CS100
Introduction to Programming

Lecture 12. C summary & 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

- **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

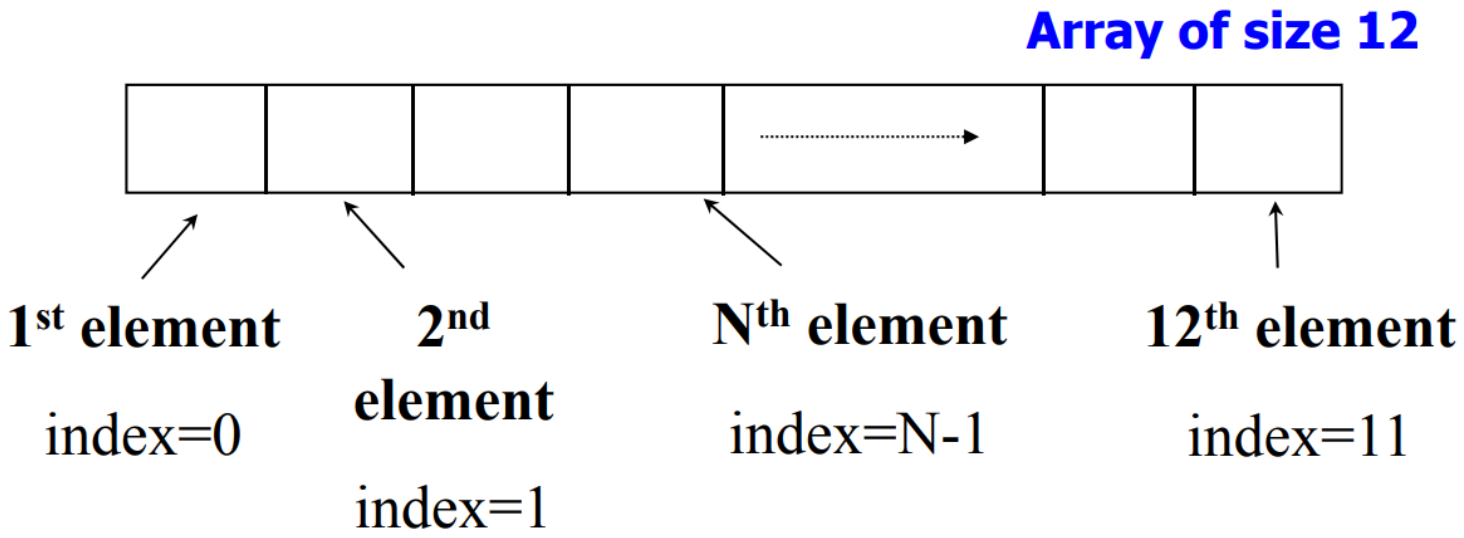
- **Structures**
  - An aggregate of values
  - Can contain members with different types

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

  - Structure can allow the lowest level of data abstraction

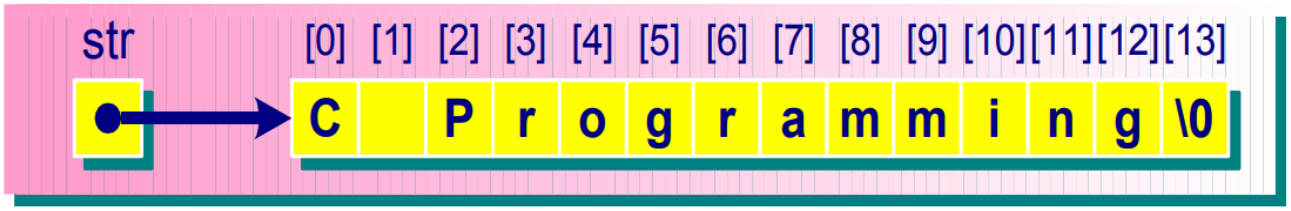
Review on what we have learned

- **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

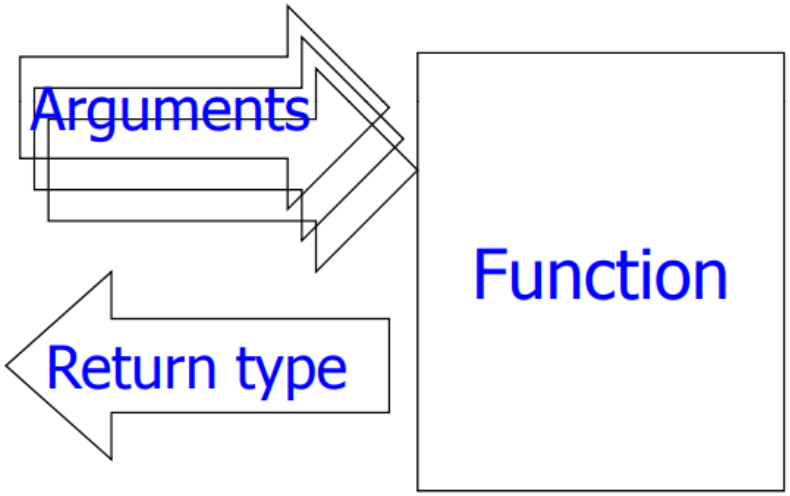
- **Strings**
  - An array of characters ended by '\0'
  - Static v.s. dynamic strings



  - String operations
    - length/concatenation/comparison…

Review on what we have learned

- **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

- **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

- **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);
```

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**

```
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**

```
...

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);
```

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);
};
```

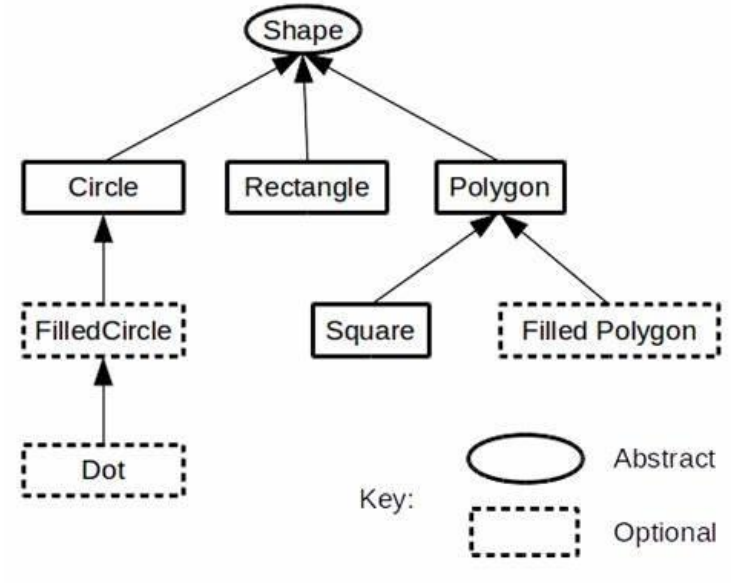
Object-Oriented Programming

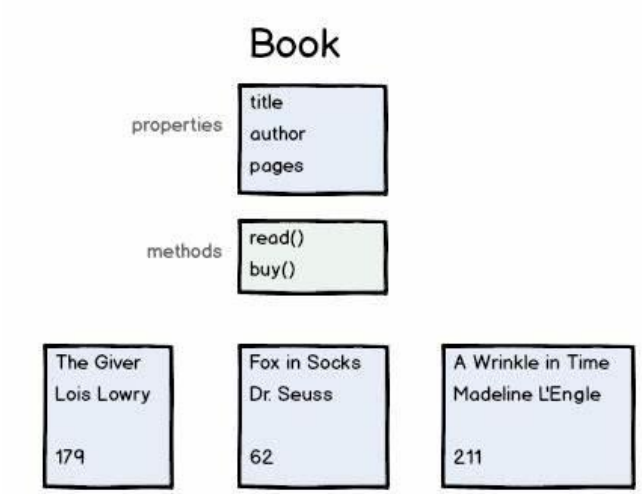
- now that we start using C++, we can start taking advantage of ***object-oriented programming***

- adding OOP to C was one of the driving forces behind the creation of C++ as a language
  - C++'s predecessor was actually called "C with Classes"

Object-Oriented Programming

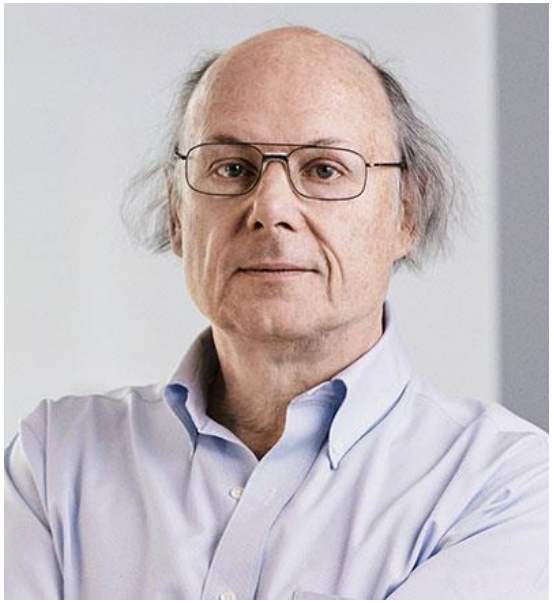
- **A programming language model**
  - programs are organized around objects
  - identify all of the objects to manipulate and how they relate to each other





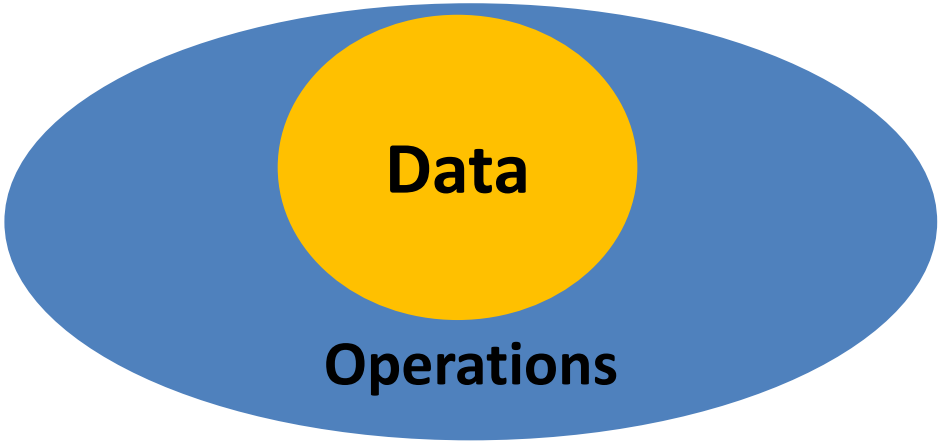
OOP: C++ Language

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



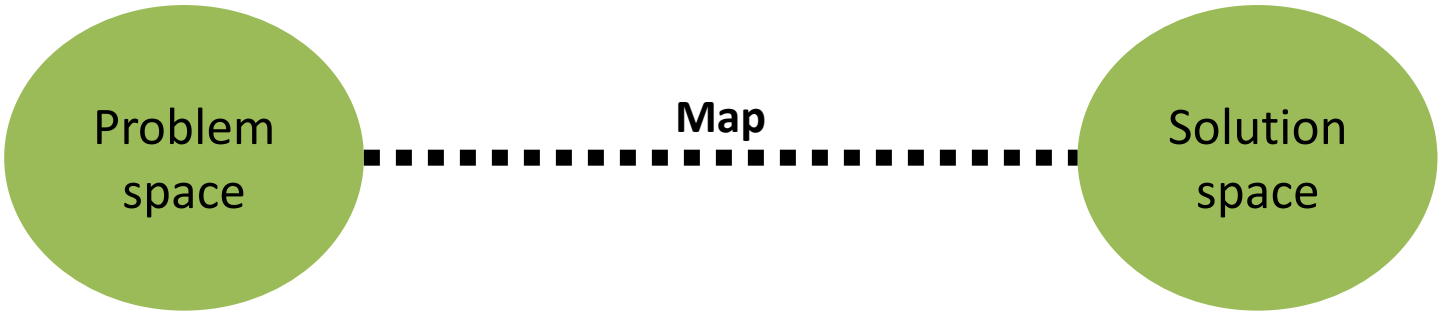
OOP: What is an object

- Object = Entity
- Object maybe visible or invisible
- Object is variable in programming language
- Objects = Attributes + Services
  - Data: the properties or status
  - Operations: the functions



Mapping

- From the problem space to the solution space



Procedural Languages in C

- C doesn't support relationship between data and functions

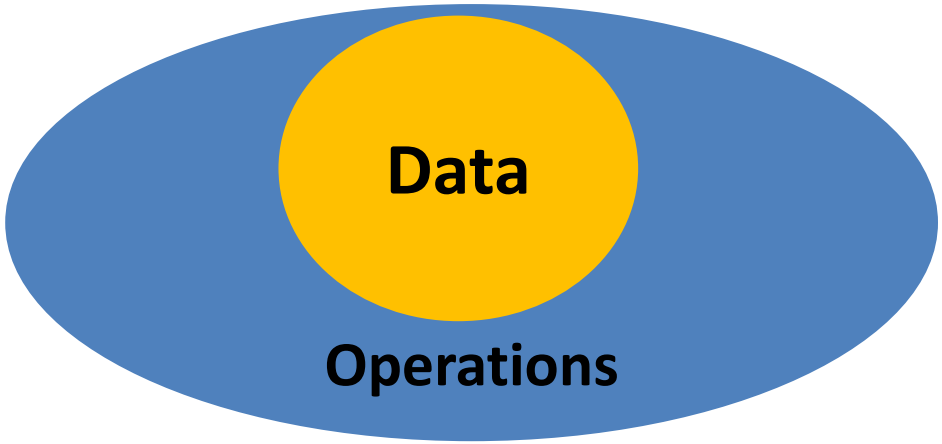
```
typedef struct point3d {
    float x;
    float y;
    float z;
} Point3d;

void Point3d_print(const Point3d* pd);

Point3d a;
a.x = 1; a.y = 1; a.z = 3;
Point3d_print(&a);
```

C++ version

```
class Point3d{
public:
    Point3d(float x, float y, float z);
    print();
private:
    float x;
    float y;
    float z;
};
Point3d a(1, 2, 3);
a.print();
```



C vs. C++

```c
typedef struct point3d {
    float x;
    float y;
    float z;
} Point3d;

void Point3d_print
        (const Point3d* pd);

Point3d a;
a.x = 1;
a.y = 1;
 a.z = 3;
Point3d_print(&a);
```

```c
class Point3d{
public:
    Point3d(float x, float y,
        float z);

    print();
private:
    float x;
    float y;
    float z;
};

Point3d a(1, 2, 3);
a.print();
```

Procedural Programming

- In C, everything we've been doing has been ***procedural programming***

- code is divided into multiple procedures
  - procedures operate on data (structures), when given correct number and type of arguments
  - program calls the procedures in sequence

- Example:
  - **`printf(<character array>,<parameters>)`**

Object-Oriented Programming

- in OOP, code and data are combined into a single entity called a ***class***
  - each ***instance*** of a given class is an ***object*** of that class type

- principles of Object-Oriented Programming
  - encapsulation
  - inheritance
  - polymorphism

OOP: Encapsulation

- ***encapsulation*** is a form of information hiding and abstraction
- data and functions that act on that data are grouped together (inside a class)

- *ideal:* separate the interface/implementation so that you can use the former without any knowledge of the latter

OOP: Inheritance

- ***inheritance*** allows us to create and define new classes from an existing class (i.e. sub-classes)

- this allows us to re-use code
  - faster implementation time
  - fewer errors
  - easier to maintain/update

OOP: Polymorphism

- ***polymorphism*** is when a single name can have multiple meanings
  - normally used in conjunction with inheritance
  - ability to decide at runtime what will be done

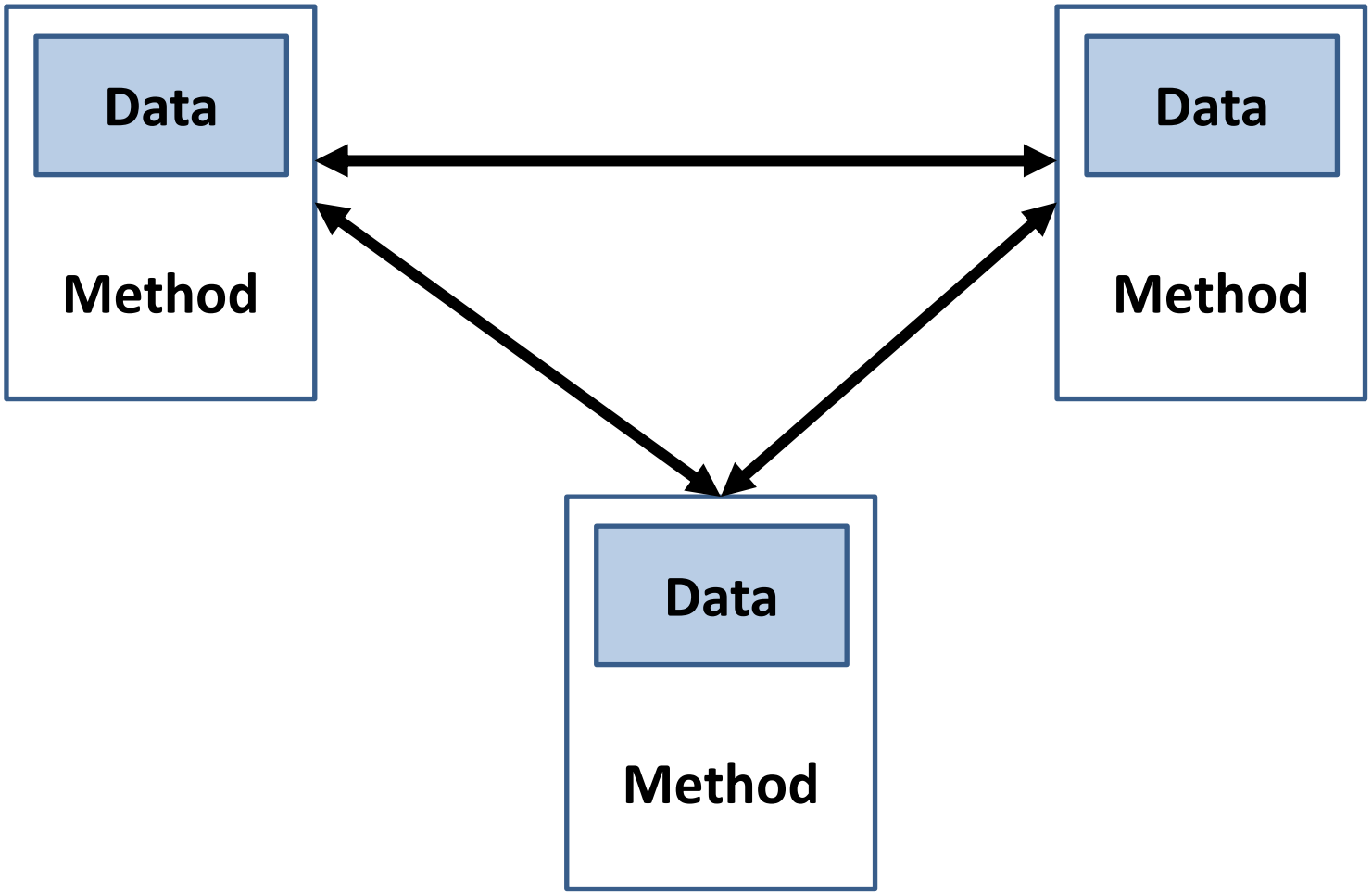
- We'll look at one form of polymorphism today:
  - overloading functions

OOP: What is an object-oriented

- A way to organize
  - Design
  - Implementation
- Objects, not control of data flow, are the primary focus of the design and implementation.
- To focus on things, not operations.

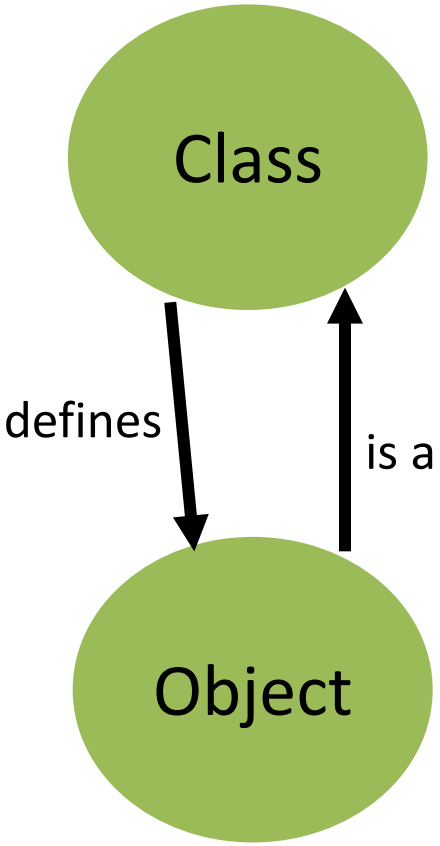
Object-Oriented Programming

- Objects send and receive messages (objects do things!)



Object vs. Class

- Objects (cat)
  - Represent things, events, concepts
  - Respond to messages at run-time
- Classes (cat class)
  - Define properties of instances
  - Act like types in C++



OOP: Characteristics

1. Everything is an object.
2. A program is a bunch of objects telling each other what to do by sending messages.
3. Each objects has its own memory made up by other objects.
4. Every object has a type.
5. All objects of a particular type can receive the same messages.

An objects has interfaces

- The interface is the way to receive messages
- It is defined in the class that the object belongs to
- Functions of the interfaces
  - Communication
  - Protection

The Hidden Implementation

- Inner part of an object, data members to present its state, and the actions it takes when messages is received are hidden
- Class creators vs. Client programmers
  - Keep client programmers' hands off portions they should not torch.
  - Allow the class creator to change the internal working of the class without worrying about how it will affect the client programmer.

OOP: Encapsulation

- ***encapsulation*** is a form of information hiding and abstraction
- Bundle data and methods dealing with these data together in an objects
- Hide the details of the data and the action
- Restrict only access to the publicized methods

Example: Ticket Machine

- Ticket machines print a ticket when a customer inserts the correct money for their fare.
- They work by customers `inserting' money into them, and then requesting a ticket to be printed. A machine keeps a running total of the amount of money it has collected throughout its operations

Ticket Machine: Procedure-Oriented

- Step to the machine
- Insert money into the machine
- The machine prints a ticket
- Take the ticket and leave

Ticket Machine: Something is there

Price

Balance

Total

Show prompt

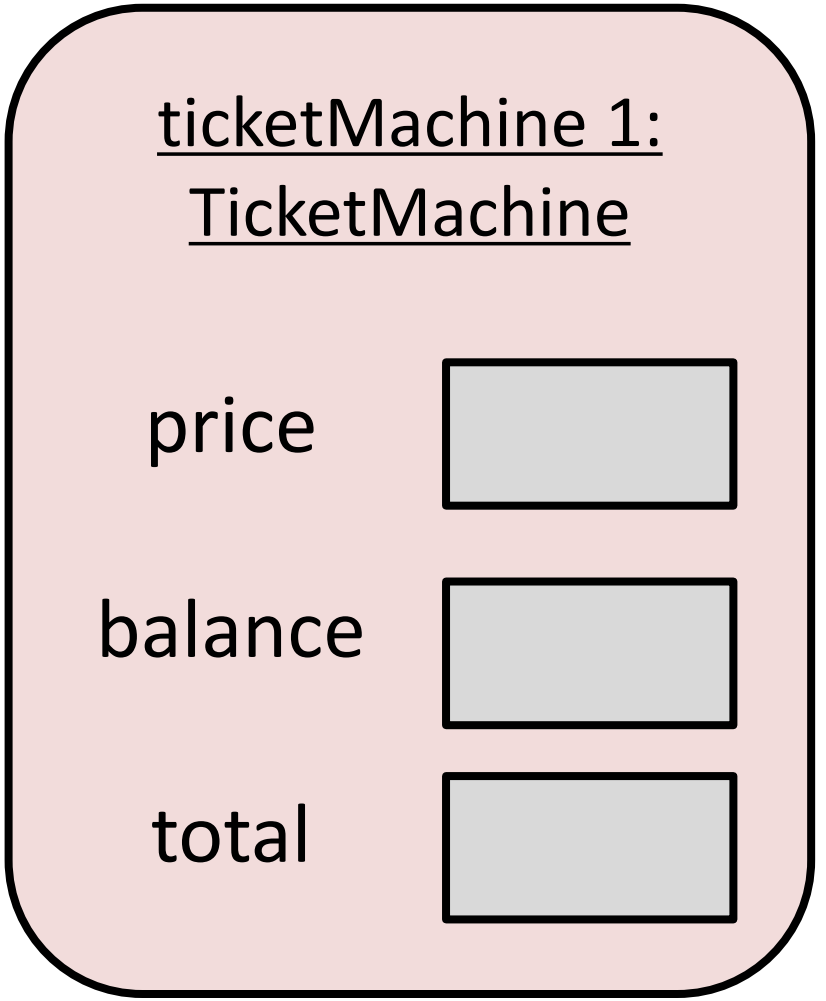
Print balance

Insert money

Print ticket

Ticket Machine: Something is there

TicketMachine
PRICE balance total
ShowPrompt getMoney printTicket showBakance printError



:: resolver

- <Class Name>::<function name>
- ::<function name>

```
Void S::f( ) {
        ::f( );     // Would be recursive otherwise
        ::a++;    // Select the global a
        a--;       // The a at class scope
}
```