibly anonymous)
- A pointer can be used for call-by-reference
Addresses, Pointers

- Compare
  
  ```c
  int x = 1;
  int y = x;
  x = 2;
  printf("y is %d\n", y);
  
  int x = 1;
  int *y = &x;
  x = 2;
  printf("*y is %d\n", *y);
  ```

- In other words, `x` is the same as `*y x`.

Notation

- `&` before a variable means get the memory location of this variable (reference)
- `*` before a any use of a variable means get the contents of memory location stored in this variable (dereference)
- Note that `*` may appear in a variable declaration and in a variable’s use; however, it has a slightly different meaning in each case.

```c
float f = 30.0;
float *g;
g = &f;
printf("%d %d\n", f, *g);
```

Pointers

- Why pointers?
  - call-by-value works well for passing in parameters, but what if we want values to be modified in the called function?
  - functions can only return a single value in return statements
  - call-by-reference semantics gets around the limitation of returning a single value.

Example

- Swap function:
  ```c
  void swap(int a, int b)
  {
    int temp = a;
    a = b;
    b = temp;
  }
  
  int x = 2;
  int y = 1;
  swap(x, y);
  printf("x == %d, y == %d\n", x, y);
  
  x == 2, y == 1
  ```
Example ...

- Why did the values not get swapped?
- The integers \(a\) and \(b\) were swapped within the scope of \(\text{swap}()\), but they were copies of the values of the \(x\) and \(y\).
- Using pointers
  ```c
  void swap (int *a, int *b)
  {
    int tmp = *a;
    *a = *b;
    *b = tmp;
  }
  swap(&x, &y);
  ```

Invalid pointers

- As a pointer contains a variable’s memory address, we must ensure the address is valid.
- C **does not check** the validity of a pointer.
- If you point to garbage (or to nothing), you will get garbage, or crash your program.
  - On Sun architecture you get this error: **segmentation fault**
- Example:
  ```c
  int *x; /* x points to what? */
  x = NULL; /* garbage */
  printf("%d\n", *x);
  ```
  Segmentation fault (core dumped)
- (NULL is defined in "stdio.h")

Notational caveats

- Pointer variables are declared by specifying the type of object they point to:
  ```c
  int *a;
  float *f;
  char *st[10]; /* array of pointers*/
  ```
- The declaration tells the compiler the kind of object the pointer points to.

Pointers and arrays

- Recall that arrays are an aggregate data type where each data element has the same **type**:
  ```c
  int grades[10];
  struct date_record info[50];
  char buffer[100];
  ```
- all elements in an array occupy contiguous memory locations
- to get the address of any data element, we use &:
  - 5th element of ”grades”: &grades[4]
  - 1st element of info: &info[0]
  - last element of ”buffer”: &buffer[99]
Pointers and arrays ...

- In C, an array variable name without the subscript represents the address of the first element

```c
char buffer[100];
char *cursor;

cursor = &buffer[0];
cursor = buffer;

*cursor = 'a';
buffer[0] = 'a';
*buffer = 'a';
```

Strings

- A C string is an array of characters terminated by the null symbol (character value 0 ‘\0’, not ‘0’)
- Pointers to character arrays (i.e., strings) are extensively used in C
- Character arrays must be large enough to store the entire string including the null terminator

```c
/* This only works when a char[] is declared, not when it is used */
char buffer[100] = "Hello, World!";
```
- what is contained in the memory allocated for buffer[100] after assigning it the above string?

Strings ...

- In C, we can manipulate pointers in many ways. Assuming char

```c
char *cp = buffer;  // same as char *cp = &buffer[0]

cp + n;  // same as &buffer[n]
*(cp + n);  // same as buffer[n]
++cp;  // same as cp = cp + 1
*++cp;  // same as cp = cp + 1, *cp
```

Functions and pointers

- As seen earlier, functions can accept pointer variables as parameters
- Functions may also return memory addresses as results
- Example:

```c
int *Max_With_Ptr(int *a, int *b)
{
    if (*a > *b) {
        return a;
    } else {
        return b;
    }
}
```
Recapping

- A pointer contains a memory address
- This memory address can point to:
  - a variable
  - a function (we will not use them in this course)
- A * in front of a pointer variable indicates a dereference
- A & in front of a variable, denotes a reference

Recap

- Therefore:
- *a dereference of a; value of what a points to
- &b memory location of variable b
- If pointer variable contains the address of a struct, extra notation is available to access struct fields
  ```c
  struct date { int dd; int mm; int yy; };  
  struct date today;  
  struct date *p_today;  
  ....  
  p_today = &today;  
  p_today->dd = 10; /* same as today.dd */  
  (*p_today).dd = 10; /* same as above */
  ```

C memory model

- Computer memory is divided into two parts: stack and heap
- Stack stores local variables, global variables, activation records/stack frames for functions
- Stack is located at the top of available memory (top addresses)
- Heap stores explicitly requested memory which must be dynamically allocated
- Heap is located at bottom of available memory (low addresses)

C memory model ...

- As the program executes, and function call depth increases, the stack grows downward
- Similarly, as memory is explicitly requested for allocation, the heap grows upward
- Eventually, if stack and heap continue to grow, all available memory will be exhausted and an out of memory condition will occur
- Every time a function call is made, a new stack frame is created and memory is allocated for the local variables of the function
Memory model ...

- Memory often needs to be allocated for storage, but the exact amount needed can only be determined at runtime
- Examples:
  - reading records from a file in order to sort them, where file size is not known at time program is written
  - creating data structures whose size vary depending on the size of the input
- One solution: write the program by hard-coding in the largest amount of memory that could possible be needed
- Problem: wasteful if program input sizes for program are almost always small

Solution

- Use heap memory for parts of programming problem where size is not known until run-time
- Use stack memory for parts where size is known at compile time
- Using stack is easy – all variables we can declare at compile time use stack memory (no extra work)
- Using heap is a bit harder
  - in Java, heap memory is automatically allocated to objects through use of the new keyword
  - also in Java, heap memory no longer used by a variable may be reclaimed for the system garbage collection

Heap memory in C

- malloc(): allocate memory in the heap
- Takes a single parameter representing the number of bytes of heap memory to be allocated
- Returns a memory address to the beginning of newly allocated memory
- If allocation fails (not enough heap memory to satisfy request), malloc() returns the NULL address (location 0)

Heap memory ...

- sizeof():
  - used to determine the number of bytes allocated for a given variable type
  - works for any data type
- combining malloc() and sizeof:
  ```c
  int *a = malloc( sizeof (int) );
  struct date *dt = malloc( sizeof(struct date) );
  char *buffer = malloc( sizeof(char)*100 );
  ```
**malloc(), sizeof()**

- Notice that we can also allocate memory for arrays using these two functions
- Example: dynamically create an array of size “n”:
  ```c
  int *grades = malloc( sizeof(int) * n );
  ```
- Once heap memory is allocated:
  - it stays allocated for the duration of the program’s execution
  - or until it is explicitly deallocated

**free()**

- A *memory leak* occurs when heap memory is constantly allocated but isn’t freed when no longer needed
- Memory leaks are almost always unintentional
- Systems with automatic garbage collection never have memory leaks
- Redundant memory is returned to heap for re-use
- Downside: garbage collection is not always under control of the programmer
- Also: some garbage collectors cannot reclaim some kinds of redundant instances of datatypes.

**Extended example**

```c
#define NUMBER 23
int *Get_Squares(int n)
{
    int *array = malloc(sizeof(int)*n);
    int i;
    for (i=0; i<n; i++) array[i] = i*i;
    return array;
}
int main(void)
{
    int *squares = NULL;
    int i;
    squares = Get_Squares(NUMBER);
    for (i=0; i<NUMBER; i++) printf("%i\n", squares[i]);
    free(squares);
}
```
**Casting in more detail**

- C does limited *type conversion* for the programmer
- Allows one data type to be forced into another
- Example:

  ```
  float x = 6.935;
  int y = x;
  printf("%d\n", y);  // prints out 6 */
  ```

- not the best result: conversion truncates float
- Solution: idiom: add 0.5 before casting (only works for positive numbers)
  ```
  int y = (x + 0.5);
  ```

**Converting chars and strings to integers**

- Recall: strings are arrays of characters
- Converting single characters to integers and vice versa is easy enough
  ```
  i = '1';
  printf(Value of i "%d", i);
  ```

- Beware, however: result is not what you might expect
  ```
  printf("%d\n", (int)'1');  // prints 49 */
  ```

- in case above, we could always ”fudge” the conversion by subtracting 48 from the casted char
- ’1’ corresponds to ASCII 49.

**Casting ...**

- The compiler will sometimes complain (with warnings turned on) if the types on either side of `'='` are incompatible
  ```
  int i = 5;
  int *ip = &i;
  char *cp;
  cp = ip;  // warning generated here */
  ```

**Converting strings to ints ...**

- Converting strings of numbers into integers is more tricky as you can’t use type casting
  ```
  char foo[] = "12345";
  int bar = (int)foo;  // warning */
  ```
Converting strings to ints...

- stdlib.h includes string-to-integer conversion functions
- atoi(char *): "a to i" converts a string into an integer
- strtol(char *): "string to long" converts a string into a long integer (8 byte on a server)

```c
#include <stdlib.h>
...
char foo[] = "12345";
int bar = atoi(foo);
printf("%d\n", bar);
```

Converting between upper & lower case

- ctype.h includes two functions for converting letters between upper and lower case
- These take an int as a parameter and return an int, but in practice, passing in a char and expecting a char in return is reasonable
- toupper(int): takes a character and returns its uppercase equivalent (if possible)
- tolower(int): takes a character and returns its lowercase equivalent (if possible)

```c
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char *s = "The Title Of Something";
    char *temp;
    for (temp = s; *temp != '\0'; temp++) {
        printf("%c", toupper(*temp));
    }
    printf("\n");
    for (temp = s; *temp != '\0'; temp++) {
        printf("%c", tolower(*temp));
    }
    printf("\n");
    return 0;
}
```

Copying & comparing strings

- string.h provides facilities for copying and comparing strings
- strcpy(char *dest, char *src): copies the contents of string “src” to the array pointed to be “dest”
- strncpy(char *dest, char *src, int n) is similar, except it takes a third parameter that specifies how many bytes to copy, at most
- strcmp(char *s1, char *s2): compares the two strings “s1” and “s2”, returning a negative, zero, or positive integer if s1 is lexicographically less then, equal to, or greater than s2.

```c
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char *s = "The Title Of Something";
    char *temp;
    for (temp = s; *temp != '\0'; temp++) {
        printf("%c", toupper(*temp));
    }
    printf("\n");
    for (temp = s; *temp != '\0'; temp++) {
        printf("%c", tolower(*temp));
    }
    printf("\n");
    return 0;
}
```
Comparing strings

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

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

    if (argc < 3) exit(1);
    cmp_result = strcmp(argv[1], argv[2]);
    if (cmp_result < 0) {
        printf("%s <%s\n", argv[1], argv[2]);
    } else if (cmp_result == 0) {
        printf("%s == %s\n", argv[1], argv[2]);
    } else {
        printf("%s > %s\n", argv[1], argv[2]);
    }
    return 0;
}
```

Copying strings

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void Print_3Versions(char *originalString) {
    char *allUpper = malloc(strlen(originalString)+1);
    char *allLower = malloc(strlen(originalString)+1);
    if (allUpper == NULL || allLower == NULL) {
        exit(-1);
    }
    strcpy(allUpper, originalString);
    strncpy(allLower, originalString, strlen(originalString)+1);
    /*... use toupper() & tolower() on copied chars ...*/
    printf("Original string: %s\n", originalString);
    printf("ALL UPPER: %s\n", allUpper);
    printf("all lower: %s\n", allLower);
    free(allUpper);
    free(allLower);
}
```

Linked lists in C

- Def: data structure consisting of series of nodes, with nodes linked together
- Nodes are structs, and links are implemented as pointer variables
- linked lists come in two flavours: single- and double-linked
- single-linked lists contain nodes composed of data elements and a pointer to the next node
- double-linked lists contain nodes composed of data elements, plus a pointer to the next node and a pointer to the previous node
- other types of linked lists exist (e.g., ”triply”, ”skip”, etc.)
- each node is an instance of a linked list ”struct” type
- as linked lists are usually dynamic, we often use malloc()

```c
malloc(sizeof(struct list_node))
```
Linked lists in C ...

- linked lists usually have a head and tail pointer, indicating the first and last nodes, respectively
- to insert a node into a single-linked list, the previous node’s "next" pointer must be updated to point to the newly inserted node, and the inserted node’s next pointer to the next node
- inserting a node into a double-linked list is the same but the affected nodes’ "prev" pointers must also be correctly updated
- removing nodes is the opposite of inserting

Using structures and pointers:

- Recall: to access variables inside a structure, use the `<structure>..<element>` syntax.
- When accessing a variable inside a structure, where the variable is pointed at by a pointer variable, use `->` notation

```c
struct list_node {
    int data;
    struct list_node *next;
}
...
struct list_node new_node =
    malloc(sizeof(struct list_node));
new_node->data = 55;
new_node->next = NULL;
```

File input and output

- like most languages, C implementations provide facilities for reading and writing files
- files are not accessed directly (i.e., sectors & tracks on a hard drive) but rather through a file control block
- this is a pointer to a file and is of type FILE
  ```c
  FILE *myfile
  ```
- the fopen() function is used to open a file; it returns a pointer to the file being opened
- different modes: "r" for reading, "w" for writing; for more details, refer to man fopen
- the fclose() function is used to close a file and flush any associated buffers
- use fgetc() to read a single character from an open file (when file is opened in "r" mode)
- similarly, fputc() will output a single character to the open file (when file is opened in "w" mode)
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int ch;
    FILE *data;
    data = fopen("data.txt", "r");
    if (data == NULL) {
        perror("Unable to open input data file ");
        exit(1);
    }
    while ((ch = fgetc(data)) != EOF) {
        printf("%c", ch);
    }
    fclose(data);
    return (0);
}

The C preprocessor is a macro processor
- It is automatically used by the C compiler to transform your program before compilation.
- It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs
- Your C program is preprocessed before it is compiled

C standard library
- ANSI standard defines a standard “library” of functions
- standard headers:
  <assert.h>  <signal.h>
  <ctype.h>   <stdarg.h>
  <errno.h>   <stddef.h>
  <float.h>   <stdio.h>
  <limits.h>  <stdlib.h>
  <locale.h>  <string.h>
  <math.h>    <time.h>
  <setjmp.h>
- See the Resources web page for a link to a list of functions available in each header file

Defining constants
- You type:
  ```c
  #define MAX_LENGTH 15
  ```
  ```c
  int main(void)
  {
      char word[MAX_LENGTH+1];
  }
  ```
- The preprocessor translates it as:
  ```c
  int main(void)
  {
      char word[15+1];
  }
  ```
- Before it is compiled
Including other files

- You type:
  ```c
  #include <stdio.h>
  int main(void)
  {
  }
  ```
- The preprocessor translates it as:
  ```c
  ... replaces #include... with the contents of the stdio.h file...
  int main(void)
  {
  }
  ```
- Before it is compiled

Defining “function-like” macros

- Function-like macros can take arguments, just like true functions.
  ```c
  #define min(X, Y) (((X) < (Y)) ? (X) : (Y))
  x = min(a, b);
  z = min(a+28, *p);
  ```
- The preprocessor generates:
  ```c
  x = ((a) < (b) ? (a) : (b));
  z = ((a + 28) < (*p) ? (a + 28) : (*p));
  ```
- Why do we need extra parenthesis in the macro definition around each parameter?