Objects in C++

Classes and Data Abstraction

C++ Object System

- Object-oriented features
  1. Classes and Data Abstraction
  2. Encapsulation
  3. Inheritance
     - Single and multiple inheritance
     - Public and private base classes
  - Objects, with dynamic lookup of virtual functions
  - Subtyping
     - Tied to inheritance mechanism

Abstraction

- **Abstraction** means that implementation details are hidden inside a program unit with a *specific interface*.

- For objects, **the interface** consists of a set of public functions (or methods) that manipulate hidden data.

- Abstraction involves restricting access to a program component according to its specified interface.

C++: Classes and Data Abstractio

- C++ supports Object-Oriented Programming (OOP)
- OOP models real-world objects with software counterparts
- OOP encapsulates data (attributes) and functions (behavior) into packages called objects
- Objects have the property of information hiding

C++: Classes and Data Abstractio

- Objects communicate with one another across interfaces
- The interdependencies between the classes are identified
  - makes use of
  - a part of
  - a specialisation of
  - a generalisation of
  - etc.

C and C++

- C programmers concentrate on writing functions
- C++ programmers concentrate on creating their own user-defined types calle d classes
- Classes in C++ are a natural evolution of the C notion of **`struct`**

A User-Defined Type Time with a Struct

```
// Create a structure, set its members, and print it
#include <iostream.h>

// structure definition
struct Time {
   int hour;     // 0-23
   int minute;   // 0-59
   int second;   // 0-59
};

void printMilitary(const Time &);  // prototype
void printStandard(const Time &);  // prototype
```

Standard I/O system:

1. cin The standard input
2. cout The standard output
3. cerr The error output

```
main()
{
   Time dinnerTime;  // variable of new type Time

   // set members to valid values
   dinnerTime.hour = 18;
   dinnerTime.minute = 30;
   dinnerTime.second = 0;

   cout << "Dinner will be held at";
   printMilitary(dinnerTime);  // 18:30:00
   cout << " military time,\nwhich is ";
   printStandard(dinnerTime);  //  6:30:00 PM
   cout << " standard time." << endl;
```

adds a newline ("\n") and flushes the buffer

```
    // set members to invalid values
    dinnerTime.hour = 29;
    dinnerTime.minute = 73;
    dinnerTime.second = 103;

    cout << "\nTime with invalid values: ";
    printMilitary(dinnerTime); // 29:73:103 bad values!
    cout << endl;

    return 0;
}// end main
```

```
// Print the time in military format
void printMilitary(const Time &t)
{
   cout << (t.hour < 10 ? "0" : "") << t.hour << ":"
  << (t.minute < 10 ? "0" : "") << t.minute << ":"
  << (t.second < 10 ? "0" : "") << t.second;
}

// Print the time in standard format
void printStandard(const Time &t)
{
   cout << ((t.hour == 0 || t.hour == 12) ? 12 :
  t.hour % 12)
  << ":" << (t.minute < 10 ? "0" : "") << t.minute
  << ":" << (t.second < 10 ? "0" : "") << t.second
  << (t.hour < 12 ? " AM" : " PM");
}
```

Comments

- Initialization is not required --> can cause problems
- A program can assign bad values to members of Time
- If the implementation of the **`struct`** is changed, all the programs that use the **`struct`** must be changed [No “interface”]

A Time Abstract Data Type with a Class

```
#include <iostream.h>
// Time abstract data type (ADT) definition
class Time {
public:
   Time();                    // default constructor
   void setTime(int, int, int);
   void printMilitary();
   void printStandard();
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};
```

```
// Time constructor initializes each data member to zero.
// No return value
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time.
// Perform validity checks on the data values.
// Set invalid values to zero (consistent state)
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}
```

```
// Print Time in military format
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
  << (minute < 10 ? "0" : "") << minute << ":"
  << (second < 10 ? "0" : "") << second;
}

// Print time in standard format
void Time::printStandard()
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
  << ":" << (minute < 10 ? "0" : "") << minute
  << ":" << (second < 10 ? "0" : "") << second
  << (hour < 12 ? " AM" : " PM");
}
```

```
// Driver to test simple class Time
main()
{
   Time t;  // instantiate object t of class Time

   cout << "The initial military time is ";
   t.printMilitary(); // 00:00:00
   cout << "\nThe initial standard time is ";
   t.printStandard(); // 12:00:00 AM

   t.setTime(13, 27, 6);
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary(); // 13:27:06
   cout << "\nStandard time after setTime is ";
   t.printStandard(); // 1:27:06 PM
```

```
    t.setTime(99, 99, 99);
    // attempt invalid settings
    cout << "\n\nAfter attempting invalid settings:\n"
         << "Military time: ";
    t.printMilitary(); // 00:00:00
    cout << "\nStandard time: ";
    t.printStandard(); // 12:00:00 AM
    cout << endl;

    return 0;
} // end main
```

Output

- The initial military time is 00:00:00
- The initial standard time is 12:00:00 AM

- Military time after setTime is 13:27:06
- Standard time after setTime is 1:27:06 PM

- After attempting invalid settings:
  - Military time: 00:00:00
  - Standard time: 12:00:00 AM

Comments

- **`hour`**, **`minute`**, and **`second`** are private data members. They are normally not accessibl e outside the class. [Information Hiding]
- Use a constructor to initiailize the data members. This ensures that the object is i n a consistent state when created.
- Outside functions set the values of data members by calling the setTime method, which provides error checking.

Classes as User-Defined Types

- Once the class has been defined, it can be used as a type in declarations as follows:

```
Time sunset              //object of type Time
Time arrayOfTimes[5]  //array of Time objects
Time *pointerToTime   //pointer to a Time object
```

Using Constructors

- Constructors can be overloaded, providing several methods to initialize a class.

Interface

```
Time();         //  default constructor
Time(int hr);
Time(int hr, int min, int sec);
```

Implementation

```
Time::Time(){ hour = minute = second = 0; }
Time::Time(int hr) { setTime(hr, 0, 0); }
Time::Time(int hr, int min, int sec)
    { setTime(hr, min, sec); }
```

Using Constructors

```
Time t1; // Time() is invoked
Time t1(); //ERROR, intended as a funct prototype

Time t2(08);  // class_name object_name(values)
Time t2 = Time(08);
Time t2 = 08;
Time t2 = (Time) 08; // cast

Time t3(08,15,04);
Time t3 = Time(08,15,04);
```

Using Constructors and dynamic objects

```
Type_name * pointer_name;
Pointer_name = new Type_name;
```

where Type is a Class or a primitive type

```
Int *ptr;
ptr = new int;

Time *t;
t = new Time;                // Time() is invoked
t = new Time(08);            // Time(int) is invoked
t = new Time(08,15,04); // Time(int, int, int)
                             // is invoked
```

Using Constructors and array of objects

```
Time arrayOfTimes[5]; //Time() is invoked
```

Explicit array initialization:

```
//Only the first four elements are inizialized
//Time() (if any) is invoked for the other elements
Time arrayOfTimes[8] = { 3, Time(05), Time(),
   Time(01,12,03)}
```

Using Constructors and dynamic arrays

```
Time *t = new Time[8];
// Time() is invoked for each element

int i = 3;
Time (*t) [20] = new Time[3*i] [20];
// Multi-dimension array
// Time() is invoked for each element
```

positive, can be variable

positive, constant

In both cases, explicit initialization is not allowed!

The constructor initializer list

- **A list of "constructor calls"** that appears only **in the definition of the constructor** – after the argument list
- The initialization in the list is executed before any of the main constructor code.
- This is the place to put all **const** initializations, primitive type variables and object variables, **except arrays**.

```
class Info
private:
 const int i;
 double m;
 Time t;
Public:
 Info(); // default constructor
};

Info::Info(int j, double n) : i(j), m(n), t(i) {}
```

Destructors (1)

- To guarantee cleanup when using dynamic memory
- destroys objects by
  - calling the destructors of object member variables
  - calling superclass destructors (if virtual)
- the destructor is called
  - at the end of object lifetime
  - or during a call to delete
- normally, there is no need to call the destructor explicitly

Destructors (2)

- A public function member **~class_name** with no parameters and no return values

```
Class_name::~class_name() {
//delete operations
…
}
```

- Operator **delete**
  - can be called only for an object created by **new**

```
delete ptr;
delete [] ptr;
```

new() and delete() (1)

for each `new` statement, you must provide exactly one corresponding `delete` statement

failing to do so causes memory and resource leaks and can cause undefined behaviour and very bad mood.

new() and delete() (2)

- allocating memory
  - `int* myInt = new int;`
  - `int* myIntArray = new int[10];`
- deallocating memory
  - `delete myInt;`
  - `delete[] myIntArray;`

Deleting zero pointers

If the pointer you're deleting is zero, nothing will happen.

For this reason, people often recommend setting a pointer to zero immediately after you delete it, to prevent deleting it twice.

```
delete p;
p = 0;
```

Deleting an object more than once is definitely a bad thing to do, and will cause problems.

Function Declaration

- A function is declared by

  returnType funcName(
  typename arg1, ...,
  typename argN)

- Member function can include a const modifier in their signature

  void helloWorld::sayHello(void) const

  - A const method cannot modify class mebers

- private/protected/public modifier ar ento part of the funtion declaration

Function declaration: *const* modifier

```
#include <iostream.h>
Class Car{
 private:
  int lenght;
  double weight;
 public:
  int fun_weight(double) const;
};

int Car::fun_weight(double new_weight) const
{
 // weight++; ERROR
 new_weigth += weight;
 return (int) new_weight;
}
```

Function Declaration Examples

```
double multiply(const double fac1,
                const double fac2);
```

pass-by-value

pass-by-reference *

```
void addHeader(void* buf, const Date& date);

int main(int argc, char* argv[]);
int main(int argc, char** argv);

void doSomething(SomeBigObject bo);
void doSomething(SomeBigObject* bo);
void doSomething(SomeBigObject& bo);
```

pass-by-reference &

In future!!

Call by value

- called function has its own local copy of the data
  - changes to the data are local and
  - will be discarded as soon as the namespace is left
- (highly) inefficient with large objects

Call by reference

- passes memory address of variables to the function (word-size variable)
- very efficient
  - allows variable modification avoiding double-copy

- Two possible realisations in C++
  - **`void doSomething(Data* data);`**
    - pointer-based
    - advantages and drawbacks of pointer approach
  - **`void doSomething(Data& data);`**
    - reference-based
    - no null-checking necessary

pass-by-reference **&**

In future!!

Object Variable Classification (like in C)

- Extern variables **`double x`**
  - global variables, the prefix *extern* when declared by other files
- Static extern variables **`static double x`**
  - global variables, but can't be used by other files
  - are zero-initialized by default
- Automatic internal variables
  - defined within a function/block
- Static internal variables
  - like static external variables,
  - but defined within a function/block
  - retains its state between calls to that function

```
int count_calls()
{ static int
calls=0;
//local static
return ++calls; }
```

Static member variables

- A *static variable*, member of a class, is a variable **shared by all objects** created from the class

```
Class Car{
 private:
 static int num_cars;
 public:
  …
};
//Outside initialized, like an external variable,
//even if private!
int Car::num_cars = 22;
```

Static member functions (1)

- Executed in the same manner for all objects of the given class, e.g., to open a file or to set *static variables*.
- They can't:
  - access to non static variables,
  - invoke non static functions,
  - use the pointer *this*
  - be declared *virtual*
- Constructors and destructors can't be *static*

Static member functions (2)

```
#include <iostream.h>
class Car{
 private:
 static int num_cars;
 public:
 Car(); // default constructor
 static void new_car();
};
```

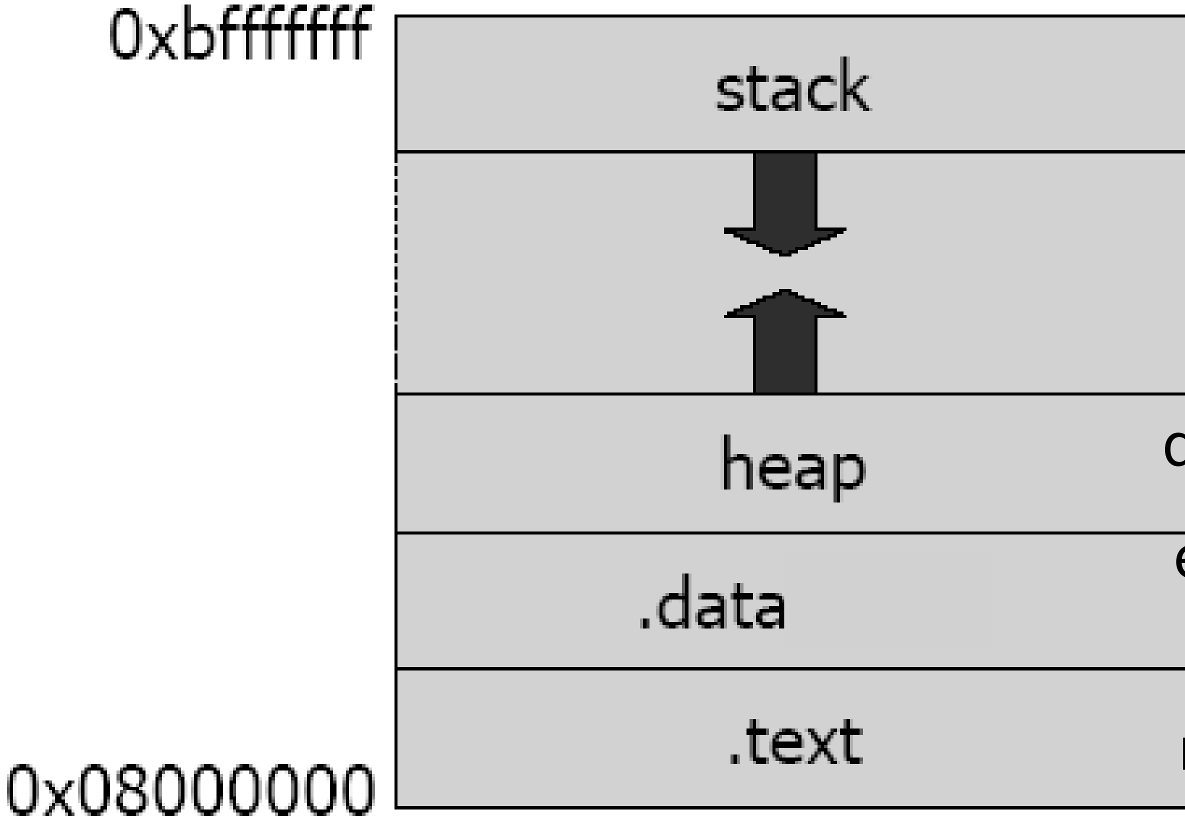
Static member functions (3)

```
Car::Car() { num_cars++; }

void Car::new_car(){cout << num_cars << '\n';}
int Car::num_cars = 0; // Access to the static
// private variable is allowed!

int main(int argc, char *argv[])
{
//cout << Car::num_cars;  ERROR Access to a
//private variable!
Car a;
Car::new_car(); // or a.new_car() bad style!
return 0;
}
```

Memory layout (1)



automatic variables
(including pointers)

dynamic variables

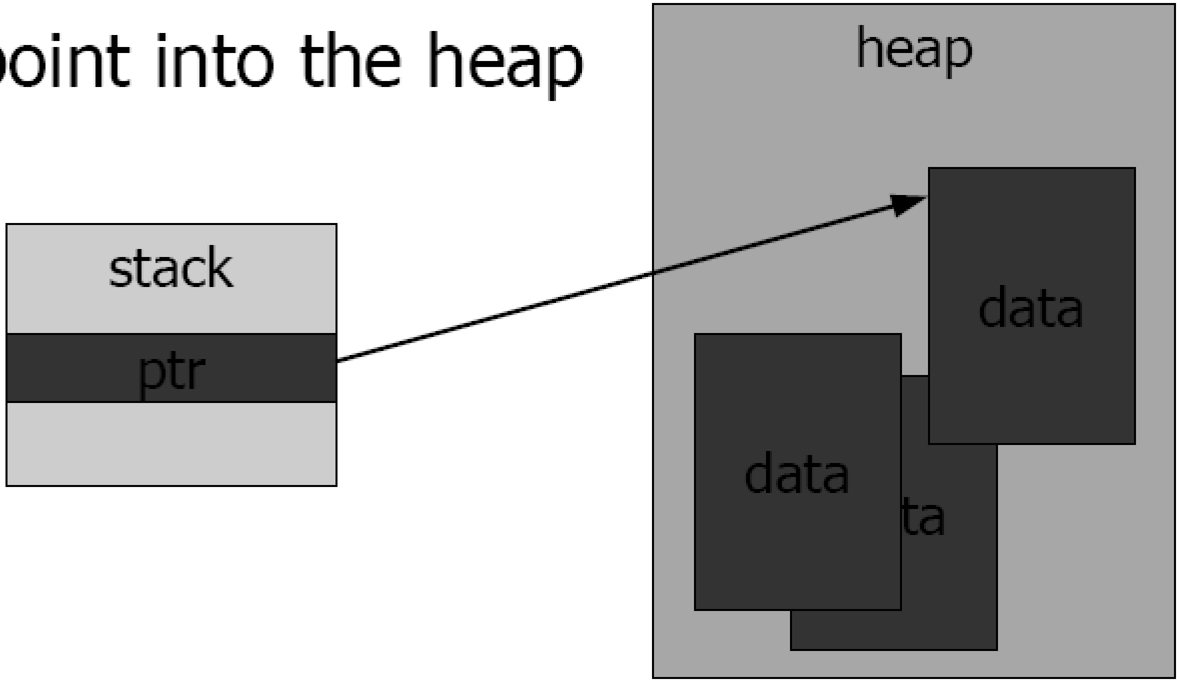
extern variables,
static variables

member functions

Memory layout (2)

Pointer have a constant size of 1 word (16 or 32 bit

- reside on the stack
- point into the heap



Inline functions

- Any function defined within a class body is automatically inline, but you can also make a non-class function inline by preceding it with the **inline** keyword.

```
inline int plusOne(int x) { return ++x; }
inline int plusOne(int x); //has no effect
```

- Any behavior you expect from an ordinary function, you get from an inline function.

- The only difference is that an inline function is expanded in place, like a preprocessor macro in C, so the overhead of the function call is eliminated.

Default arguments

- When functions have long argument lists, it is tedious to write (and confusing to read) the function calls
  - when most of the arguments are the same for all the calls.

- A commonly used feature in C++ is called *default arguments*.
  - A *default argument* is one the compiler inserts if it isn’t specified in the function call.

```
void f(int size, int initQuantity = 0);
void g(int x, int = 0, float = 1.1);
void h(int = 0, int x, float = 1.1); //ERROR
```

Function overloading

```
void f(int size, int initQuantity);
void f(int size, double initQuantity);
int f(int size, int initQuantity);//ERROR
```

- The compiler resolves the correct version of an overloaded function based on the number/type of arguments in each call
- Functions differing only in their return type cannot be overloaded.
  - Since the returned value may be implicitly converted, the compiler cannot resolve which version is intended to use
- An immediately useful place for overloading is in constructors.