## Pointers, Arrays and Structures

- pointers: example, as parameters
- dynamically allocated arrays
- command line parameters
- structures, arrays of
- coding style



## Pointer Example: pointer1.c

```
#include <stdio.h>
int main(void) {
  short int *x;    /* x is a pointer to a variable of type int */
  short int a = 2;

  printf("␣sizeof(a)␣=␣%d\n", sizeof(a));
  printf("␣sizeof(x)␣=␣%d\n", sizeof(x));

  x = &a;     /* the address of a is assigned to x */
              /* x points now to where a is in memory */

  *x = *x+1; /* increment the variable that x points */
              /* to, i.e. the variable a */
  printf("␣a␣=␣%d\n",  a);
  printf("*x␣=␣%d\n", *x);
  printf("␣x␣=␣%p\n",  x);    /*note pointer format specification*/
  printf("&a␣=␣%p\n", &a);
  printf("&x␣=␣%p\n", &x);

  return 0;
}
```



## Pointers as Parameters

- passing a pointer as a parameter to a function allows the function to change the variable the pointer refers to
  - this allows us to work around the restriction that a function cannot modify any variables (other than those local to it)
- we don't change a pointer, only the variable it points to
- remember that all variables are passed-by-value,
  - i.e. the parameter values are copied to local variables when a function is called
  - a memory address is just a value, so a copy of a pointer is just as good as the original pointer itself



## Pointer Example: pointer3.c

```
#include <stdio.h>
void nextYear(int *y);

int main(void) {
  int thisYear;

  thisYear = 1981;
  nextYear(&thisYear);

  printf("Year:␣%d\n", thisYear);

  return 0;
}

void nextYear(int *y) {
  *y = *y + 1;
}
```

## Dynamically Allocated Arrays

- array names are effectively pointers
- thus a pointer variable set to an address in dynamically allocated memory can be used as an array
  - the **stdlib** library `void *malloc(size_t size)` can be used to allocate `size` contiguous bytes of such memory
    (n.b. `size_t` = `unsigned int` on 32-bit systems)
  - e.g. allocArray.c

```
int i, n, *A, si;
scanf("%d", &n);
A = (int *) malloc(n * sizeof(int)); // allocates n elements
for (i=0; i<n; i++)
  A[i] = i;  // alt: *(A+i) = i;
```

- should deallocate when finished by `free(A);` (avoid memory leaks!)
- pitfalls: (1) `A[-1] = 0; A[n] = ...;`, (2) `free(A+1);`, (3) `free(A); free(A);`, (4) `free(A); ... x = A[i];`
  - the MCheck library (`-lmcheck`) can check for these (upon `free()`)
- Q: what are the advantages over static allocation?



## Command Line Parameters

- recall Unix processes are invoked by naming the corresponding executable program and suppling an optional number of string parameters (arguments)
  - e.g. `gcc -Wall -o x x.c`
  - convention: parameters preceded by '-' are called options (the parameter(s) following may be associated with this option)
  - a process can thus be thought of as a mathematical function
    (domain: array of strings, range: integer)
- by declaring `int main(int argc, char *argv[])`, can access these through `argv[0]`, `argv[argc-1]` e.g. printArg.c

```
#include <stdio.h>
int main(int argc, char *argv[]){
  int i;
  printf("number of arguments=%d\n", argc);
  for (i=0; i<argc; i++){
    printf("argument %i value: %s\n", i, argv[i]);
  }
  return 0;
}
```



## User Defined Data Types: Structures

- a structure is a collection of logically related data items of different types grouped together as a single data type

```
struct structure_tag {   // structure_tag is name of structure
 variable_declarations ; // set of data type declarations ...
};                       // making up the structure
```

- e.g. consider an employee record:

```
struct employee {
  char name[40]; /* an array member */
  int code;      /* an int type member */
  float salary;  /* a float type member */
};
```

  - to declare one or more variables of this type:

```
struct employee person1, person2, person3;
```

  - members (fields) are accessed using the dot operator and component name

```
strcpy(person1.name, "Jones");
person1.code = 387;
person1.salary = 36000.00;
```

- unlike arrays, members can be heterogeneous and are accessed by name



## Structures: Operations

- the `sizeof` operator gives the number of bytes used for a structure:

```
printf("size of struct employee = %d\n", sizeof(person1));
```

- structures can be initialised upon declaration using:

```
struct employee person1 = {"Jones", 387, 36000.00};;
```

- we can define pointers to structures:

```
struct employee person1, *pEmployee;
pEmployee = &person1;
```

- C provides a special operator `->` called the arrow operator to access structure members

  `pEmployee->name` is same as `(*pEmployee).name;`

- cannot have function members (c.f. Java classes), but can have function pointer members...

## Arrays of Structures and Linked Lists

- we can have arrays of structures, e.g. `struct employee person[30];`
- we can also nest structures:

```
struct FEIT {
  char dean[40];
  struct employee dcsPerson[30];
  struct employee engPersons[30];
}
```

- more interestingly we can build linked lists:

```
struct employee {
  char name[40];
  float salary;
  struct employee *next; /* pointer to next employee */
};
```



## Coding Style

- why?
  - if your code is easy to read, it will be easier to work with (debug or extend) later
  - some C code in Unix has been in use for 30 years!
    Even code in the Linux kernel is now more than 10 years old.
  - some code in Windows dates back to 1981 (MS-DOS)
- suggestions:
  - `/* use explanatory but non-trivial comments */`
  - successively indent code within code-blocks (editors like `emacs` or `kate` can help with this)
  - break code into related chunks (logical blocks) separated by blank lines – white space costs nothing, so use it!
  - use meaningful variable names (`i`, `j` are often used for loop indices, so are OK)
  - use `( )`'s if there is any lack of clarity
  - avoid assignments and `i++` within expressions
  - use defensive programming, e.g. `assert()` what you think is true



## Coding Style: The Good and the Bad/Ugly

```
// changes the characters in string[] from lower to upper case.
// Assumes all characters are originally lower case letters.
void uppercase(char string[]) {
  int i = 0;

  while (string[i] != '\0') {
    assert((string[i] >= 'a') &&
           (string[i] <= 'z'));
    string[i] = string[i] - 'a' + 'A';
    i = i + 1;
  }
} //uppercase()
```

is (almost!) functionally equivalent to:

```
void uppercase(char *q) {
  int 0; for(0=1>2;*(q+0);*(q+0++)-='␣');
}
```