CS100
Introduction to Programming

Lecture 8. More Advanced C & Object-Oriented Programming

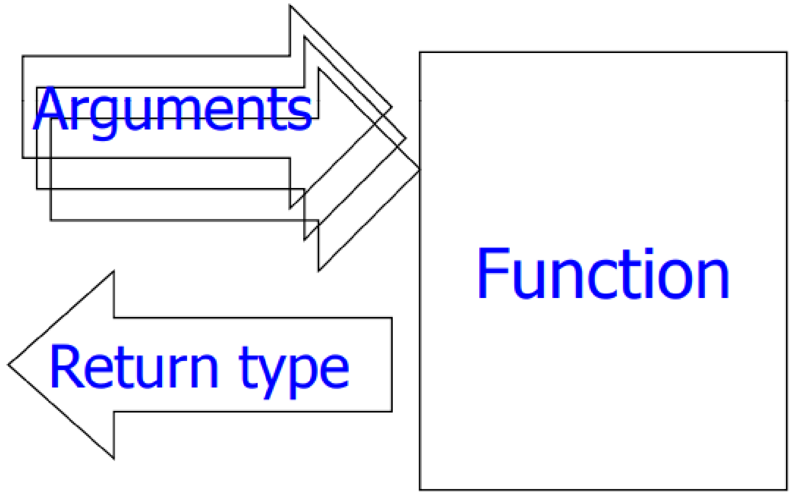
Review on what we have learned

- **Variables**
  - A name given to a continuous range of memory
  - Data type
    - How many memory cells are reserved
    - In what format the data are represented and stored
    - The operations that can be performed on it

char:	1 byte
int:	4 bytes
long:	8 bytes
float:	4 bytes
double:	8 bytes

Review on what we have learned

- **Functions**
  - A function is a self-contained unit of code to carry out a specific task
    - Input argument list
    - Return value



```
float findMaximum
          (float x, float y)
{
   // variable declaration
   float maxnum;

   // find the max number
   if (x >= y)
      maxnum = x;
   else
      maxnum = y;

   return maxnum;
}
```

Review on what we have learned

- **Pointers**
  - Variables which store the addresses of memory locations of some data objects
  - Pointer type
    - How to increment/decrement the pointer address
    - How to retrieve the data value pointed by the pointer

`int *ptrI;`	/* Variable `ptrI` is a pointer. It stores the address of a memory location for an **integer** */
`float *ptrF;`	/* Variable `ptrF` is a pointer. It stores the address of a memory location for a **float** */
`char *ptrC;`	/* Variable `ptrC` is a pointer. It stores the address of a memory location for a **char** */

Review on what we have learned

- **Passing arguments to a function**
  - Call by value
    - The arguments of a function have a local copy of variable value

```
int num1 = 5, num2=10;
int r=add(num1, num2); //int add(int, int);
```

  - Call by pointer
    - The arguments of a function have a local copy of pointer variable value (the address)

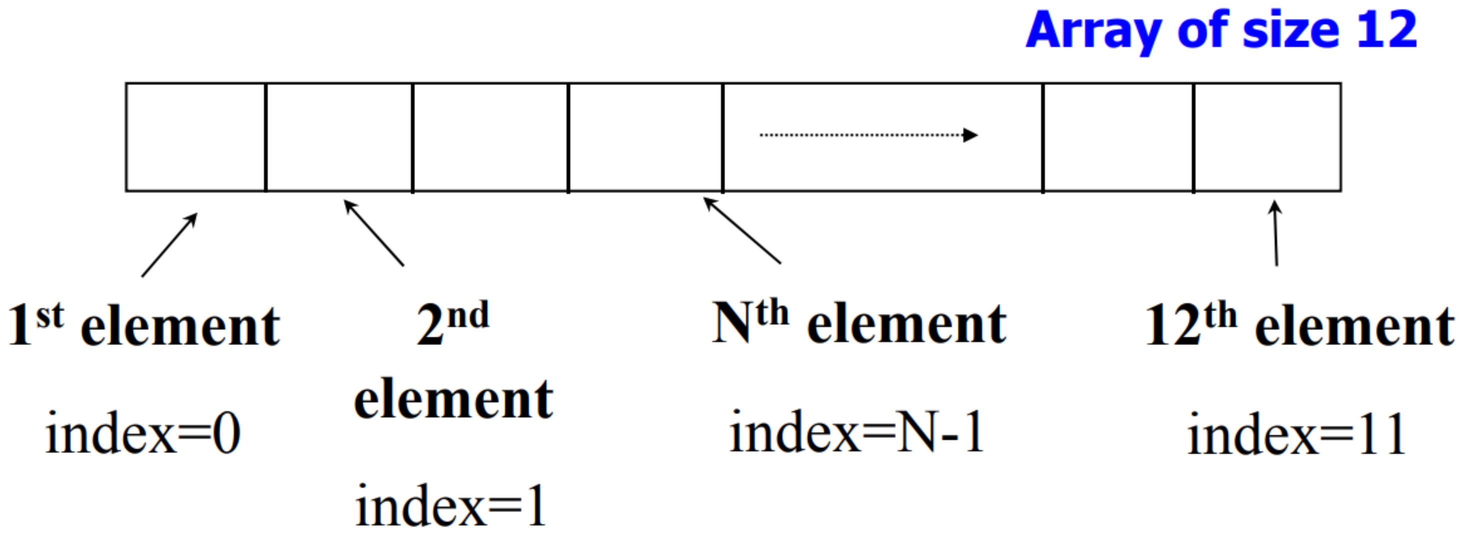
```
int num1 = 5, num2=10;
int r=add(&num1, &num2); //int add(int*, int*);
```

  - Call by reference
    - The arguments of a function do not have a local copy; only another name of the same memory

```
int num1 = 5, num2=10;
int r=add(num1, num2); //int add(int&, int&);
```

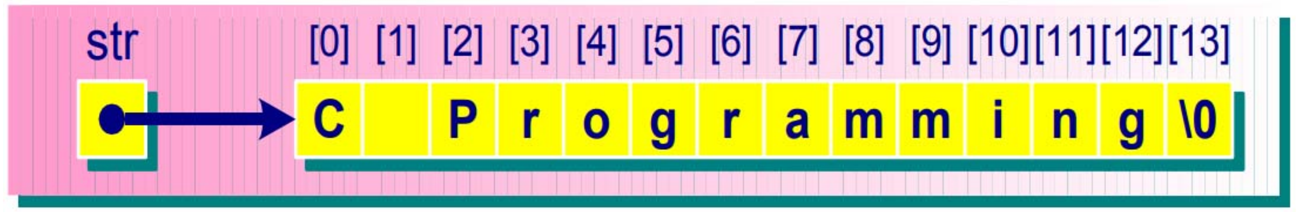
Review on what we have learned

- **Array**
  - A continuous range of data values in memory
  - Zero-based index in C
  - Static v.s. dynamic array



Review on what we have learned

- **Strings**
  - – An array of characters ended by '\0'
  - – Static v.s. dynamic strings



  - – String operations
    - length/concatenation/comparison…

Review on what we have learned

- **Memory copy**
  - You can use loops to copy one by one
    - Not recommended unless necessary (slow)
  - Use memory copy functions
    - memcpy(…)/strcpy(…)

```
float* data=(float*)malloc(sizeof(float)*100);
…
float* data_new[100];
memcpy(data_new, data, sizeof(float)*100);
```

Review on what we have learned

- **Structures**
  - An aggregate of values
  - Can contain members with different types

```
typdef struct
{
   int shipClass;
   char *name;
   int speed, crew;
} warShip;
```

  - <u>Structure can allow the lowest level of data abstraction</u>

Practical Example 1: Vector

- **Recall what is a vector in linear algebra?**
  - A 1D arrangement (array) of variables/numbers

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

  - There are several mathematical operations on vectors

Practical Example 1: Vector

- **Vector operations**
  - Element-wise addition / subtraction / multiplication / division

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mathbf{y} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1+3 \\ 2+4 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$$

  - Scaling

$$2\mathbf{x} = 2\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2*1 \\ 2*2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

Practical Example 1: Vector

- **Vector operations**
  - Norm

$$||\mathbf{x}|| = \sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{x_1^2 + x_2^2 + x_3^2 + \ldots + x_n^2}$$

  - Dot product

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \ldots\ldots + x_n y_n$$

Practical Example 1: Vector

- **How to define a vector**

```
struct VECTOR
{
    int dim;
    float* data;
};
```

Used for vector presentation

```
struct VECTOR_FILE_HEADER
{
    int dim;
    int data_element_size; //size in byte
};
```

Used for writing/reading vector data onto hard disk

Practical Example 1: Vector

- **What functions are needed for a vector?**

```
bool create_vector(int, VECTOR*);
void destroy_vector(VECTOR*);

bool vector_assign(VECTOR*, const VECTOR*);

int get_vector_dim(const VECTOR*);

float& get_vector_element(int, const VECTOR*);
void set_vector_element(int, float, VECTOR*);

float* get_vector_data(const VECTOR*);
```

Practical Example 1: Vector

- **What functions are needed for a vector?**

```
bool vector_add(VECTOR*, const VECTOR*);
bool vector_sub(VECTOR*, const VECTOR*);
bool vector_mul(VECTOR*, const VECTOR*);
bool vector_div(VECTOR*, const VECTOR*);
float vector_dot(const VECTOR*, const VECTOR*);

float get_vector_norm(const VECTOR*);

void print_vector(const VECTOR*);

bool write_vector(const char*, const VECTOR*);
bool read_vector(const char*, const VECTOR*);
```

Practical Example 1: Vector

- **Create a vector dynamically**

```
bool create_vector(int dim, VECTOR* p_vec_out)
{
    if (dim <= 0)
    {
        printf("invalid vector dimension!\n");
        return false;
    }

    if (p_vec_out != NULL)
    {
        p_vec_out->data = (float*)malloc(sizeof(float) * dim);
        if (p_vec_out->data != NULL)
        {
            memset(p_vec_out->data, 0, sizeof(float) * dim);
            p_vec_out->dim = dim;

             return true;
        }
    else
    ...
}
```

Practical Example 1: Vector

- **Create a vector dynamically**

```
bool create_vector(int dim, VECTOR* p_vec_out)
{
    ...

    else
    {
        printf("unable to allocate vector data!\n");
        p_vec_out->dim = 0;

        return false;
        }
    }
    else
        return false;
}
```

Practical Example 1: Vector

- **Destroy a vector**

```c
void destroy_vector(VECTOR* p_vec)
{
    if (p_vec != NULL)
    {
        if (p_vec->data != NULL)
        free(p_vec->data);
        p_vec->dim = 0;
    }
}
```

Practical Example 1: Vector

- **Assign one vector to another**

```
bool vector_assign(VECTOR* p_vec1, const VECTOR* p_vec2)
{
     if (p_vec1 != NULL && p_vec2 != NULL)
     {
          if (p_vec1->dim != p_vec2->dim)
          {
               if (p_vec1->data != NULL)
                    free(p_vec1->data);
               p_vec1->data = (float*)malloc(sizeof(float) * p_vec2->dim);
               if (p_vec1->data == NULL)
                    return false;
               memcpy(p_vec1->data, p_vec2->data, sizeof(float) * p_vec2->dim);
               p_vec1->dim = p_vec2->dim;
          }
          else
          {
               memcpy(p_vec1->data, p_vec2->data, sizeof(float) * p_vec2->dim);
          }
          return true;
     }
     else
          return false;
}
```

Practical Example 1: Vector

- **Get relevant data in a vector**

```
int get_vector_dim(const VECTOR* p_vec)
{
        return p_vec->dim;
}

float& get_vector_element(int i, const VECTOR* p_vec)
{
        return p_vec->data[i];
}

void set_vector_element(int i, float data_value, VECTOR* p_vec)
{
        p_vec->data[i] = data_value;
}

float* get_vector_data(const VECTOR* p_vec)
{
        return p_vec->data;
}
```

Practical Example 1: Vector

- **Element-wise addition / subtraction / multiplication / division**

```
bool vector_add(VECTOR* p_vec1, const VECTOR* p_vec2)
{
    if (p_vec1->dim != p_vec2->dim)
        return false;
    if (p_vec1->data == NULL || p_vec2->data == NULL)
        return false;

    for (int i = 0; i < p_vec1->dim; i++)
        p_vec1->data[i] += p_vec2->data[i];

    return true;
}
```

Practical Example 1: Vector

- **Dot product between two vectors**

```
float vector_dot(const VECTOR* p_vec1, const VECTOR* p_vec2)
{
    if (p_vec1->dim != p_vec2->dim)
        return false;
    if (p_vec1->data == NULL || p_vec2->data == NULL)
        return false;

    float dot_ret = 0;
    for (int i = 0; i < p_vec1->dim; i++)
        dot_ret += p_vec1->data[i] * p_vec2->data[i];

    return dot_ret;
}
```

Practical Example 1: Vector

- **Print the vector onto the screen**

```
void print_vector(const VECTOR* p_vec)
{
    for (int i = 0; i < get_vector_dim(p_vec) - 1; i++)
        printf("%f, ", get_vector_element(i, p_vec));
    printf("%f\n", get_vector_element
                    (get_vector_dim(p_vec) - 1, p_vec));
}
```

Practical Example 1: Vector

- **Write a vector to a file**

```
bool write_vector(const char* path, const VECTOR* p_vec)
{
    FILE* p_file = fopen(path, "wb");
    if (p_file == NULL)
        return false;

    VECTOR_FILE_HEADER header;
    header.dim = p_vec->dim;
    header.data_element_size = sizeof(float);

    if (fwrite(&header, sizeof(VECTOR_FILE_HEADER), 1, p_file) != 1)
    {
        printf("writing vector header error!\n");
        return false;
    }
     ...
}
```

Practical Example 1: Vector

- **Write a vector to a file**

```
bool write_vector(const char* path, const VECTOR* p_vec)
{
    ...

    if (fwrite(get_vector_data(p_vec),
                header.data_element_size, header.dim, p_file) != header.dim)
    {
        printf("writing vector data error!\n");
        return false;
    }

    fclose(p_file);

    return true;
}
```

Practical Example 1: Vector

- **Read a vector from a file**

```
bool read_vector(const char* path, const VECTOR* p_vec)
{
     FILE* p_file = fopen(path, "rb");
     if (p_file == NULL)
          return false;

     VECTOR_FILE_HEADER header;
     memset(&header, 0, sizeof(VECTOR_FILE_HEADER));
     if (fread(&header,sizeof(VECTOR_FILE_HEADER), 1, p_file) != 1)
     {
          printf("reading vector header error!\n");
          return false;
     }

     if (fread(get_vector_data(p_vec), header.data_element_size, header.dim, p_file) != header.dim)
     {
          printf("reading vector data error!\n");
          return false;
     }

     fclose(p_file);

     return true;
}
```

Practical Example 1: Vector

- **Use the vector functions for computation**

```
VECTOR v1;
create_vector(5, &v1);
for (int i = 0; i < get_vector_dim(&v1); i++)
         set_vector_element(i,float(rand()) / RAND_MAX,&v1);

printf("vector v1 is:\n");
print_vector(&v1);
printf("the norm of v1 is: %f", get_vector_norm(&v1));

printf("\n\n");

VECTOR v2;
create_vector(5, &v2);
vector_assign(&v2, &v1);
vector_mul(&v2, &v1);
printf("vector v2 is:\n");
print_vector(&v2);

printf("the norm of v2 is: %f\n\n", get_vector_norm(&v2));
...
```

Practical Example 1: Vector

- **Use the vector functions for computation**

```
...

printf("v1 dot v2 is: %f\n", vector_dot(&v1, &v2));

printf("saving vectors...\n");
write_vector("D:\\v1.dat", &v1);
write_vector("D:\\v2.dat", &v2);

destroy_vector(&v1);
destroy_vector(&v2);
```

Practical Example 2: Matrix

- **Recall what is a matrix in linear algebra?**
  - A two dimensional arrangement (array) of variables/numbers

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots\cdots & a_{1n} \\ a_{21} & a_{22} & \cdots\cdots & a_{2n} \\ \vdots & & & \\ a_{m1} & a_{m2} & \cdots\cdots & a_{mn} \end{bmatrix}$$

  - There are several mathematical operations on matrices

Practical Example 2: Matrix

- **Matrix operations**
  - Element-wise addition / subtraction / multiplication / division
  - Scaling
  - Matrix-matrix product

$$c_{ij} = \sum_k a_{ik} * b_{kj} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + \ldots + a_{ik} * b_{kj}$$

Practical Example 2: Matrix

- **Matrix operations**
  - Matrix-matrix product example

$$C = A * B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} * \begin{bmatrix} 3 & 2 & 1 & 2 \\ 2 & 1 & 4 & 5 \end{bmatrix}$$

$$= \begin{bmatrix} 1*3+2*2 & 1*2+2*1 & 1*1+2*4 & 1*2+2*5 \\ 3*3+4*2 & 3*2+4*1 & 3*1+4*4 & 3*2+4*5 \\ 5*3+6*2 & 5*2+6*1 & 5*1+6*4 & 5*2+6*5 \end{bmatrix}$$

$$= \begin{bmatrix} 7 & 4 & 9 & 12 \\ 17 & 10 & 19 & 26 \\ 27 & 16 & 29 & 40 \end{bmatrix}$$

  - Matrix-vector product

Practical Example 2: Matrix

- **Matrix operations**
  - Transpose

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad X^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

  - Norm
  - Inverse
  - Eigenvector/eigenvalues

    …

Practical Example 2: Matrix

- **How to define a matrix**

```
struct MATRIX
{
    int row_dim;
    int col_dim;
    float* data;
};
```

Used for matrix presentation

```
struct MATRIX_FILE_HEADER
{
    int row_dim;
    int col_dim;
    int data_element_size; //size in byte
};
```

Used for writing/reading matrix data onto hard disk

Practical Example 2: Matrix

- **What functions are needed for a matrix?**

```
bool create_matrix(int, int, MATRIX*);
void destroy_matrix(MATRIX*);

bool matrix_assign(MATRIX*, const MATRIX*);

int get_matrix_row_dim(const MATRIX*);
int get_matrix_col_dim(const MATRIX*);

float& get_matrix_element(int, int, const MATRIX*);
void set_matrix_element(int, int, float, MATRIX*);

float* get_matrix_data(const MATRIX*);
```

Practical Example 2: Matrix

- **What functions are needed for a matrix?**

```
bool matrix_add(MATRIX*, const MATRIX*);
bool matrix_sub(MATRIX*, const MATRIX*);
MATRIX * matrix_mul(const MATRIX*, const MATRIX*);
bool matrix_element_mul(MATRIX*, const MATRIX*);
bool matrix_element_div(MATRIX*, const MATRIX*);

VECTOR* matrix_vector_mul(const MATRIX*, const VECTOR*);

float get_matrix_norm(const MATRIX*);

void print_matrix(const MATRIX*);

bool write_matrix(const char*, const MATRIX*);
bool read_matrix(const char*, const MATRIX*);
```

Practical Example 2: Matrix

- **Create a matrix dynamically**

```
bool create_matrix(int row_dim, int col_dim, MATRIX* p_mat_out)
{
    if (row_dim <= 0 || col_dim <= 0)
    {
        printf("invalid matrix dimension!\n");
        return false;
    }

    if (p_mat_out != NULL)
    {
        p_mat_out->data = (float*)malloc(sizeof(float) * row_dim * col_dim);
        if (p_mat_out->data != NULL)
        {
            memset(p_mat_out->data, 0, sizeof(float) * row_dim * col_dim);
            p_mat_out->row_dim = row_dim;
            p_mat_out->col_dim = col_dim;

            return true;
        }
        else
    ...
}
```

Practical Example 2: Matrix

- **Create a matrix dynamically**

```
bool create_matrix(int row_dim, int col_dim, MATRIX* p_mat_out)
{
    ...

        else
        {
            printf("unable to allocate matrix data!\n");
            p_mat_out->row_dim = 0;
            p_mat_out->col_dim = 0;

            return false;
        }
    }
    else
        return false;
}
```

Practical Example 2: Matrix

- **Destroy a matrix**

```
void destroy_matrix(MATRIX* p_mat)
{
    if (p_mat != NULL)
    {
        if (p_mat->data != NULL)
                free(p_mat->data);
        p_mat->row_dim = 0;
        p_mat->col_dim = 0;
    }
}
```

Practical Example 2: Matrix

- **Assign one matrix to another**

```
bool matrix_assign(MATRIX* p_mat1, const MATRIX* p_mat2)
{
    if (p_mat1 != NULL && p_mat2 != NULL)
    {
        if (p_mat1->row_dim != p_mat2->row_dim
                || p_mat1->col_dim != p_mat2->col_dim)
        {
            if (p_mat1->data != NULL)
                free(p_mat1->data);
            p_mat1->data = (float*)malloc(
                sizeof(float) * p_mat2->row_dim * p_mat2->col_dim);
            if (p_mat1->data == NULL)
                return false;
    ...
}
```

Practical Example 2: Matrix

- **Assign one matrix to another**

```
bool matrix_assign(MATRIX* p_mat1, const MATRIX* p_mat2)
{
    ...
        memcpy(p_mat1->data, p_mat2->data,
                sizeof(float) * p_mat2->row_dim * p_mat2->col_dim);
        p_mat1->row_dim = p_mat2->row_dim;
        p_mat1->col_dim = p_mat2->col_dim;
        }
        else
        {
            memcpy(p_mat1->data, p_mat2->data,
                sizeof(float) * p_mat2->row_dim * p_mat2->col_dim);
        }
        return true;
    }
    else
        return false;
}
```

Practical Example 2: Matrix

- **Ge t relevant matrix data**

```
int get_matrix_row_dim(const MATRIX* p_vec)
{
        return p_vec->row_dim;
}

int get_matrix_col_dim(const MATRIX* p_vec)
{
        return p_vec->row_dim;
}

float& get_matrix_element(int i, int j, const MATRIX* p_mat)
{
        return p_mat->data[i* p_mat->row_dim + j];
}
```

Practical Example 2: Matrix

- **Ge t relevant matrix data**

```
void set_matrix_element(int i, int j, float data_value, MATRIX* p_mat)
{
        p_mat->data[i * p_mat->row_dim + j] = data_value;
}

float* get_matrix_data(const MATRIX* p_mat)
{
        return p_mat->data;
}
```

Practical Example 2: Matrix

- **Element-wise addition / subtraction / multiplication / division**

```
MATRIX * matrix_mul(const MATRIX* p_mat1, const MATRIX* p_mat2)
{
    if (p_mat1->row_dim <= 0 || p_mat2->col_dim)
        return NULL;
    if (p_mat1->col_dim != p_mat2->row_dim)
        return NULL;
    if (p_mat1->data == NULL || p_mat2->data == NULL)
        return NULL;

    MATRIX* mat_ret = NULL;
    if (!create_matrix(p_mat1->row_dim, p_mat2->col_dim, mat_ret))
        return NULL;
    ...
}
```

Practical Example 2: Matrix

- **Element-wise addition / subtraction / multiplication / division**

```
MATRIX * matrix_mul(const MATRIX* p_mat1, const MATRIX* p_mat2)
{
    ...
    for (int i = 0; i < p_mat1->row_dim; i++){
        for (int j = 0; j < p_mat2->col_dim; j++){
            float element_value = 0;
            for (int k = 0; k < p_mat1->col_dim; k++){
                element_value += get_matrix_element(i, k, p_mat1)
                * get_matrix_element(k, j, p_mat2);
            }
                set_matrix_element(i, j, element_value, mat_ret);
            }
        }

    return mat_ret;
}
```

Practical Example 2: Matrix

- **Matrix-matrix multiplication**

```
MATRIX * matrix_mul(const MATRIX* p_mat1, const MATRIX* p_mat2)
{
     if (p_mat1->row_dim <= 0 || p_mat2->col_dim)
          return NULL;
     if (p_mat1->col_dim != p_mat2->row_dim)
          return NULL;
     if (p_mat1->data == NULL || p_mat2->data == NULL)
          return NULL;

     MATRIX* mat_ret = NULL;
     if (!create_matrix(p_mat1->row_dim, p_mat2->col_dim, mat_ret))
          return NULL;

      ...
}
```

Practical Example 2: Matrix

- **Matrix-matrix multiplication**

```
MATRIX * matrix_mul(const MATRIX* p_mat1, const MATRIX* p_mat2)
{
    ...
    for (int i = 0; i < p_mat1->row_dim; i++)
    {
        for (int j = 0; j < p_mat2->col_dim; j++)
        {
            float element_value = 0;
            for (int k = 0; k < p_mat1->col_dim; k++)
            {
                element_value += get_matrix_element(i, k, p_mat1) *
                get_matrix_element(k, j, p_mat2);
            }
            set_matrix_element(i, j, element_value, mat_ret);
        }
    }

    return mat_ret;
}
```

Practical Example 2: Matrix

- **Matrix-vector multiplication**

```
VECTOR* matrix_vector_mul(const MATRIX* p_mat, const VECTOR* p_vec)
{
    VECTOR* p_vec_ret = NULL;
    if(!create_vector(get_vector_dim(p_vec), p_vec_ret))
        return NULL;

    for (int i = 0; i < get_vector_dim(p_vec_ret); i++)
    {
        float vec_element_value = 0;
        for (int j = 0; j < get_matrix_col_dim(p_mat); j++)
        {
            vec_element_value += get_matrix_element(i, j, p_mat) *
                                        get_vector_element(j, p_vec);
        }
        set_vector_element(i, vec_element_value, p_vec_ret);
    }

    return p_vec_ret;
}
```

Practical Example 2: Matrix

- **Write a matrix to a file**

```
bool write_matrix(const char* path, const MATRIX* p_mat)
{
    FILE* p_file = fopen(path, "wb");
    if (p_file == NULL)
    return false;

    MATRIX_FILE_HEADER header;
    header.row_dim = p_mat->row_dim;
    header.col_dim = p_mat->col_dim;
    header.data_element_size = sizeof(float);

    if (fwrite(&header, sizeof(MATRIX_FILE_HEADER), 1, p_file) != 1)
    {
        printf("writing matrix header error!\n");
        return false;
    }
    ...
}
```

Practical Example 2: Matrix

- **Write a matrix to a file**

```
bool write_matrix(const char* path, const MATRIX* p_mat)
{
    ...

    int total_count = header.row_dim * header.col_dim;

    if (fwrite(get_matrix_data(p_mat), header.data_element_size,
                    total_count, p_file) != total_count)
    {
        printf("writing matrix data error!\n");
        return false;
    }

    fclose(p_file);

    return true;
}
```

Practical Example 2: Matrix

- **Reading a matrix from a file**

```
bool read_matrix(const char* path, const MATRIX* p_mat)
{
    FILE* p_file = fopen(path, "rb");
    if (p_file == NULL)
    return false;

    MATRIX_FILE_HEADER header;
    memset(&header, 0, sizeof(MATRIX_FILE_HEADER));
    if (fread(&header, sizeof(MATRIX_FILE_HEADER), 1, p_file) != 1)
    {
        printf("reading MATRIX header error!\n");
        return false;
    }

    return true;
}
```

Practical Example 2: Matrix

- **Reading a matrix from a file**

```
bool read_matrix(const char* path, const MATRIX* p_mat)
{
    ...

    int total_count = header.row_dim * header.col_dim;

    if (fread(get_matrix_data(p_mat), header.data_element_size,
                        total_count, p_file) != total_count)
    {
        printf("reading matrix data error!\n");
        return false;
    }

    fclose(p_file);

    return true;
}
```

Practical Example 2: Matrix

- **Use the matrix functions for computation**

```
MATRIX m1;
init_matrix(&m1);
create_matrix(3, 3, &m1);
for (int r = 0; r < get_matrix_row_dim(&m1); r++)
{
     for (int c = 0; c < get_matrix_col_dim(&m1); c++)
     {
          set_matrix_element(r, c, float(rand()) / RAND_MAX, &m1);
     }
}
printf("matrix m1 is:\n");
print_matrix(&m1);
printf("\n");
```

Practical Example 2: Matrix

- **Use the matrix functions for computation**

```
MATRIX m2;
init_matrix(&m2);
matrix_assign(&m2, &m1);
matrix_mul(&m2, &m1);
printf("matrix m2 is:\n");
print_matrix(&m2);

printf("\n");
printf("the norm of m1 is: %f\n", get_matrix_norm(&m1));
printf("the norm of m2 is: %f\n", get_matrix_norm(&m2));

printf("\n");
printf("saving matrices...\n");
write_matrix("D:\\m1.dat", &m1);
write_matrix("D:\\m2.dat", &m2);

destroy_matrix(&m1);
destroy_matrix(&m2);
```

Can we do things better?

- **Procedural programming in C**
  - The programs are organized in terms of functions
  - Function design plays a central role

- **Problem for procedural programming**
  - The logic is not consistent to human thinking
  - Data abstraction is relatively poor
  - Code sharing is difficult

Encapsulation

- **Encapsulation in C**
  - Look at structure again

```
struct Person
{
	char* name;
	int age;
	float height;
	float weight;
};
```

  - What we lack for structure in C
    - The related operations in a structure variable (object)

```
struct Person
{
	char* name;
	int age;
	float height;
	float weight;

	void increase_age_by(int age_increment = 1);
};
```

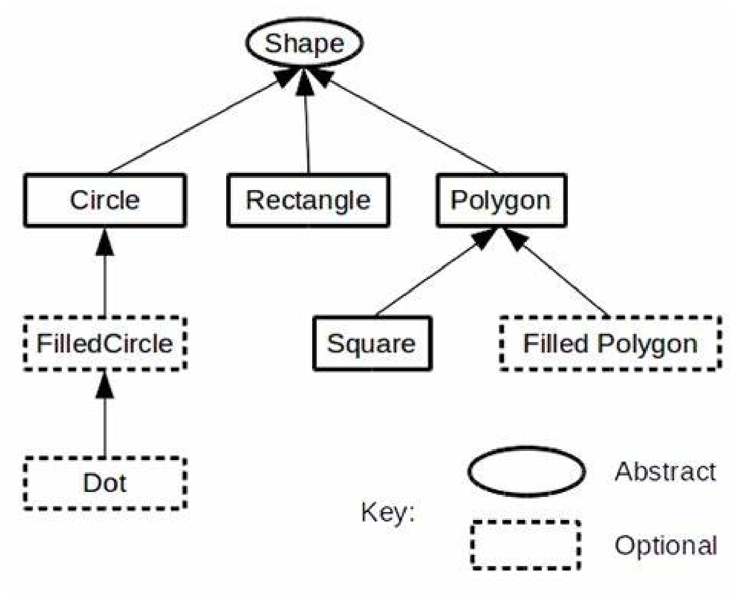
Encapsulation

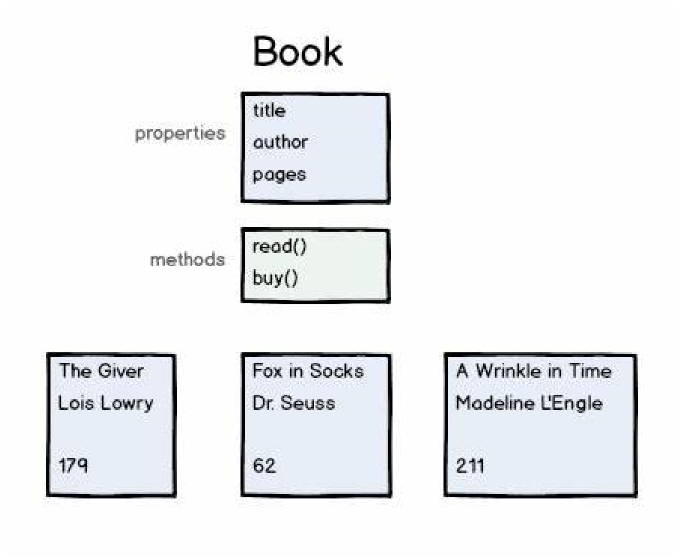
- **The concept of object**
  - *Object means a real-world entity* such as a pen, chair, table, computer, watch, etc.

  - *How to define an object in a program?*
    - Data: define the necessary properties of an object
    - Functions: the related operations of data for an object
    - Accessibility: The ability to make the data values within an object inaccessible
      - To maintain integrity of an object
      - To protect an object from being falsely operated
      - To prevent the influence from other data or objects

Object-Oriented Programming

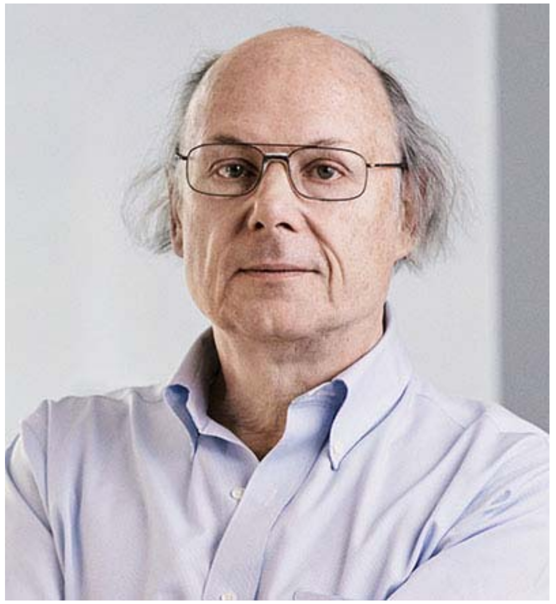
- **A programming language model**
  - programs are organized around objects
  - identify all of the objects to manipulate and how they relate to each other





C++ Language

- **An object-oriented programming language**
  - A significant extension of C
  - Code C++ in a "C" with "object-oriented" style.
  - Invented by Bjarne Stroustrup
    - Classes and objects
    - Function overloading
    - Inheritance
    - Polymorphism
    - Templates



Class

- **From structure to class**
  - Augment with (member) functions
  - Add data accessibility (public, private, etc.)
  - Add automatic construction/destruction functions for data initialization/deletion

```
struct vertex
{
    float x, y, z;
};
```

```
class vertex{
    public:
    float x, y, z;

    vertex& operator = (const vertex&);
    float get_dist();//get distance to origin

    vertex();
    ~vertex();
};
```

Member variables

Member functions

Constructor/destructor

Class Object

- **A class can be used to declare an object**
  - Like a structure variable declaration in C
  - For example:

```
vertex v1;
v1.x = 10.2f;
v1.y = 5.8f;
v1.z = -6.3f;

vertex v2;
v2 = v1;

printf("The distance to origin is: %f", v2.get_dist());
```

Accessibility in a Class

- **Public members**
  - Declared with the “public” specifier
  - Accessed without any restriction

- Private members
  - “private”
  - Accessed within class

```
class vertex{
    public:
    float x, y, z;

    vertex& operator = (const vertex&);
    float get_dist();

    vertex();
    ~vertex();
};
```

Converting Previous “Vector” to a Class

- **For example**: declaring a vector class

```
class vector
{
public:
    vector& normalize();
    float get_norm(float p = 2.0f);

    vector& add(const vector&);
    vector& sub(const vector&);
    vector& mul(const vector&);
    vector& div(const vector&);
    float dot(const vector&);

    float& operator [] (long);
    const float& operator [] (long)const;
    ...
};
```

Converting Previous “Vector” to a Class

- **For example**: declaring a vector class

```
class vector
{
public:
    ...
    int get_dim();
    float* get_data();

    vector& resize(int dim);

    void print_to_console();

    vector();
    vector(int dim);
    vector(const vector&);
    ~vector();
private:
    float* m_data;
    int m_dim;
};
```

Implementing Member Functions

- **Scope operator**
  - The operator "::"
  - Specify which scope a member belongs to

- **For example**
  - Implement get_dist() in vertex class

```
float vertex::get_dist()
{
        return (float)sqrt(x * x + y * y + z * z);
}
```

Implementing Member Functions

- **Implementing more complex member function**
  - The resize() function in vector class

```
vector& vector::resize(int dim)
{
    if (m_dim != dim)
    {
        if (m_data != NULL)
            delete []m_data;
        m_data = new float[dim];
        if (m_data != NULL)
        {
            memset(m_data, 0, sizeof(float) * dim);
            m_dim = dim;
        }
    }

    return (*this);
}
```

Initialization of Member Variables

- **What is the role of "constructor"**
  - Called when an object is created
  - Responsible for initialization tasks

- **Simple example**
  - Initializing vertex member variables

```
vertex::vertex()
{
    x = y = z = 0.0f;
}
```

```
class vertex
{
    public:
        float x, y, z;
        ...
        vertex();
        ~vertex();
};
```

Initialization of Member Variables

- **More example**
  - Initializing a vector with dynamic memory allocation
  - Private variables only used within the class scope

```
vector::vector(int dim)
{
    m_data = new float[dim];
    m_dim = dim;
}
```

```
class vector
{
public:
    ...
    vector();
    vector(int dim);
    vector(const vector&);
    ~vector();
private:
    float* m_data;
    int m_dim;
};
```

Copy Constructor

- **If we want to initialize based on other object?**
  - Copy constructor allows you to do this
  - Can access private data members from input object
  - **Simple example**

```
vertex::vertex(const vertex& v)
{
    x = v.x;
    y = v.y;
    z = v.z;
}
```

```
class vertex
{
public:
    float x, y, z;
    ...
    vertex();
    vertex(const vertex&);
    ~vertex();
};
```

Copy Constructor

- **If we want to initialize based on other object?**
  - More complex example

```
vector::vector(const vector& v)
{
    if (m_dim != v.m_dim)
        delete []m_data;
    m_data = new float[v.m_dim];
    if (m_data != NULL)
    {
        memcpy(m_data, v.m_data,
              sizeof(float) * v.m_dim);
        m_dim = v.m_dim;
    }
    else
        m_dim = 0;
}
```

```
class vector
{
public:
    ...
    vector();
    vector(int dim);
    vector(const vector&);
    ~vector();
private:
    float* m_data;
    int m_dim;
};
```

Destructor

- When object is deleted in memory
  - Go out of its scope
    - E.g., when function call is finished
  - Actively freed

```
vector::~vector()
{
    if (m_data != NULL)
        delete []m_data;
}
```

```
class vector
{
public:
    ...
    vector();
    vector(int dim);
    vector(const vector&);
    ~vector();
private:
    float* m_data;
    int m_dim;
};
```

“this” Pointer within a Class

- **A built-in pointer within the class scope**
  - Point to the starting address of a class object
  - Can be used to access member variables and functions
  - **Simple example**

```
vertex& vertex::add(const vertex& v)
{
    x += v.x;
    y += v.y;
    z += v.z;

    return (*this);
}
```

```
class vertex
{
public:
    float x, y, z;

    vertex& add(const vertex&);

    vertex();
    vertex(const vertex&);
    ~vertex();
};
```

“this” Pointer within a Class

- **A built-in pointer within the class scope**
  - Point to the starting address of a class object
  - Can be used to access member variables and functions
  - **Simple example**

```
vertex& vertex::add
        (float x, float y, float z)
{
    this->x = x;
    this->y = y;
    this->z = z;

    return (*this);
}
```

```
class vertex
{
public:
    float x, y, z;

    vertex& add(float, float, float);

    vertex();
    vertex(const vertex&);
    ~vertex();
};
```

Size of a Class Object

- **What does “sizeof” operator return for object?**
  - Similar as structure
  - The total size for all the data members
    - Both public and private members

```
vertex v;
sizeof(v)=sizoef(v.x) + sizoef(v.y) + sizoef(v.z)

vector v;
sizeof(v)=sizoef(float*) + sizoef(int)
```

Dynamic Construction of Class Object

- **Static construction**

```
vector v;
vector* p_v = &v;
v.resize(10);
for (int i = 0; i < p_v->get_dim(); i++)
    (*p_v)[i] = float(i);
```

- **Dynamic construction**
  - Whether we can still use malloc()/free()?
    - Not recommended -> usually do not produce right result
  - Why? – Constructor/destructor not called

Dynamic Construction of Class Object

- **Dynamic construction**
  - The new/delete operators in C++
  - Constructor and destructor will always be called
  - Dynamic construction/deletion of a single object:

```
vector* p_v = new vector(10);

for (int i = 0; i < p_v->get_dim(); i++)
    (*p_v)[i] = float(i);

delete p_v;
```

Dynamic Construction of Class Object

- **Dynamic construction of class object**
  - Dynamic construction/deletion of an array of objects:

```
int vec_num = 0;
printf("Please input the vector number:");
scanf("%d", &vec_num);

vector* p_va = new vector[vec_num];
for (int i = 0; i < vec_num; i++)
{
    p_va[i].resize(10);
    for (int j = 0; j < p_va[i].get_dim(); j++)
        p_va[i][j] = float(i+j);
}

delete []p_va;
```

The Structure in C++

- **C++ still preserves structure type, but**
  - Augmented with accessibility
  - Enable both member variables and functions
- **Difference from class**
  - By default, without scope specifiers
    - Class -> all private variables
    - Structure -> all public variables

```
struct Person
{
	char* name;
	int age;
	float height;
	float weight;

	Person& operator
	           = (const Person&);

	Person();
	Person(const Person&);
};
```