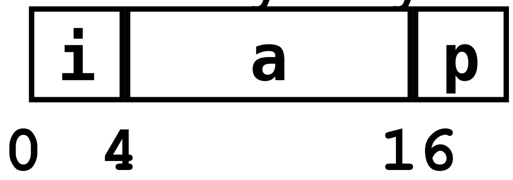
# Structs (C,C++)

# Structs

- Contiguously-allocated region of memory
- Members may be of different types
- No methods
- Example:

```
struct rec
{
  int i;
  int a[3];
  int *p;
};
```

Memory Layout



# C++

- No need to write “struct Complex” each time even if we don’t use a typedef
- Can have methods.

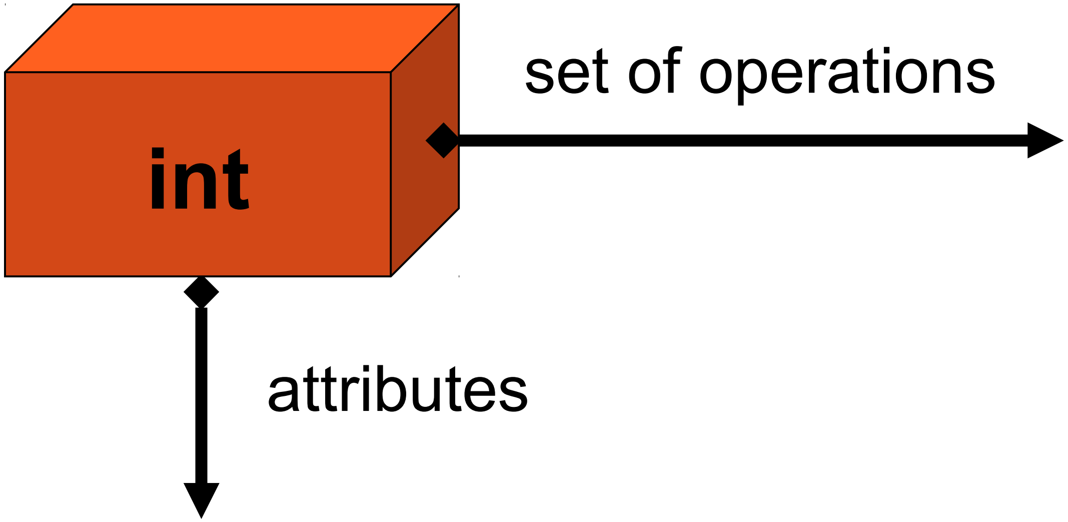
*Complex.h*

```
struct Complex
{
  double _real, _imag;
};

Complex addComplex(Complex, Complex);
Complex subComplex(Complex, Complex);
```

# Classes (C++)

# Abstract Data Type (ADT)



int

set of operations

attributes

Operations:

- numerical operators
- logical operations
- bit operations
- I/O operations

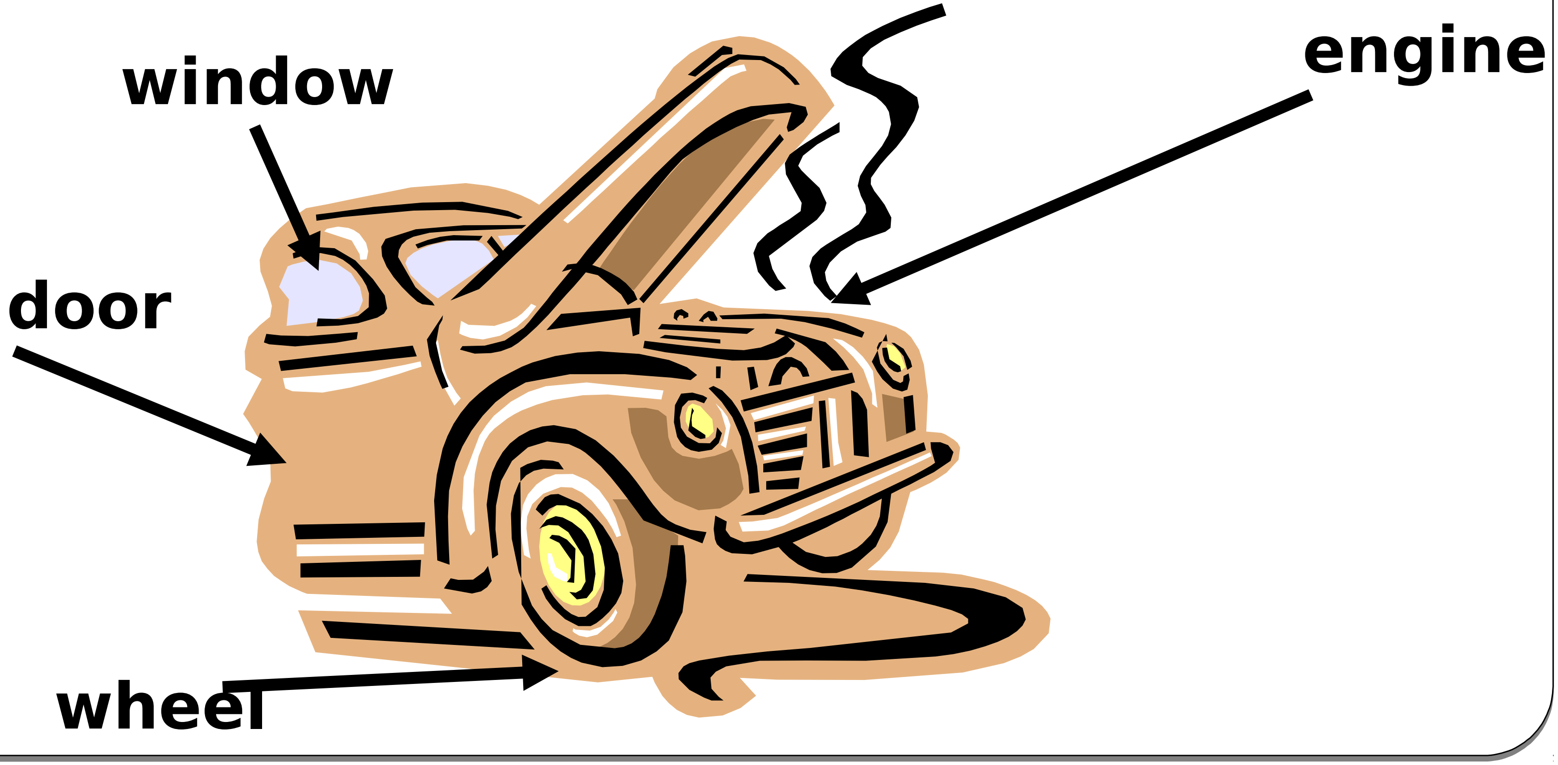
Attributes:

- 4 bytes.
- Integer numbers.

*Data Types define the way you use storage (memory) in the .programs you write*

# How should we describe a car?

- attributes
- operations



# Classes

In C++ we use **classes** for defining ADTs.

The syntax:

```
class ClassName
{
  //attributes and operations
};
```

**Objects** are instances of classes. That is, objects are to classes what variables are to types.

A class definition does not allocate storage for any objects.

# Classes

You **can** write a C++ class such that its objects can be used like **primitives** both in term of **usage** (e.g., a+b) and in terms of **efficiency**

This is different from Java where objects are restricted pointers

# Simple Class Declaration

File: Counter.hpp

It is also common to use .h for C++ headers

```
#pragma once

class Counter
{
public:
  Counter(); //Constructor
  void increment(); //A method
  int value(); //Another one
private:
  int _count;
};
```

# Class Implementation: Counter.cpp

```
#include "Counter.hpp"

void Counter::increment()
{
  _count++;
}

int Counter::value()
{
  return _count;
}
```

Scope operator

# Class Implementation: Counter.cpp

Constructor - you implement it like a function, no return type,

There might be “hidden” code inside it (more later)

```
Counter::Counter()
{
  _count= 0;
}
```

## Using the class
## File: app.cpp

```cpp
#include "Counter.hpp"
#include <cstdio>
int main()
{
  Counter cnt; // Call to constructor!
  printf("Initial value = %d\n", cnt.value());
  cnt.increment();
  printf("New value = %d\n", cnt.value());
}
```

# How do we compile it?

```
g++ -Wall -Wvla -Werror -g -D_GLIBCXX_DEBUG -std=c++11 –c Counter.cpp –o Counter.o
g++ -Wall -Wvla -Werror -g -D_GLIBCXX_DEBUG -std=c++11 –c app.cpp –o app.o

g++ -Wvla -Werror -g -D_GLIBCXX_DEBUG -std=c++11 -Wall Counter.o app.o –o app
```

# Declaration + implementation

```
#pragma once

class Counter
{
public:
  Counter(); //Constructor
  //A method with inline (we will learn about this later) implementation:
  void increment(){ _count++;}
private:
  int _count;
};
```

# Class Basics: Public/Private

Declare which parts of the class are accessible outside the class

```
class Foo
{
public:
    //accessible from outside
private:
//private - not accessible from outside //
(compilation error)
//but visible to user!
};
```

# Example

```cpp
class MyClass
{
public:
    int a();
    double _x;
private:
    int b();
    double _y;
};
```

```cpp
int main()
{
    MyClass foo;
    //legal:
    foo._x = 1.0;
    foo.a();
    //compile error:
    foo._y = 2.0;
    foo.b();
}
```

# Example

```
class MyClass
{
public:
  int a();
  double _x;
private:
  int b();
  double _y;
};
```

```
int MyClass::a()
{
  // legal
  _x = 1.0;
  // also legal
  _y = 2.0;
  b();
}
```

# Example – Point

```
class Point
{
    public:
      Point(int x, int y);
      int getX();
      int getY();
private:
      int _x, _y;
};
```

# this

The address of the instance for which the member method was invoked

# this

The address of the instance for which the member method was invoked

```
class Node
{
  Node* next;
public:
  bool isChild(const Node*);
  // …
;}
```

```
bool Node::isChild(const Node* other)
{
  for (const Node* curr = this; curr; curr = curr->next)
  {
    if (curr == other) return true;
  }
  return false;
}
```

# this

The address of the instance for which the member method was invoked

```
class Node
{
  Node* next;
public:
  bool isChild(const Node*);
  // ...
;}
```

```
bool Node::isChild(const Node* other)
{
  for (const Node* curr = this; curr; curr = curr->next)
  {
    if (curr == other) return true;
  }
  return false;
}
```

Type of “this”: Node*

# structs and classes

Where did structs go?

- In C++ class == struct, except that by default struct members are **public** and class members are **private** (also inheritance diff later):

```
struct MyStruct
{
  int x;
};
class MyClass
{
  int x;
};
```

```
int main()
{
  MyStruct s;
  s.x = 1; // ok
  MyClass c;
  c.x = 1; // error
}
```

# structs & classes

All of these are the same:

```
struct A
{
  int x;
};

struct A
{
  public:
  int x;
};

class A
{
  public:
  int x;
};
```

# structs & classes

All of these are the same (and useless):

```
class A
{
  int x;
};

class A
{
  private:
  int x;
};

struct A
{
  private:
  int x;
};
```

# Class Basics - member/static

```
class List
{
public:
  static int getMaxSize();
  int getSize();
  static int max_size=1000; //error! (only outside, below)
  int size=0; //error! (only in ctor, coming slides)
};

int List::max_size=1000; //ok, in one cpp file

int main()
{
  List l;
  l.getSize();
  List::getMaxSize();
  l.getMaxSize(); //compiles ok, but bad style
}
```

# this

```
static int List::getMaxSize() //no this!
{
  return this->size; //compile error!
  return max_size; //ok
}
int List::getSize()
{
  return this->size; //ok
}
```

# Class Basics: Constructors

Initialize the class object upon construction

```
class MyClass
{
public:
1  MyClass();
2  MyClass(int i);
3  MyClass(double x, double y);
   …
};


MyClass a; //Calls 1
MyClass b(5); //Calls 2
MyClass c(1.0, 0.0); //Calls 3
```

# Constructors – parameterless ctor

```
class MyClass
{
public:
  MyClass(); //parameterless ctor.
  //…
};
//…
int main()
{
  MyClass a; //parameterless ctor called
  // …
```

# Constructors – default parameterless ctor

```
class MyClass
{
public:
  //No ctors
  //...
};
//...
int main()
{
  MyClass a; //default parameterless
         //ctor called
  //
```

# Constructors – no default parameterless ctor

```
class MyClass
{
public:
 MyClass(int x); //no parameterless ctor.

};

int main()
{
 MyClass a; //compiler error –
     //no parameterless ctor.
```

# Destructors

1. Ensure propose “cleanup” when the object is destructed
2. Use for freeing memory, notifying related objects, etc.

# Class Basics: Destructors

```
#include <cstdlib>
class MyClass
{
public:
  MyClass();
  ~MyClass(); //destructor
private:
  char* _mem;
};
MyClass::MyClass()
{
  _mem = (char*)malloc(1000);
}
MyClass::~MyClass()
{
  free(_mem);
}
```

```
int main()
{
  MyClass a;
  if( … )
  {
    MyClass b;

  }

}
```

# C struct and functions

```
struct IntList;
typedef struct IntList IntList;
IntList* intListNew();
void intListFree(    IntList* List );
void intListPushFront( IntList* List, int x);
void intListPushBack( IntList* List, int x);
int intListPopFront( IntList* List );
int intListPopBack(  IntList* List );
int intListIsEmpty(  IntList const* List);

typedef void (*funcInt)( int x, void* Data );
void intListMAPCAR(  IntList* List,
                funcInt Func, void* Data );
```

# C++ Class

**In header file:**

```
class IntList
{
public:
  IntList();
  ~IntList();
  void pushFront(int x);
  void pushBack(int x);
  int popFront();
  int popBack();
  bool isEmpty() const;
private:
  struct Node
  {
    int value;
    Node *next;
    Node *prev;
  };
  Node* m_start;
  Node* m_end;
};
```

# Classes & Memory allocation

Consider this C++ code

```
main()
{
  IntList L;

}
```

Compare to C style:

```
main()
{
  IntList* L =
  intListNew()
}
```

What is the difference?

# Classes & Memory allocation

```
IntList* L =
(IntList*)malloc(sizeof(IntList));
```

Does not call constructor!

Internal data members are not initialized

```
free(L);
```

Does not call destructor!

Internal data members are not freed

# new & delete

Special operators:

```
IntList *L = new IntList;
```

1. Allocate memory
2. Call constructor
3. Return pointer to the constructed object

```
delete L;
```

4. Call destructor
5. Free memory

# new

Can be used with any type:

```
int *i= new int;
char **p = new (char*);
```

- new is a global operator
- new ***expression*** invokes the new ***operator*** to allocate memory, and then calls ctor
- Can be overloaded (or *replaced*)
- By default, failure throws exception. Can be changed.
- See <new> header

# Global ***operator*** new (simplified)

```
void *operator new (size_t size)
{
  void *p;
  if((p = malloc(size)) == 0)
  {
    throw std::bad_alloc;
  }
  return p;
}
```

# New & Constructors

```
class MyClass
{
public:
①MyClass();
②MyClass(int i);
③MyClass(double x, double y);


};


MyClass* a;
a = new MyClass; //Calls        ①
a = new MyClass(5); //Calls        ②
a = new MyClass(1.0, 0.0); //Calls        ③
```

# New & arrays

To allocate arrays, use

```
int *a = new int[10]; // array of 10
              // ints
size_t n = 4;
IntList *b = new IntList[n];
         // array of n IntLists
```

Objects in allocated array must have an argument-less constructor!

# Delete & arrays

Special operation to delete arrays

```
int *a = new int[10];
int *b = new int[10];

delete [] a; //proper delete command
delete b;    //may work, but may
         //cause memory leak!
```

# Allocate array of objects w/o def. cons.

```
size_t n = 4;
MyClass **arr = new MyClass*[n];
//array of n pointers to MyClass (no
//cons. is invoked)

for (size_t i=0; i<n; ++i)
{
  arr[i] = new MyClass(i);
  //each pointer points to a MyClass
  //object allocated on the heap, and
  //the cons. is invoked.
}
```

# Free an allocated array of pointers to objects on the heap

```
size_t n = 4;
for (size_t i = 0; i < n; ++i)
{
  delete (arr[i]);
  // invoked the dest. of each MyClass
  // object allocated on the heap, and
  // free the memory.
}
delete [] arr;
// free the memory allocated for the
// array of pointers. No dest. is invoked
```

We will see different (and in many cases better) alternatives to directly using new!