Structures and Basic File I/O

MATH 5061: Fundamentals of Computer Programming for Scientists and Engineers

Dr. Richard Berger
richard.berger@temple.edu

Department of Mathematics
Temple University

11/01/2016

TEMPLE UNIVERSITY

Outline

Outline

Declaration

Declaration

```
struct Vector {
    double x;
    double y;
    double z;
};
```

Definition

```
// create Vector on stack
Vector v1;

// create Vector on heap
Vector * v2 = new Vector;
```

- Structures enable the definition of custom data types built out of primitive data types
- they allow you to group data, which belongs together, to one type
- once declared, you can use a **`struct`** like any other data type
- the **`sizeof`** operator returns the number of bytes the structure occupies in memory, e.g., **`sizeof`**`(Vector)` returns 24 bytes

Accessing elements of a struct

Direct access

```
Vector v;
v.x = 1.0;
v.y = 0.0;
v.z = 0.0;
```

Pointer access

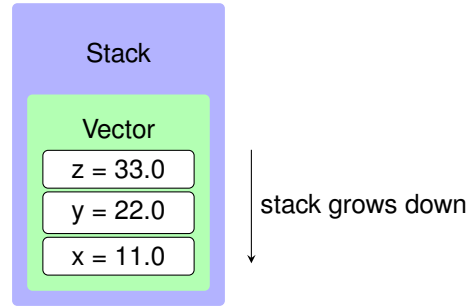
```
Vector * pv = &v;
pv->x = 1.0;
pv->y = 0.0;
pv->z = 0.0;

Vector* pv2 = new Vector;
pv2->x = 0.0;
pv2->y = 1.0;
pv2->z = 0.0;
```

References

```
Vector& v2 = v;
v2.x = 0.0;
v2.y = 1.0;
v2.z = 0.0;

Vector& v3 = *pv;
v3.x = 0.0;
v3.y = 0.0;
v3.z = 1.0;
```

Structs on the stack

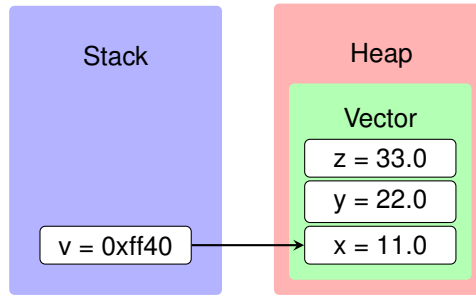
```
int main()
{
    Vector v;

    v.x = 11.0;
    v.y = 22.0;
    v.z = 33.0;

    printf("&v.x = %p\n", &v.x);
    printf("&v.y = %p\n", &v.y);
    printf("&v.z = %p\n", &v.z);
    return 0;
}
```



- like in arrays, addresses of elements inside a **`struct`** grow towards larger addresses

Structs on the heap

```
int main()
{
    Vector * v = new Vector;
    v->x = 11.0;
    v->y = 22.0;
    v->z = 33.0;

    ...
    delete v;
}
```



TEMPLE UNIVERSITY

Composing more complex data types

```
struct Date {
    int day;
    int month;
    int year;
};

struct Person {
    char first_name[50];
    char last_name[50];
    Date day_of_birth;
};
```

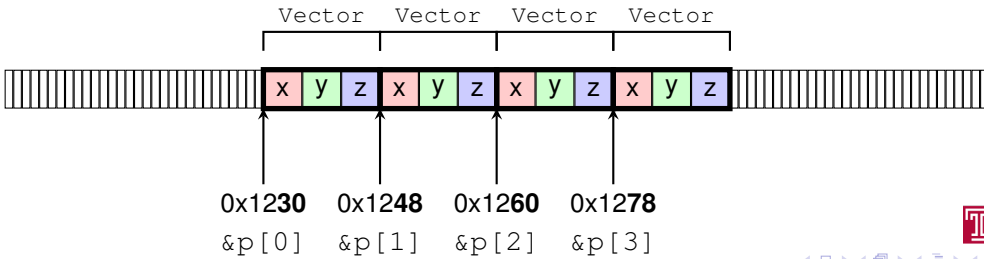
- Structs themselves can be combined inside other structs to build more complex data types

```
Person p;

strcpy(p.first_name, "Albert");
strcpy(p.last_name, "Einstein");

p.day_of_birth.day = 14;
p.day_of_birth.month = 3;
p.day_of_birth.year = 1879;
```

Array of structs

- Structs can also be elements of arrays
- All rules we learned so far about element access, pointers, and arrangement in memory still apply

```
Vector p[4];
p[0].x = 1.0;
p[0].y = 0.0;
p[0].z = 0.0;
```



Example: Stack using arrays

Example: Stack using arrays

```
struct Stack {
    int data[10];
    int head;
};

void init(Stack & stack) {
    stack.head = -1;
}

bool push(Stack & stack, int value) {
    if(stack.head + 1 == 10) {
        printf("Stack is full!\n");
        return false;
    }
    stack.data[++stack.head] = value;
    return true;
}
```

```
int pop(Stack & stack) {
    if(stack.head < 0) {
        printf("Stack is empty!\n");
        return -1;
    }
    return stack.data[stack.head--];
}
```

```
Stack s;
init(s);
push(s, 10);
push(s, 20);
push(s, 30);
pop(); // 30
pop(); // 20
pop(); // 10
```

TEMPLE UNIVERSITY

Example: Stack with growing array

```
struct Stack {
    int * data;
    int size;
    int head;
};

void init(Stack & stack) {
    stack.data = new int[10];
    stack.size = 10;
    stack.head = -1;
}

void grow(Stack & stack) {
    int new_size = stack.size * 2;
    int* new_data = new int[new_size];
    memcpy(new_data, stack.data,
           stack.size*sizeof(int));
    delete [] stack.data;
    stack.data = new_data;
    stack.size = new_size;
}
```

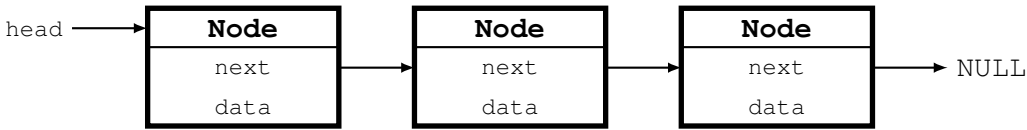
```
bool push(Stack & stack, int value) {
    if(stack.head + 1 > stack.size) {
        grow(stack);
    }
    stack.data[++stack.head] = value;
    return true;
}

int pop(Stack & stack) {
    if(stack.head < 0) {
        printf("Stack is empty!\n");
        return -1;
    }
    return stack.data[stack.head--];
}
```

TEMPLE UNIVERSITY

Linked List

- A linked-list is a data structure built out of multiple blocks of memory which are connected through pointers

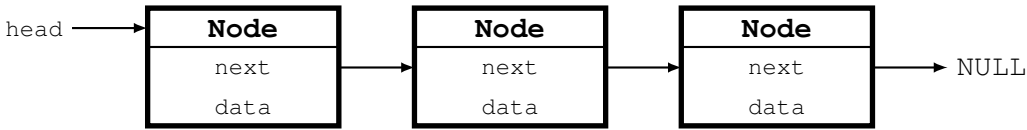
Single Linked-List



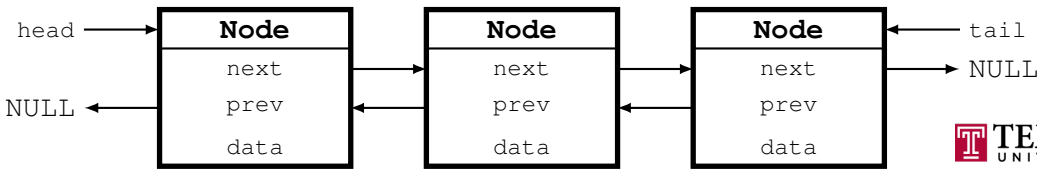
Linked List

- A linked-list is a data structure built out of multiple blocks of memory which are connected through pointers

Single Linked-List



Double Linked-List



TEMPLE UNIVERSITY

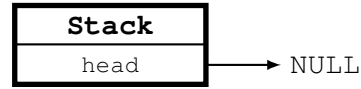
Implementing a stack using a linked list

```
struct Node {
    Node * next;
    int value;
};

struct Stack {
    Node * head;
};
```

- To implement a linked list we define two types
- The `Node` type contains one piece of data and the pointer to the next node
- The `Stack` type just stores a pointer to a node which starts the chain of nodes

Linked List: Pushing elements on a stack

```
Stack stack;
stack.head = NULL;

push(stack, 11);
push(stack, 22);
push(stack, 33);
```

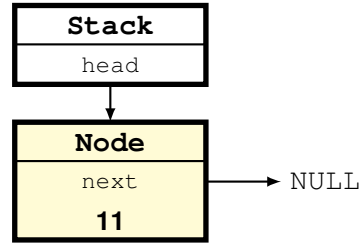



```
void push(Stack & stack, int value) {
    Node * n = new Node;

    // store current head as next
    n->next = stack.head;
    n->value = value;

    // put new element as head
    stack.head = n;
}
```

Linked List: Pushing elements on a stack

```
Stack stack;
stack.head = NULL;

push(stack, 11);
push(stack, 22);
push(stack, 33);
```

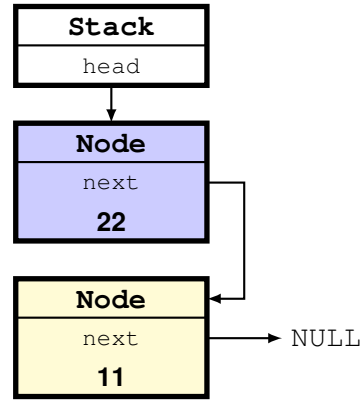
```
void push(Stack & stack, int value) {
    Node * n = new Node;

    // store current head as next
    n->next = stack.head;
    n->value = value;

    // put new element as head
    stack.head = n;
}
```



Linked List: Pushing elements on a stack

```
Stack stack;
stack.head = NULL;

push(stack, 11);
push(stack, 22);
push(stack, 33);
```

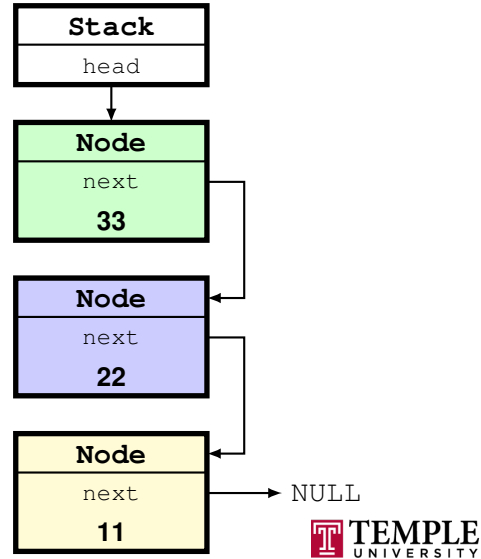
```
void push(Stack & stack, int value) {
    Node * n = new Node;

    // store current head as next
    n->next = stack.head;
    n->value = value;

    // put new element as head
    stack.head = n;
}
```



Linked List: Pushing elements on a stack

```
Stack stack;
stack.head = NULL;

push(stack, 11);
push(stack, 22);
push(stack, 33);
```

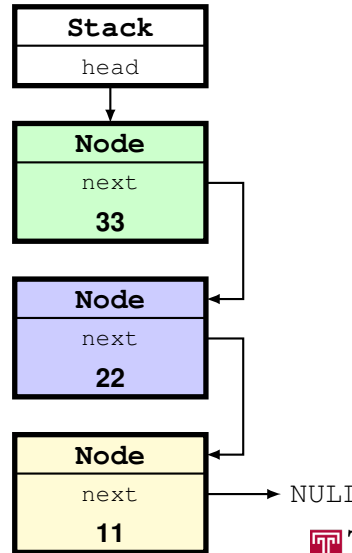
```
void push(Stack & stack, int value) {
    Node * n = new Node;

    // store current head as next
    n->next = stack.head;
    n->value = value;

    // put new element as head
    stack.head = n;
}
```



Linked List: Popping elements from a stack

```
pop(stack); // returns 33
pop(stack); // returns 22
pop(stack); // returns 11
```

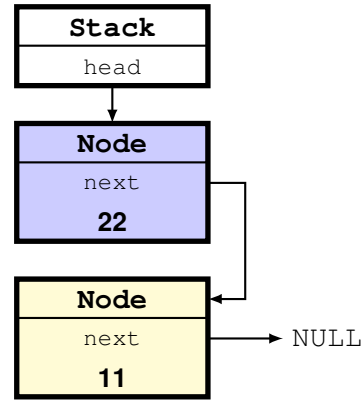
```
int pop(Stack & stack) {
    Node * n = stack.head;

    // retrieve value from head
    int value = n->value;

    // make next new head of stack
    stack.head = n->next;

    // delete node
    delete n;

    return value;
}
```



Linked List: Popping elements from a stack

```
pop(stack); // returns 33
pop(stack); // returns 22
pop(stack); // returns 11
```

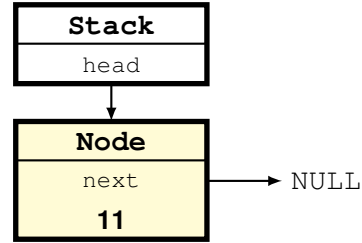
```
int pop(Stack & stack) {
    Node * n = stack.head;

    // retrieve value from head
    int value = n->value;

    // make next new head of stack
    stack.head = n->next;

    // delete node
    delete n;

    return value;
}
```



Linked List: Popping elements from a stack

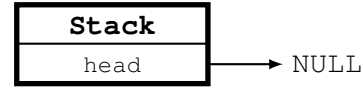
```
pop(stack); // returns 33
pop(stack); // returns 22
pop(stack); // returns 11
```

```
int pop(Stack & stack) {
    Node * n = stack.head;

    // retrieve value from head
    int value = n->value;

    // make next new head of stack
    stack.head = n->next;

    // delete node
    delete n;

    return value;
}
```



Linked List: Popping elements from a stack

```
pop(stack); // returns 33
pop(stack); // returns 22
pop(stack); // returns 11
```




```
int pop(Stack & stack) {
    Node * n = stack.head;

    // retrieve value from head
    int value = n->value;

    // make next new head of stack
    stack.head = n->next;

    // delete node
    delete n;

    return value;
}
```

```
struct Vector {
    double x; // 8 bytes
    double y;
    double z;
    int w;    // 4 bytes
};
```

- Q: What is **sizeof**(Vector) now?

TEMPLE UNIVERSITY

```
struct Vector {
    double x; // 8 bytes
    double y;
    double z;
    int w;    // 4 bytes
};
```

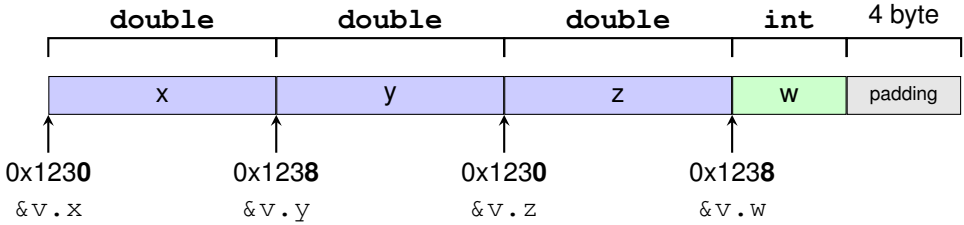
- Q: What is **sizeof**(Vector) now?
- Let's check!

TEMPLE UNIVERSITY

```
struct Vector {
    double x; // 8 bytes
    double y;
    double z;
    int w;    // 4 bytes
};
```

- Q: What is **sizeof**(Vector) now?
- A: 32 bytes?!? What's going on?

Struct Padding & Alignment



- For performance reasons compilers add padding to structs in order to align the addresses of basic data types. In this case it makes sure it is a multiple of **`sizeof(double)`**.
- This is an advanced topic, in many cases you do not have to know this.
- But you should at least be aware of it

Remember

The size of a structure is not only dependent on the individual member data types but also on how the compiler pads and aligns it. Always use **`sizeof`** to get the actual size, do not guess.

Struct Padding & Alignment

Forcing compact structs

- compilers allow you to turn off this behaviour, but it's not standardized
- here is how you can do it with the GCC compiler

```
struct Vector {
    double x;
    double y;
    double z;
    int w;
} __attribute__((packed));
// now sizeof(Vector) == 28
```

Outline

Reading a text file character by character

```
#include <stdio.h>

int main() {
  FILE* infile = fopen("a.txt", "r");
  char c;

  while ((c = getc(infile)) != EOF){
    printf("%c", c);
  }

  fclose(infile);
  return 0;
}
```

- `fopen()` opens a file in a given mode ("r" = read, "w" = write, "a" = append)
- it returns a file handle of type `FILE*`
- all subsequent operations on the file use this handle
- `getc()` reads a single **`char`** (1 byte), or `EOF` if the end of file is reached
- finally a file is closed using `fclose()` on its handle

Writing a text file character by character

```
#include <stdio.h>

int main() {
  FILE* in  = fopen("a.txt", "r");
  FILE* out = fopen("copy.txt", "w");
  char c;

  while ( (c = getc(in)) != EOF ) {
    putc(c, out);
  }

  fclose(out);
  fclose(in);
  return 0;
}
```

- `putc()` writes a character to a file

TEMPLE UNIVERSITY

Read a text file line by line

```c
#include <stdio.h>

int main() {
  const int MAX_LINE = 1024;
  char line[MAX_LINE];

  FILE* infile = fopen("a.txt", "r");
  char c;
  int nline = 1;

  printf("Output:\n");

  while(fgets(line, MAX_LINE, infile)) {
    printf("%4d: %s", nline, line);
    nline++;
  }

  fclose(infile);
  return 0;
}
```

- `fgets()` read a characters until it reaches a newline, EOF or the end of the buffer.
- also using a new construct here: **`const`**. This way we define a constant value and tell the compiler this variable is not allowed to change. The compiler will produce an error if you try to change it afterwards.

Writing text to a file

```c
#include <stdio.h>

int main() {
  FILE* outfile = fopen("a.txt", "w");

  fputs("Line 1\n", outfile);
  fputs("Line 2\n", outfile);

  fclose(outfile);
  return 0;
}
```

- `fputs()` writes a string without its 0-byte to the file
- it also does not write a newline, you have to add it yourself

Writing formatted text to a file

```c
#include <stdio.h>

int main() {
  const int MAX_LINE = 1024;
  char line[MAX_LINE];

  FILE* infile = fopen("a.txt", "r");
  FILE* outfile = fopen("b.txt", "w");
  char c;
  int nline = 1;

  while(fgets(line, MAX_LINE, infile)) {
    fprintf(outfile, "%4d: %s", nline, line);
    nline++;
  }

  fclose(outfile);
  fclose(infile);
  return 0;
}
```

- `fprintf` writes a formatted string to a file
- it works like `printf`, but takes a file handle as first parameter

TEMPLE UNIVERSITY

Binary I/O

Write binary values to file

```
// open file for writing
FILE * outfile = fopen("data.bin", "w");

// write single integer
int value = 50;
fwrite(&value, sizeof(int), 1, outfile);

// write float array
float array[100];
fwrite(array, sizeof(float), 100, outfile);

// close file
fclose(outfile);
```

- `fwrite` writes a sequence of bytes to a file
- the first parameter is the pointer to the beginning of the sequence
- the second parameter is the element size
- the third parameter is the number of elements
- finally the last parameter is the file handle

Binary I/O

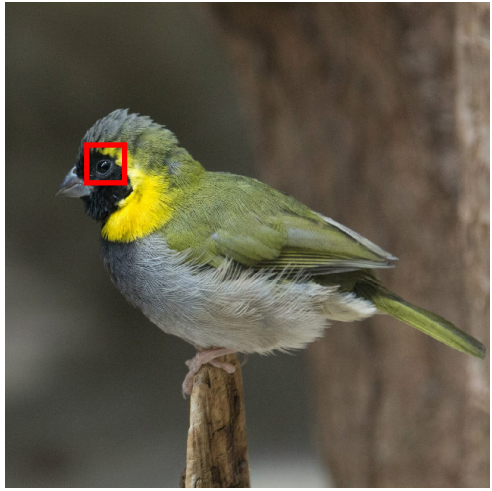
Reading binary values to file

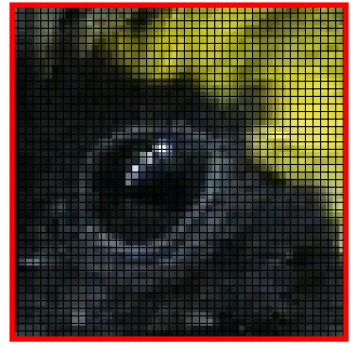
```
// open file for writing
FILE * infile = fopen("data.bin", "r");

// read single integer
int value;
fread(&value, sizeof(int), 1, infile);

// read float array
float array[100];
fread(array, sizeof(float), 100, infile);

// close file
fclose(out);
```

- `fread` is the opposite of `fwrite`. It reads a sequence of bytes from a file
- the first parameter is the pointer to the beginning of memory where data should be read into
- the second parameter is the element size
- the third parameter is the number of elements
- finally the last parameter is the file handle

Images: a real world binary format

Images: a real world binary format



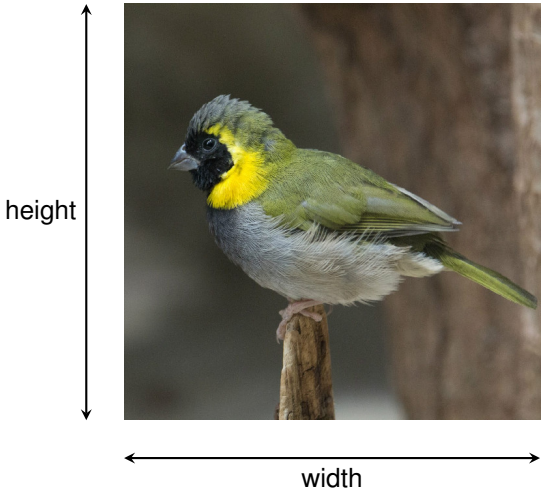


Pixel (24 bit)

- every image on a computer is made out of pixels
- each pixel stores the color information of that point
- there are various color formats, a common one being RGB colors
- each color is made out of a red, green and blue component
- if we store each component as one byte, we can store 256 values per component
- that gives us 256^3 or $\approx$16 million colors

```
struct pixel24 {
    unsigned char b; // blue  (0-255)
    unsigned char g; // green (0-255)
    unsigned char r; // red   (0-255)
};
```

Images: a real world binary format

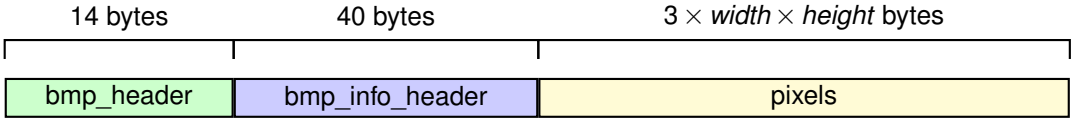


$$\#\text{ of pixels} = \text{width} \times \text{height}$$

Windows Bitmap Format (24bit)

- one very old image format is the Windows Bitmap Format
- first version existed in Windows 1.0
- the current version consists of three parts:

  file header: describing the file
  info header: describing image properties
  pixel data: sequence of pixels stored in row-major order



TEMPLE UNIVERSITY

Bitmap File Header

```
#include <stdint.h>  // for uint16_t, uint32_t

struct bmp_file_header {
    uint16_t bfType;      // always 'BM', which is 0x4D42
    uint32_t bfSize;      // total size of file
    uint32_t bfReserved; // 0
    uint32_t bfOffBits;  // offset to pixel data
} __attribute__((packed)); // no padding between elements
```

TEMPLE UNIVERSITY

Bitmap Info Header

```
struct bmp_info_header {
    uint32_t biSize;           // size of this header
    int32_t  biWidth;          // image width
    int32_t  biHeight;         // image height
    uint16_t biPlanes;         // 1
    uint16_t biBitCount;       // 24
    uint32_t biCompression;    // 0
    uint32_t biSizeImage;      // total size of pixel data
    int32_t  biXPelsPerMeter; // 0
    int32_t  biYPelsPerMeter; // 0
    uint32_t biClrUsed;        // 0
    uint32_t biClrImportant;   // 0
};
```

memset: Initializing blocks of memory

```
int width = 640;
int height = 480;

bmp_file_header bf;
bmp_info_header bi;
pixel24 pixels[height][width];

// initialize memory with 0
memset(&bf, 0, sizeof(bmp_file_header));
memset(&bi, 0, sizeof(bmp_info_header));
memset(&pixels, 0, sizeof(pixels));
```

Bitmap Settings

```
// bitmap file header
bf.bfType = 0x4D42;
bf.bfOffBits = sizeof(bmp_file_header) +
               sizeof(bmp_info_header);
bf.bfSize = bf.bfOffBits + sizeof(pixels);

// bitmap info header
bi.biSize = sizeof(bmp_info_header);
bi.biWidth = width;
bi.biHeight = height;
bi.biPlanes = 1;
bi.biBitCount = 24;
bi.biSizeImage = sizeof(pixels);
```

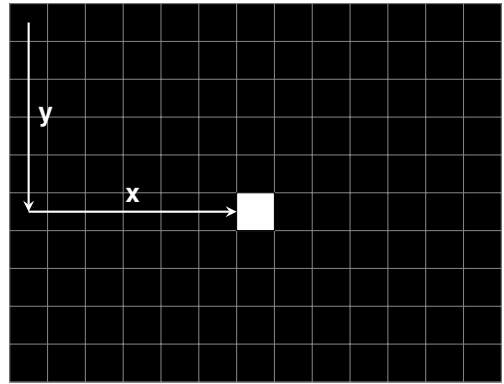
Writing a bitmap file

```
FILE * outfile = fopen("picture.bmp", "w");

fwrite(&bf, sizeof(bmp_file_header), 1, outfile);
fwrite(&bi, sizeof(bmp_info_header), 1, outfile);
fwrite(pixels, sizeof(pixels), 1, outfile);
// (note pixels is an array, so it's already a pointer)

fclose(outfile);
```

Setting color of a single pixel

```
// set pixel white
pixels[y][x].r = 255;
pixels[y][x].g = 255;
pixels[y][x].b = 255;
```

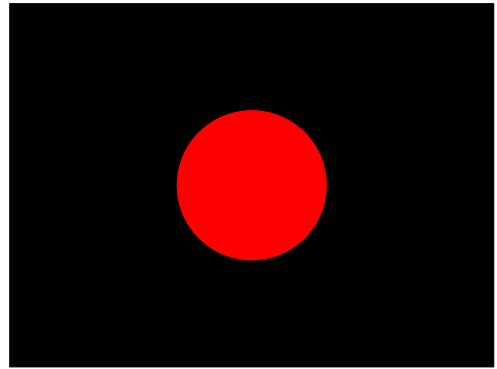
```
// alternative
Pixel & p = pixels[y][x];
p.r = 255;
p.g = 255;
p.b = 255;
```



Drawing a circle

$$(x - x_c)^2 + (y - y_c)^2 < r^2$$

```
int cx = 320;
int cy = 240;
int r  = 100;

for(int j = 0; j < height; ++j) {
    for(int i = 0; i < width; ++i) {
        int x2 = (i - cx)*(i - cx);
        int y2 = (j - cy)*(j - cy);

        if ((x2 + y2) < r*r) {
            pixels[j][i].r = 255;
        }
    }
}
```



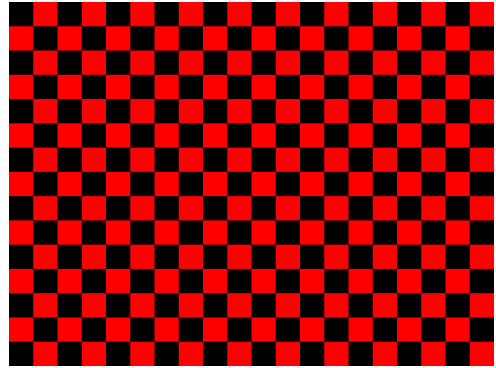
Drawing patterns

```
int cx = 320;
int cy = 240;
int r  = 100;

for(int j = 0; j < height; ++j) {
  for(int i = 0; i < width; ++i) {
    int x_tile = i / 32;
    int y_tile = j / 32;

    if( (x_tile + y_tile) % 2)
    {
      pixels[j][i].r = 255;
    }
  }
}
```

Any guesses what this could be?

Drawing patterns

```
int cx = 320;
int cy = 240;
int r  = 100;

for(int j = 0; j < height; ++j) {
  for(int i = 0; i < width; ++i) {
    int x_tile = i / 32;
    int y_tile = j / 32;

    if( (x_tile + y_tile) % 2)
    {
      pixels[j][i].r = 255;
    }
  }
}
```



Binary formats

- This example shows you that there is no black magic behind binary formats
- All you really have to know is the structure
- Most formats consist of a header, followed by the actual data

- Binary formats are usually more efficient to read
- Use less disk space, because data is directly stored in native machine format
- No loss of precision because of text conversion of floating-point numbers
- However, less portable, because you have to know and use the correct byte count and order