More C Topics

- `typedef`
- void and const
- Pointer arithmetic
- Mathematical Functions and Operators in C
- `#define` macros with arguments
- C99 vs. C

typedef

- Define new name for existing type, e.g. `typedef int int_32`
- New name comes last in definition
- One frequent use: machine-dependent types `int_32, int_64, uint_32,...`
- Other use: give a name to structs (see next slide)
- Note: typedef does NOT give you a new type, it is just an abbreviation for an existing type.
- The C standard library defines a few typedef's, e.g. `size_t` for size of objects in malloc, qsort, ...

typedef (2)

```
/* without typedef: always need keyword struct
struct point1
{
int x, y;
};

struct point1 p1, q1;

/* with typedef */
typedef struct
{
int x, y;
} point2;

point2 p2, q2;
```

typedef (3)

- Why not just use `#define`???
- Subtle difference - typedef is safer, since compiler is more powerful than preprocessor
- Example (from King): pointer type
- In `int *a, b, c;` - only `a` is a pointer.
- Need to write `int *a, *b, *c;`
- `#define IntPtr int *`
- `IntPtr a, b, c;` This fails, as above.
- `typedef int * IntPtr;`
- `IntPtr a, b, c;` This works, a, b, c are all pointers.

void pointers

- Usually, pointers are to a specific type of data, e.g. `int *` or `char **` or `struct pair *`.
- A `void *` is a raw memory address, does not tell us the type of data stored at that address.
- `void *` is used for low level memory manipulation in C, and to achieve "general" code that works for any type - for example, malloc, calloc, free use `void*`
- Typically we need to know what type of data is stored, and use a cast to convert to the right type. Example: compare function in qsort.

The Keyword const

- Usually, the value of variables and parameters can be changed by assignment. `const` is used to mark variables or parameters that should not change.
- Example:

```
int a = 5;
a = 7; /* OK, changed value stored in a */
const int b = 5;

b = 7; /* error:  assignment of read-only variable */
```

- Example: qsort uses `int(*compar)(const void *, const void *)`. The arguments are pointers to two data items. Const means a "promise" or "contract" that compar should not modify the data.

The Keyword const (2)

- Caution: even if data is declared const, C does not guarantee that data will remain constant. The compiler can catch only simple cases. Can use a cast to "cast away const", but this is dangerous and may result in incorrect programs.
- Compiler can use const information to optimize a program, or put data in read-only memory.
- Often used for tables of data that are only initialized but never changed.
- Use const as often as possible, it makes the code safer (compiler can catch some bugs) and improves readability.
- Pass by value vs pass by pointer to const: Pass by value is clearer, pointer to const more efficient for large data.

The Keyword const (3)

- Syntax with pointers: `const int *p` means that the integer *p is constant, but the pointer p may be modified, to point to a different variable of type `const int`.
- `int * const p` means that p is constant, but *p may be changed. Rarely useful.
- `const int * const p` means both p and *p are const.

Pointer arithmetic

- You can add pointers (to an array) and integers
- `p + n` is equivalent to `&p[n]`
- `p[n]` is equivalent to `*(p+n)`
- In terms of addresses, `p+n` evaluates to `p + n*sizeof(type that p points to)`
- incrementing a pointer makes it point to the next array element
- decrementing a pointer makes it point to the previous array element
- As always, be careful to remain within array bounds.

Pointer arithmetic (2)

Example: using a pointer to step through an array.

```
#define NU_ELEMENTS 6
int a[NU_ELEMENTS] = {2, 4, 6, 8, 10, 12};
int *p = a;
printf("Forwards: ");
for(p=a; p < a+NU_ELEMENTS; ++p)
    printf("%d ", *p);
printf("\nBackwards: ");
for(p=a+NU_ELEMENTS-1; p >= a; --p)
    printf("%d ", *p);
printf("\n");

% gcc pointers.c
% ./a.out
Forwards: 2 4 6 8 10 12
Backwards: 12 10 8 6 4 2
```

Pointer arithmetic (3)

- Example: sort elements with index 5..10 in array `a`
- `qsort(a+5, 10-5+1, sizeof(int), compare_int)`
- Comment: iterators in C++ and other languages are a generalization of this technique.

Operators and Mathematical Functions in C

- Similar to other languages (Details: King: Chapter 4)
- Arithmetic: +, -, *, /
- Precedence: *, / higher than +, -
- many other operators (some later in this course)
- King, Appendix A, has full list of operator precedences
- assignment: a = b
- compound assignment: a += b computes a = a + b
- most operators have a compound form: +=, -=, *=, /=,...
- increment, decrement: ++, –

Mathematical Functions and Operators in C (2)

- Modulo, remainder: a % b computes remainder of integer division a/b
- Careful if a or b are negative! See King p. 54
- non-integers: use `fmod` in math.h
- Power. `pow(a, b)` in math.h computes a^b.
- standard math functions log, exp, sin, cos, tan, ...
- See King 23.3 about math.h header

`#define` macros with parameters

- `#define` macros can take parameters (King 14.3)
- `#define print_int(x) printf("%d", (x))`
- preprocessor replaces text, substitutes argument(s)
- e.g. `print_int(5)` becomes `printf("%d", (5))`
- Can have more than one argument
- Example: `#define print_ints(x, y, z) printf("%d%d%d", (x), (y), (z))`

`#define` macros with parameters (2)

- Parameterized macros look like function calls, but they are not
- Caution: always enclose each parameter and whole macro in brackets
- Example: `#define add(x, y) x + y` fails (why?)
- Example: `#define add(x, y) (x + y)` still fails (why?)
- Example: `#define add(x, y) (x) + (y)` also fails (why?)
- Example: `#define add(x, y) ((x) + (y))` works
- C99, C++ have `inline` functions as a better solution

C99 vs Classical C

- Classical C: C89 standard, still most popular
- C99 has many small improvements and some bigger changes. See King Appendix B for a long list
- Some highlights:
  - C++ style comments starting with
  - Variable-length arrays (King 8.3)
  - New data types: booleans and complex numbers
  - inline functions
  - many more math functions

C99 vs Classical C (2): Variable-length arrays

- Example: array dimension given at runtime

```
int main( int argc, char * argv[] )
{
    long n = atoi(argv[1]);
    long number[n];
    ....
}
```

- Restrictions:
  - can not be static storage
  - can not have an initializer
- Also extends to multi-dimensional arrays (not covered in 201)

C99 vs Classical C (3): inline functions

- Example:

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

- See King 18.6
- safer than macros
- can be as efficient
- `inline` is only a hint to the compiler, compiler can choose not to inline
- A little tricky to use correctly - see examples in King textbook