# include <stdio.h>

int main() {
    int i;
    extern int gcd(int x, int y);
    for (i = 0; i < 20; i++)
        printf("gcd of 12 and %d is %d\n", i, gcd(12, i));
    return (0);
}

int gcd(int x, int y) {
    int t;
    while (y) {
        t = x; x = y; y = t % y;
    }
    return (x);
}
About C

• Similar to Java - Java took best of C
• `#include` - use declarations of functions
• `main()` returns int, the exit status
• Functions must be
  – declared - tells compiler how to use function
  – defined - creates the item
• Declarations must appear before code
Basic Control Structures

- Functions - can omit `extern` declaration
- `for` loop - like Java
  - body is one statement
  - braces `{ }` enclose blocks
  - blocks introduce scope level
  - can't mix declarations and non-declarations
    - `for (int i ...` illegal in ANSI C
More about C

• Uninitialized variables have no default value!
• No run-time checking!
• No polymorphism (printf format strings)
• No objects (C predates object-oriented)

Compile: gcc -Wall -g -o gcd gcd.c
C data types

• basic types and literals (King: Ch 7)

```c
int i = 38;        long el = 38L;
int hex = 0x2a;    int oct = 033;
printf("i = %d, el = %ld, hex = %d, oct = %d\n",
    i, el, hex, oct);

i = 38, el = 38, hex = 42, oct = 27

double d1 = 0.3;   double d2 = 3.0;
double d3 = 6.02e23;
printf("d1 = %f, d2 = %f, d3 = %e\n", d1, d2, d3)

d1 = 0.300000, d2 = 3.000000, d3 = 6.020000e+23
```
Data Type Conversion

• The expression on the right side is converted to the type of the variable on the left.

```c
char c;
int i = c;    /* c is converted to int */
double d = i; /* i is converted to double */
```

• This is no problem as long as the variable's type is at least as “wide” as the expression.

```c
char c = 500;    /* compiler warning */
int k = d1;
printf("c = %c, k = %d\n", c, k);
c = , k = 0
```
Data Type Capacity

• What happens when the following code is executed?

```c
char c = 127;
int d;

printf("c = %d\n", c);
c++;

d = 512 / c;
printf("c = %d, d = %d\n", c, d);
```
Mixed Mode Arithmetic

double m = 5/6; /* int / int = int */
printf("Result of 5/6 is %f\n", m);
Result of 5/6 is 0.000000

double n = (double)5/6; /* double / int = double */
printf("Result of (double)5/6 is %f\n", n);
Result of (double)5/6 is 0.833333

double o = 5.0/6; /* double / int = double */
printf("Result of 5.0/6 is %f\n", o);
Result of 5.0/6 is 0.833333

int p = 5.0/6; /* double / int = double but then converted to int */
printf("Result of 5.0/6 is %d\n", p);
Result of 5.0/6 is 0
Memory model

- Memory is just a sequence of bytes
- A memory location is identified by an address.

<table>
<thead>
<tr>
<th>Logical address</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>Static Data</td>
<td></td>
</tr>
<tr>
<td>Dynamic Data</td>
<td></td>
</tr>
<tr>
<td>Unused Logical Address Space</td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td></td>
</tr>
</tbody>
</table>

\[2^{32} - 1\]
int x = 10;
int y;

int f(int p, int q) {
    int j = 5;
    return p * q + j;
}

int main() {
    int i = x;
    y = f(i, i);
    return 0;
}
Arrays

- Arrays in C are a contiguous chunk of memory that contain a list of items of the same type.
- If an array of ints contains 10 ints, then the array is 40 bytes. There is nothing extra.
- In particular, the size of the array is not stored with the array. There is no runtime checking.
int x[5];
for (i = 0; i <= 5; i++) {
    x[i] = i*i;
}

- No runtime checking of array bounds
- Behaviour of exceeding array bounds is “undefined”
  - program might appear to work
  - program might crash
  - program might do something apparently random
Initializing arrays

Declaration/Definition

```c
int a[10]; /* declare 'a' as an array of 10 ints*/
sizeof(a) == 10 * sizeof(int) == 40;
```

Static initialization:

```c
char letters[4] = {'a', 'q', 'e', 'r'};
```

Initialization loop:

```c
for(i = 0; i < N; i++) {
    a[i] = 0;
}
```
Arrays

- Warning: It is the programmer's responsibility to keep track of the size of an array.
- Often define a maximum size.
- Pre-processor directives used for constants:
  - E.g., `#define MAXSIZE 30`
Pointers

• A pointer is a higher level version of an address.
• A pointer has type information.

```c
int i;
int *p; /* declare p to point to type int */
*p = i; /* dereference p – set what p point to*/
p = &i /* Give p the value of the address of i*/
char *c = p; /* Warning: initialization from incompatible pointer type */
```
Important!

• `int *p;`
• Memory is allocated to store the **pointer**
• No memory is allocated to store what the pointer points to!
• Also, `p` is **not** initialized to a valid address or null.
• I.e., `*p = 10;` is wrong unless memory has been allocated and `p` set to point to it.
int i = 19;
int *p;
int *q;
*p = i; /*error*/
q = &i

Symbol Table

<table>
<thead>
<tr>
<th></th>
<th>0x80493e0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0x80494dc</td>
</tr>
<tr>
<td>q</td>
<td>0x80494e0</td>
</tr>
</tbody>
</table>

0x80493e0 19
0x80494dc ?
0x80494e0 0x80493e0
int i = 19;
int *p;
int *q;

q = &i
p = (int *)malloc(sizeof(int));
*p = i;

Symbol Table

<table>
<thead>
<tr>
<th>i</th>
<th>0x80493e0</th>
<th>0x80493e0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0x80494dc</td>
<td>0x8049530</td>
</tr>
<tr>
<td>q</td>
<td>0x80494e0</td>
<td>19</td>
</tr>
</tbody>
</table>