COSC 407
Intro to Parallel Computing

Topic 2 – 2: Intro to C – Part 2

1

Outline

- Previously:
  - Intro to C, Java vs C, Data types, variables, Operators
  - I/O
  - Arrays, Functions, Pointers
- Today:
  - Pointers (memory allocation, 2D arrays, functions)
  - Error Handling
  - String processing
  - struct, typedef
  - Preprocessors, Compiling C programs

2

Pointers (Review)

What is a pointer?

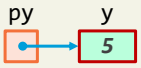
- A **pointer** is a variable that holds a **memory address**
  - A pointer is declared using *

```
type *identifier
```

* can appears anywhere between identifier and type

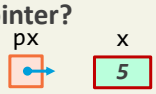
- Example1:

```
int y = 5;      // integer variable initialized to 5
int *py, *px;   // pointers to an integer
py = &y;        // p has the memory address of y
```

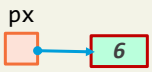


How to access a value at an address available in a pointer?

- **use the * operator which means "value at"**




```
int x = *py;    // x = the value stored at the address in p
px = &x;        // px points to x now
*px++;          // now x is 6
```



Pointers, *cont'd*

- Remember these two operators:
  - Value at address (*):
    - gives the value stored at a particular address
    - ***p is y** in the example in the previous slide
      - This is called "***dereferencing***" p
  - Address of (&):
    - gives address of a variable
      - Also called 'indirection operator'

Heap vs Stack

- In C, there are three different pools of memory available
  - **static**: global variable storage
  - **stack**: local variable storage (automatic, continuous memory)
    - A stack frame is created when you call a function
    - Used when you declare a variable inside a function
    - When you leave the function, the stack frame is 'popped off' the stack
      - Variables disappear (you don't have to worry about cleaning up
  - **heap**: dynamic storage (large pool of memory, not allocated in contiguous order).
    - **heap**: dynamic storage (large pool of memory, not allocated in contiguous order).
      - Managed by the programmer, the ability to modify it is somewhat boundless
      - Variables are allocated and freed using functions like malloc() and free()
      - Heap is large, and is usually limited by the physical memory available
      - Requires pointers to access it

A nice example:
https://craftofcoding.wordpress.com/2015/12/07/memory-in-c-the-stack-the-heap-and-static/

5

malloc() and free()

- The C function **malloc** allocates memory space given in bytes.
  - **calloc** allocates a number of bytes and *initializes them to zero*
  - malloc **returns a pointer** to the allocated memory, or **NULL** if the memory cannot be allocated.
- The C function **free** releases the allocated space

```
int* buffer = malloc(10 * sizeof(int));
int i;
for (i = 0; i < 10; i++)
        buffer[i] = i;
for (i = 0; i < 10; i++)
        printf("buffer[%d] is %d\n", i, buffer[i]);
free(buffer);
```

allocates 10 integers in the memory

Using an offset with a pointer

Free the memory

6

Difference Between [ ] and * to Declare Arrays

- This statement creates the array **on the stack**

```
int arr[3];
```

  - Once we exit the function where the array is created, the array is removed from the stack (as well as other local variables).

- This statements creates the array **on the heap**

```
int* arr = malloc(3*sizeof(int));
```

  - The memory reserved on the heap is not released until we call free()

- ***When to use which?***
  - **Stack**: for small arrays that you only need locally in a function.
  - **Heap**: for large arrays that you want to keep around for a longer time (e.g. like a global). Heap allocated arrays can also be **dynamically resized using `realloc()`** function.

7

Using Pointers with an Offset

- As you have seen in previous example, an offset can be used with a pointer using array notation to access a value in the memory

- Example:

```
int y = 10;
int* p = &y;// p = y address
printf("%d\n", p + 1);
printf("%d\n", &p[1]);
printf("%d\n", &y + 1);
printf("%d\n", p[1]);
printf("%d\n", *(p+1));
```

print (y address + 4 bytes)

print value at (y address + 4 bytes)

8

Pointers and 2D Arrays

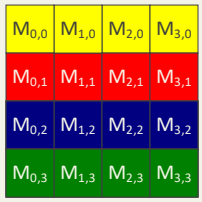
- Let's say we have two 2D arrays that we want to process in loops (e.g. initialize to 0's)

```
int  A[10][10];                          //10x10 array
int* B = malloc(100 * sizeof(int));      //10x10 array

for (int r = 0; r < 10; r++){
    for (int c = 0; c < 10; c++) {
        A[r][c] = 0; // this is ok
        B[r][c] = 0; // ERROR!
    }
}
```

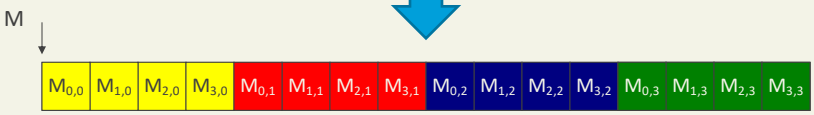
You cannot use [r][c] with dynamically allocated arrays (B can only be a pointer or a vector).
Solution: use row-major format!

9

Memory Layout of a Matrix in C

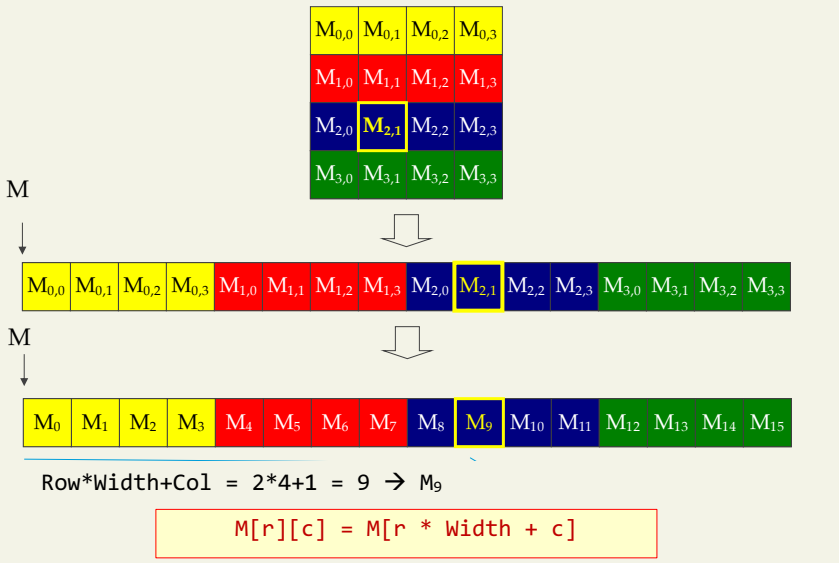


2D arrays in C are stored like this in the memory



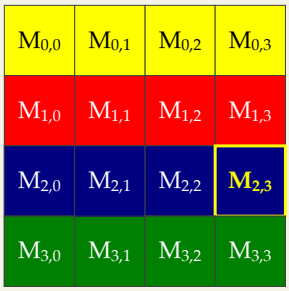
10

Row-Major Layout in C/C++



Question…

- How do we access $M_{2,3}$ if M was created using malloc?



A. M[2][3]
B. M[14]
C. M[11]
D. M[23]
E. None of the above

Using Pointers with Functions

- When passing an address as an argument and store it in a pointer parameter, the function will have direct access to the passed variable. This is called **"passing by reference"**
  - Actually passing a copy of the address

```
void incrementBy1(int*);
int main() {
   int x = 10;
   printf("x before: %d\n", x);
   incrementBy1(&x);
   printf("x after: %d\n", x);
   return 0;
}
void incrementBy1(int* xp) {
   *xp = *xp + 1;// changes directly applied to x
}
```

pass the address of x

13

Pointers to Functions

- In C, the **name of a function is a pointer to that function**.
- For example, foo is a pointer to its function.

```
int foo(){
  ------
  ------
}
```

- In C, we can create additional ***function pointers***,
  - i.e. variables that point to functions.
  - Function pointers can be used to access the functions they point to.

f →

```
int foo(){
  ------
  ------
}
```


14

Pointers to Functions

Declaration:

```
int (*f)();
```

- Here, **f** is declared as a pointer to a function that returns `int` type - the return type must be specified (e.g. int, double, etc), but the argument can be left out.
  - i.e. function pointed to by `f` could receive any arguments or no arguments.
- Note: the parentheses around ***f** are essential in the declarations.
  - Without them, i.e. `int *f()`, then we are declaring a function `f` that returns an int pointer: i.e. `int* f();`

15

Pointers to Functions, *cont'd*

Assume we have a function pointer declared as follows:

```
int (*f)();
```

and assume we have two functions:

```
int sum(int a, int b){...}
int mult(int a, int b){...}
```

We can now assign **fp** to any of the above two functions :

```
f = sum;
printf("3+4=%d", f(3,5));  //output:8
printf("3+4=%d", sum(3,5));//output:8
f = mult;                  //now fp points to mult()
printf("3*4=%d", f(3,5));  //output:15
```

This feature becomes handy when you want to ***pass a function to another***.

16

Passing a Function to Another

- In this example, process is a generic function that receives two integers and a process function
  - When we pass sum, the two integers are added
  - When we pass mult, they are multiplied

```
int sum(int a, int b) {return a + b;}
int mult(int a, int b){return a * b;}

int process(int a, int b, int(*f)()){
  return f(a, b);
}

int main() {
  int r1 = process(3, 5, sum);
  printf("3 + 5 = %d\n", r1);//out:8

  int r2 = process(3, 5, mult);
  printf("3 x 5 = %d\n", r2);//out:15

  return 0;
}
```


17

Passing a Function to Another

- In this example, process is a generic function that receives two integers and a process function
  - When we pass sum, the two integers are added
  - When we pass mult, they are multiplied

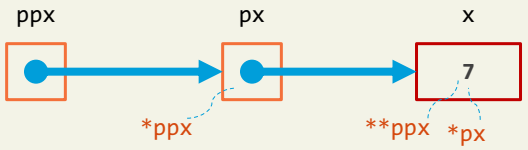
```
int sum(int a, int b) {return a + b;}
int mult(int a, int b){return a * b;}

int process(int a, int b, int f() ){
  return f(a, b);
}

int main() {
  int r1 = process(3, 5, sum);
  printf("3 + 5 = %d\n", r1);//out:8

  int r2 = process(3, 5, mult);
  printf("3 x 5 = %d\n", r2);//out:15

  return 0;
}
```

Simpler syntax

18

Aside: Pointer to a Pointer

```
int  x = 7;
int  *px, **ppx;
px = &x;      // take the address of x
ppx = &px;    // take the address of px
printf("Value available at *px = %d\n", *px );//prints 7
printf("Value available at **ppx = %d\n", **ppx);//prints 7
```



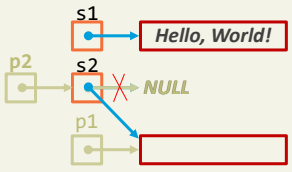
- ppx contains the address of px, which points to the location of x.
  - *px = x's value
  - **ppx = x's value
  - *ppx = the address in px
    - *ppx is px
      - so saying `px = &x` is the same as `*ppx = &x`

19

Aside: Pointer to a Pointer

- **Useful when passing a pointer by-reference to a function**
  - And the function is expected to modify the pointer's value.
  - The example demonstrate a function that stores the address of a new allocated memory within a pointer argument s2 so that s2 points to the new memory space.

```
int allocstr(int, char**);
int main(void) {
   char* s1 = "Hello, world!";        // s1 points to a string
   char* s2 = NULL;                   // s2 points to nothing (s2=0)
   if ( allocstr(strlen(s1), &s2) )
         strcpy(s2, s1);
   else
         fprintf(stderr, "out of memory\n");

   return 0;
}

// funct to allocate memory for a string and retruns 1, otherwise returns 0
int allocstr(int len, char** p2) { //p2 = address of s2 (*p2 is s2)
   char* p1 = malloc(len + 1);     // allocate space in memory
   if (p1 == NULL) return 0;       // if cannot allocate space
   *p2 = p1;       // s2 points at same memory space pointed at by p1
   return 1;}
```



20

Aside: void* pointers

- Void pointers can be assigned any data type. You can cast void pointers to other types.

```
int x = 10;  double y = 10;

int *p1;                // p1 should point to int
double *p2, *p3;                // p2 and p3 should point to
double
void *vp;               // pv can point to any type

vp = &x;                // pv can point to int
p1 = vp;                // no need to cast (p1 is int*)
p1 = (int*)vp;          // explicit casting is not needed

vp = &y;                // pv can also point to double
p2 = vp;                // no need to cast
p3 = (double*)vp;       // casting not needed

printf("x: %d\n", *p1);         // print value of x
printf("y: %f\n", *p2);         // print value of y
printf("y: %f\n", *p3);         // print value of y
```


Aside: void* pointers

- We cannot dereference a void pointer
  - Cast before we dereference a void pointer

```
int x = 10;

int *p1;                        // p1 should point to int
void *vp;                       // pv can point to any type
vp = &x;                        // pv can point to int

//*vp = 10;         //error! CANNOT dereference void pointers
*(int*)vp = 10;  //we must cast before dereferencing vp

p1 = vp;          //ok to have int* p1 = vp then dereference p1
*p1 = 10;

printf("x: %d\n", *p1);          // p1 is int*, no need to cast
printf("x: %d\n", *(int*)vp);    // again, void* needs casting
```

Aside: Functions that Return a Void Pointer

- The code below should print the address of x twice

```
void* foo(int* ptr){
     return ptr;
}

int main() {
     int x = 10;

     int* p = &x;
     printf("p: %p\n", p);
     p = foo(p);    // p is int*,  foo returns *void
     printf("p: %p\n", p);
     return 0;
}
```


Error Handling

```
#include <stdio.h> /* for fprintf and stderr */
#include <stdlib.h> /* for exit and EXIT_FAILURE */
int main(void) {
   int dividend = 50;
   int divisor = 0;
   int quotient;
   if (divisor == 0) {
        fprintf(stderr, "Division by zero! Aborting...\n");
        exit(EXIT_FAILURE); /* indicate failure.*/
   }
   quotient = dividend / divisor;
   exit(EXIT_SUCCESS); /* indicate success.*/
}
```

Strings

- A string in C is a 1-D array of characters terminated by the null character '\0'

```
char name[] = "UBC Okanagan";
```

index	0	1	2	3	4	5	6	7	8	9	10	11	12
data	U	B	C		O	k	a	n	a	g	a	n	\0

- Useful string functions (s1, s2 are strings, ch is a character)
  - **`strlen(s1);`**
    - Returns the length of s1 as integer
  - **`strcat(s1, s2);`**
    - Concatenates s2 to the end of s1.
  - **`strcmp(s1, s2);`**
    - Returns 0 if s1 and s2 are the same; -ve if s1<s2; +ve if s1>s2
  - **`strcpy(s1, s2);`**
    - Copies s2 into s1.
  - **`strchr(s1, ch);`**
    - Returns a pointer to the first occurrence of ch in s1
  - **`strstr(s1, s2);`**
    - Returns a pointer to the first occurrence of s2 in s1

Strings, cont'd

- Declaring a string with a variable length

```
int len = 10;                //10 characters
char* temp = malloc(len+1); //10 chars + null terminator
```

- There is **no** *substring* function. The following code can be used
  - Don't forget to **declare** your function before main.

```
char* substring(char* s1, int start, int end){
    int length = end - start + 1;
    char* temp = malloc(length+1);  //+1 for null terminator
    int i;
    for (i = 0; i < length; ++i)
        temp[i] = s1[start+i];
    return temp;
}
```

struct

- C has a complex data type called "**struct**" that groups a list of variables to be placed under one name in the memory
  - Unlike arrays, which hold several data items of the same type, the items in C structs may be of different types
  - A "struct" CANNOT contain any functions

Syntax:

```
struct [name]{
   type member1;
   ...
   type memberN;
} [structure variables];
```

Example:

```
struct Account {
   int id;
   char name[20];
   float balance;
};
int main(){
    struct Account c1, c2;
    //accessing members using the (.) operator
    c1.id = 212210;
    strcpy(c1.name, "John Smith");
    c1.balance = 100.5;
    //printing c1 info
    printf("ID: %d\n", c1.id);
    printf("Name: %s\n", c1.name);
    printf("Balance: %f\n", c1.balance);
    return 0;
}
```


28

typedef

- `typedef` is used to give a ***type*** a new name
  - **Synonyms for a datatype**
- Example

```
typedef unsigned char BYTE;
```

  - After this statement, you can use BYTE to declare any variables of the type 'unsigned char'

```
BYTE b1;
```


29

typedef with struct

- typedef may be used to give a name to a struct
  - Instead of declaring using "`struct point p`", use only "`point p`"

```c
#include <stdio.h>
/* Define a type 'point' to be a struct with integer members x, y */
typedef struct {
   int    x;
   int    y;
} point;
int main(void) {
    /* Define a variable p of type point. Initialize its members inline */
    point p = {1,3};
    /* Define a variable q of type point. Members are uninitialized. */
    point q;
    /* Assign the value of p to q, copies the member values from p into q. */
    q = p;
    /* Change the member x of q to have the value of 3 */
    q.x = 3;
    /* Demonstrate we have a copy and that they are now different. */
    if (p.x != q.x)
        printf("The members are not equal! %d != %d", p.x, q.x);
    return 0;
}
```

Adapted from: https://en.wikipedia.org/wiki/Struct_%28C_programming_language%29

Standard Library `stdlib.h`

- Useful values:
  - `NULL`
    - the value of a null pointer constant.
  - `EXIT_FAILURE`
    - return value for the exit function in case of failure.
  - `EXIT_SUCCESS`
    - return value for the exit function in case of success.
- Useful functions
  - Conversion from string to numbers:
    - `atoi` → returns int
    - `atol` or `strtol` → returns long int
    - `atof` or `strtod` → returns double
  - `void exit(int status)`
    - terminate the program normally.
  - `int rand(void)`
    - Returns a pseudo-random number in the range from 0 to maximum possible integer.

Preprocessors

- #include : header files
  - #include <stdio.h>
    - During compilation, this statement is replaced with the text of stdio.h file
- #define : Macro Expansions ***(constants and function-like macros)***
  - `#define PI 3.14159 /* global constant */`
  - `#define UBC "The University of British Colombia"`
  - `#define SQUARE(x) ( (x) * (x) ) /* function-like macro */`
  - `#define MAX(x,y) ( (x)>(y)? (x) : (y) )`
- #ifdef : Conditional Compilation

```
#ifdef  MACRONAME
    // Do something if MACRONAME is defined
#else
    // Do something if MACRONAME is not defined
#endif
```


32

Preprocessors, cont'd

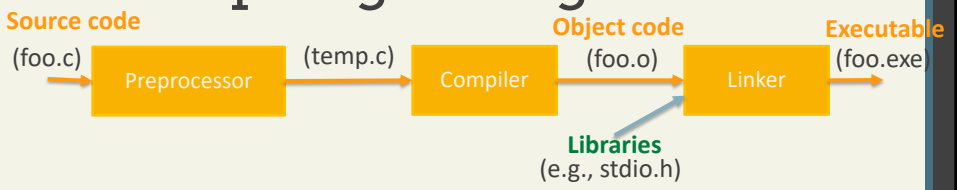
```
#include <stdio.h>

//#define DEBUG

#ifdef DEBUG
  #define mydebug(s) printf("%s\n", s);
#else
  #define mydebug(s) printf("undefined!");
#endif

int main(){
    mydebug("abc");
    return 0;
}
```

Uncomment to print "abc" (the argument)

Prints "undefined"

33

Compiling C Programs



- ***Preprocessor*** produces C code that has no preprocessing statements.
  - ***Remove comments*** **(/* */ , //)**
  - ***Include header files*** **(#include)**
  - ***Expand macros*** **(#define)**
  - ***Join continued lines*** **(\ )**
- ***Compiler and Assembler*** produce **object file** (machine code for your statements, but not for functions from the header files)
  - Think of object files as partially complete program with missing blocks of code (i.e. the code for functions from the header files).
- ***Linker*** replaces the missing parts in the object files with the appropriate machine code.

35

Keep in mind

- Always initialize (especially pointers to NULL)
  - Do not use pointers non-initialized or pointer to a memory that was released
- Don't return a function local variables by reference
- Check for errors
  - no exceptions
- An array is a pointer

36

Up Next

- Next Lecture:
  - Basics of Parallel Computing
    - Concepts and Terminology
  - Introduction to POSIX threads

38

Homework

- Read An introduction to C Programming for Java Programmer, Handley. (AGAIN)
  - Available on Connect.
- Find a C reference for later use (library, online)
  - e.g., http://www.tutorialspoint.com
  - Others?
- Practice on C!!!
  - Rewrite all exercises we had in the notes ON YOUR OWN
  - Find some online quizzes and practice

39