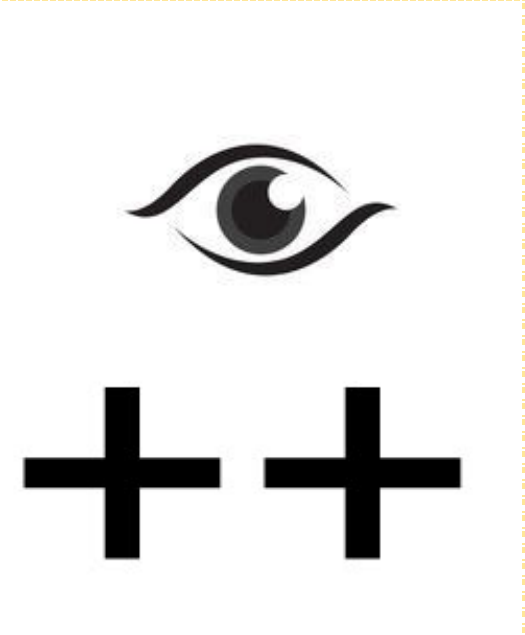
Chapter 1
OOP Revision

By: Dr. Aryaf A. Al-adwan
Faculty of Engineering Technology
Computer and Networks Engineering Dept.
Data Structures Course

Outline

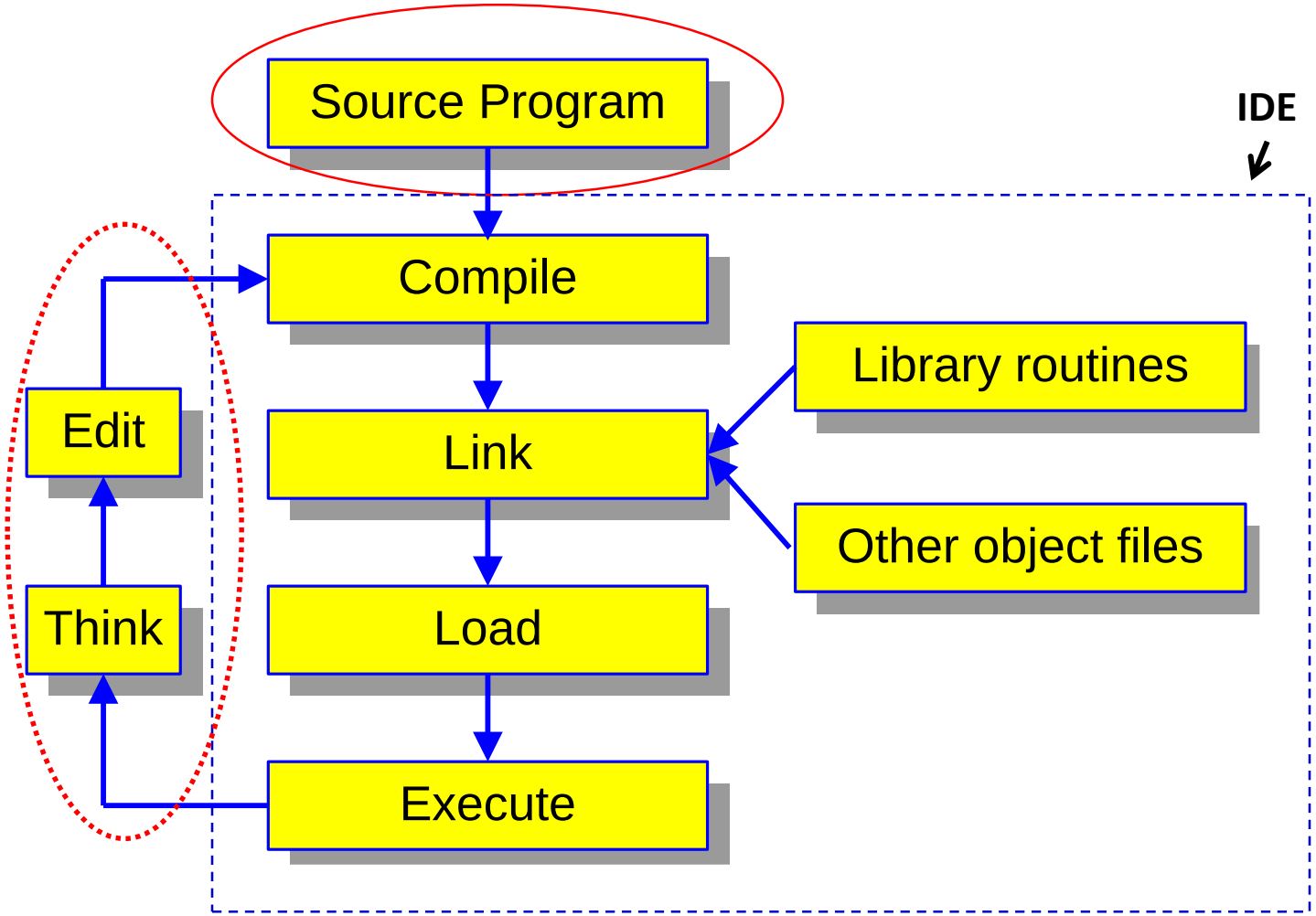
- Introduction to C++
- Functions
- Classes
- Pointers
- Dynamic Memory Allocation
- Templates



Introduction

Introduction to Programming in C++

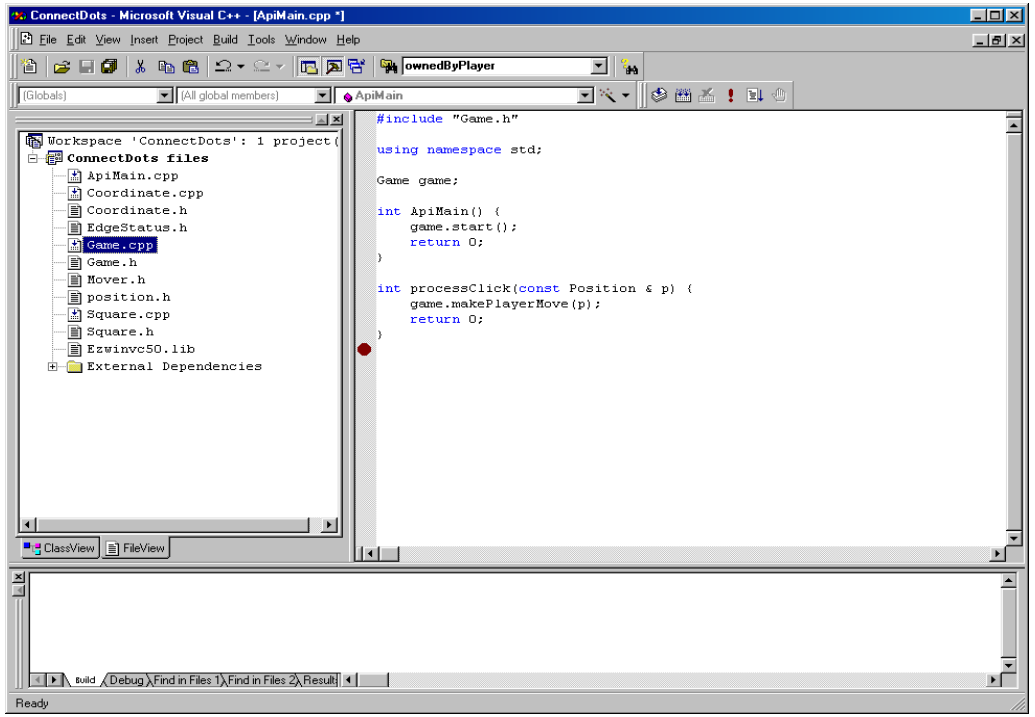
Software Development Cycle



IDE

- Integrated Development Environments or IDEs
  - Supports the entire software development cycle
    - E.g. MS Visual C++, Borland, Code Warrior

- Provides all the capabilities for developing software
  - Editor
  - Compiler
  - Linker
  - Loader
  - Debugger
  - Viewer



What are the Key Ingredients of a Program?

- Common elements in programming languages:
  - Keywords (reserved words)
    - Defined in C++. Have a special meaning
  - Programmer-defined entities
    - Names made up by programmer
    - Used to represent variables, functions, etc.
  - Operators
    - Defined in C++
    - For performing arithmetic, comparison, etc. operations
  - Constructs
    - Defined in C++
    - Sequence, selection, repetition, functions, classes, etc.

Every programming language has a syntax (or grammar) that a programmer must observe

The syntax controls the use of the elements of the program

Object Oriented Programming

- C++ is derived from C programming language – an extension that added object-oriented (OO) features.

  - Lots of C++ syntax similar to that found in C, but there are differences and new features.

- Popular because of the reuse of objects
  - Classes – template or blueprint for an object
  - Objects
  - Methods

Program Organization

- Program statement
  - Definition, e.g. function prototype
  - Declaration, e.g. variables and constants
  - Action
- Executable unit - Action
  - Set of program statements – may be named
  - Different languages refer to named executable units by different names
    - Subroutine: Fortran and Basic
    - Procedure: Pascal
    - Function: C++
    - Method: C++, C#, Java

The simplest C++ program consists of a single function named main.

The syntax of such programs is shown below:

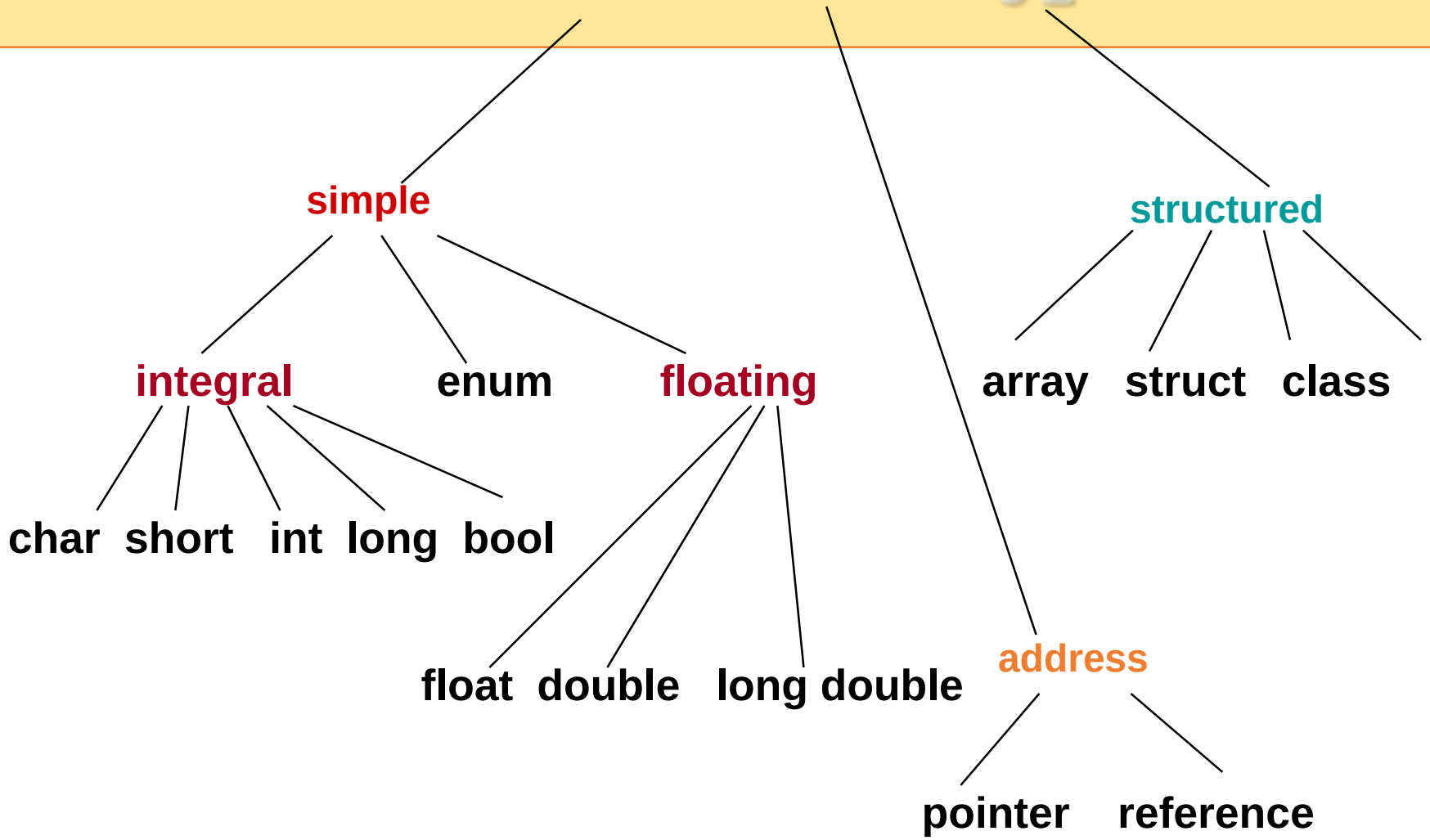
```
#include <iostream>
using namespace std;

int main()
  {
  declaration(s)
  statement(s)
  return 0;
  }
```

The portions of the program shown in blue should always be present. The declarations specify the data that is used by the program. These declarations can declare either constants or variables, for example.

The statements specify the algorithm for the solution to your problem.

C++ Data Types


simple
structured
integral
enum
floating
array
struct
class
char
short
int
long
bool
float
double
long double
address
pointer
reference

Functions

- Built in Functions
- User defined Functions
- Function Properities :

1) Name
2) Type
3) Parameters

Function Example

```
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a + b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

Classes

Object-Oriented Programming
Introduction to Classes

- Class Definition: it is a user defined data type that has Properities (data members) and operations (member functions).
- Class Examples
- Objects
- Constructors
- Destructors

Define a Class Type

Header

```
class class_name
{
    permission_label:
        member;
    permission_label:
        member;
    ...
};
```

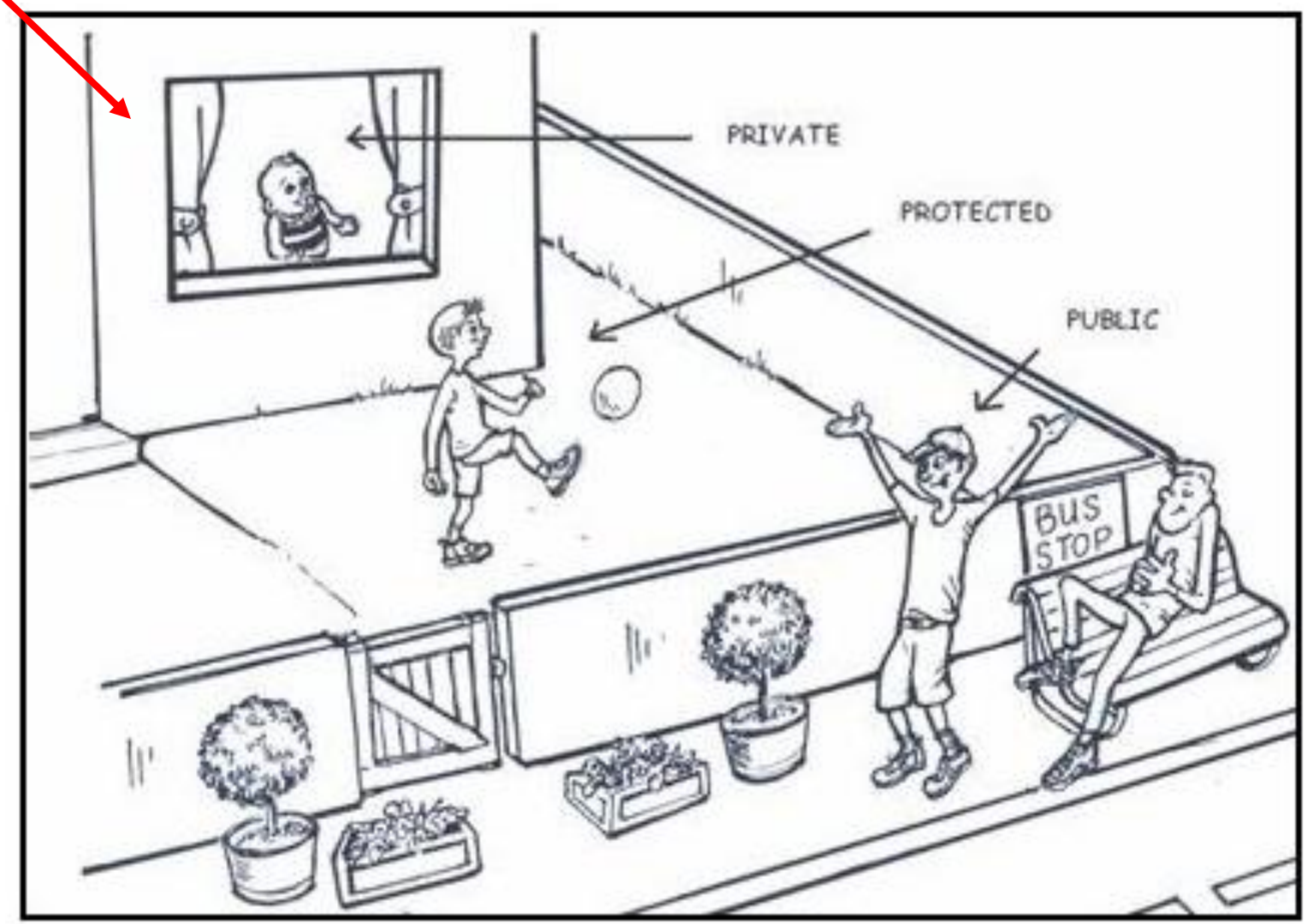
Body

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

Class Definition - Access Control

- Information hiding
  - To prevent the internal representation from direct access from outside the class
- Access Specifiers
- Public is the simplest modifier. A component set as public can be accessed anywhere.
- A private component, on the other hand, can only be accessed within its class.
- A protected component stands in the middle. It can be accessed within its class and all derived classes.

Main Class



Define a Member Function

```
class Rectangle
{
    private:
      int width, length;
    public:
      void set (int w, int l);
      int area() {return width*length; }
};
```

inline

```
r1.set(5,8);
```

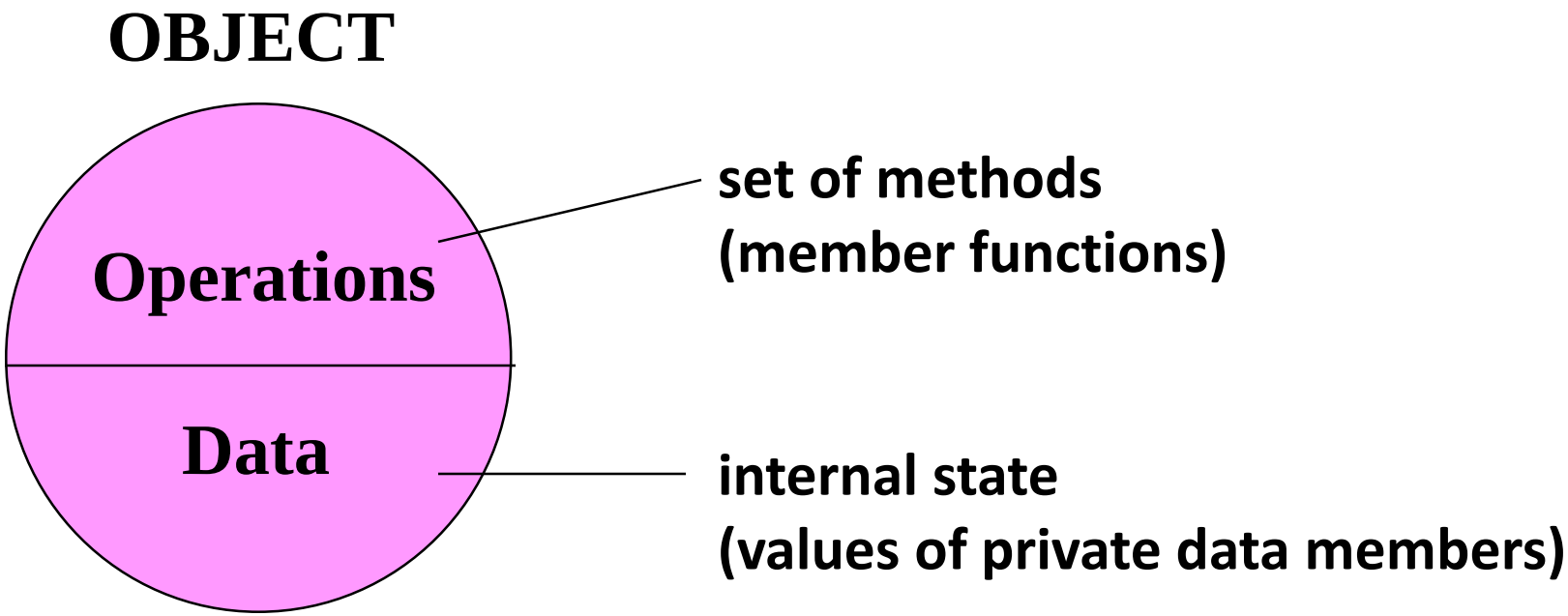
class name

member function name

```
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

scope operator

What is an object?



Declaration of an Object

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
     int color;
      void set(int w, int l);
      int area()
     {
       return (width * length);
}
};
```

```
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

```
int main()
{
    Rectangle r1;
    Rectangle r2;

    r1.set(5, 8);
    cout<<r1.area()<<endl;

    r2.set(8,10);
    cout<<r2.area()<<endl;
}
```

Another Example

```
#include <iostream.h>

class circle
{
  private:
     double radius;

  public:
     void store(double);
     double area(void);
     void display(void);

};
```

```
// member function definitions

void circle::store(double r)
{
   radius = r;
}
double circle::area(void)
{
   return 3.14*radius*radius;
}
void circle::display(void)
{
   cout << “r = “ << radius << endl;
}
```

```
int main() {
    circle c;   // an object of circle class
    c.store(5.0);
    cout << "The area of circle c is " << c.area() << endl;
    c.display();
}
```

Object Initialization

```
#include <iostream.h>

class circle
{
  public:
     double radius;
};
```

1. By Assignment

- Only work for public data members
- No control over the operations on data members

```
int main()
{
   circle c1;                    // Declare an instance of the class circle
   c1.radius = 5;        // Initialize by assignment

}
```

Object Initialization

```
#include <iostream.h>

class circle
{
  private:
    double radius;

   public:
    void set (double r)
            {radius = r;}
    double get_r ()
            {return radius;}

};
```

2. By Public Member Functions

```
int main(void) {
    circle c;              // an object of circle class
    c.set(5.0);            // initialize an object with a public member function
    cout << "The radius of circle c is " << c.get_r() << endl;
      // access a private data member with an accessor
}
```

Object Initialization

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        Rectangle();
        Rectangle(int w, int l);
        void set(int w, int l);
        int area();
}
```

3. By Constructor

- Default constructor
- Constructor with parameters

They are publicly accessible

Have the same name as the class

There is no return type

Are used to initialize class data members

Object Initialization

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
      Rectangle(int w, int l)
            {width =w; length=l;}
      void set(int w, int l);
      int area();
}
```

- Initialize with constructor

```
Rectangle r5(60,80);
```

Cleanup of An Object

```
class HelloWorld{
public:
  //Constructor
  HelloWorld(){
    cout<<"Constructor is called"<<endl;}
  //Destructor
  ~HelloWorld(){
    cout<<"Destructor is called"<<endl;
   }
   //Member function
   void display(){
     cout<<"Hello World!"<<endl;
   }
};
void main(){
   //Object created
   HelloWorld obj;
   //Member function called
   obj.display();
}
```

Output:

```
Constructor is called
Hello World!
Destructor is called
```

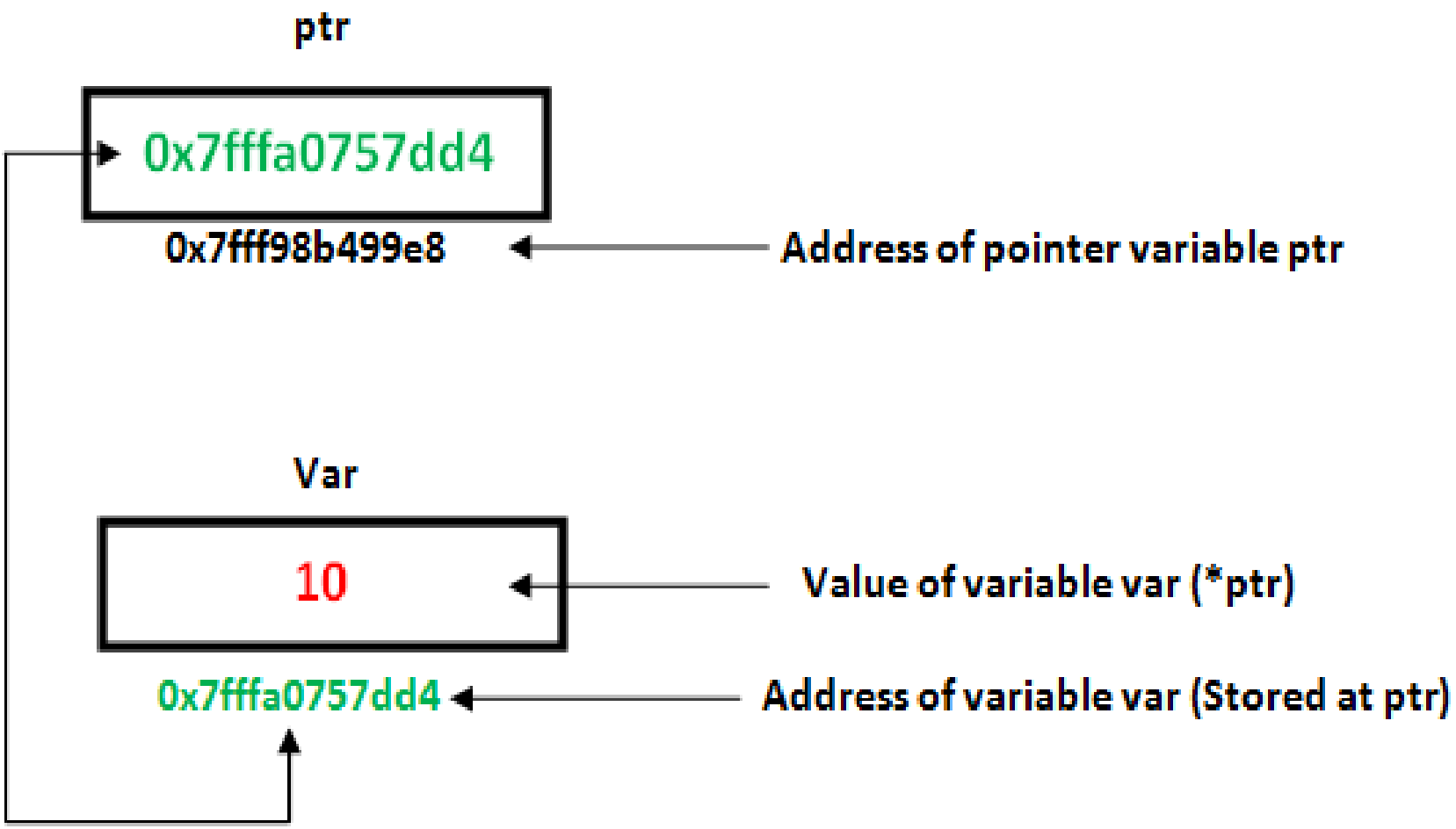
Destructor

```
Account :: ~Account()
{
    delete[] name;
}
```

- Its name is the class name preceded by a ~ (tilde)
- A destructor is a special member function that works just opposite to constructor,
  unlike **constructors** that are used for initializing an object, destructors destroy (or delete) the object.

Pointers

Pointers and memory concept



Review: Pointers & Dynamic Data

- A pointer variable is a variable whose value is the address of a location in memory

```
  int x;
   x = 5;

int* ptr1;
ptr1 = &x;

int* ptr2;
ptr2 = ptr1;
*ptr1 = 6;

cout << ptr1 << endl;
cout << *ptr2 << endl;
```

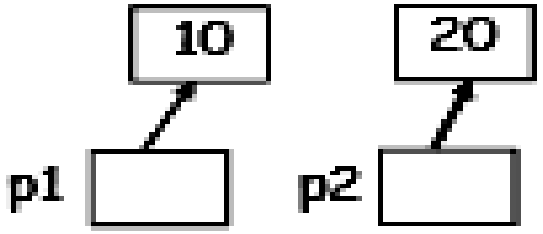
Examples

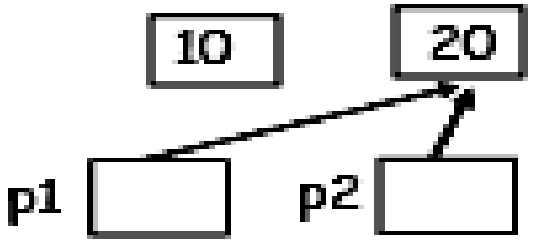
```
int x;
x=9;
int *w;
w=&x;
cout<<x;            //9
cout<<&x;           //0044
cout<<w;            //0044
cout<<*w;           //9
*w=10;
cout<<x;            //10
```

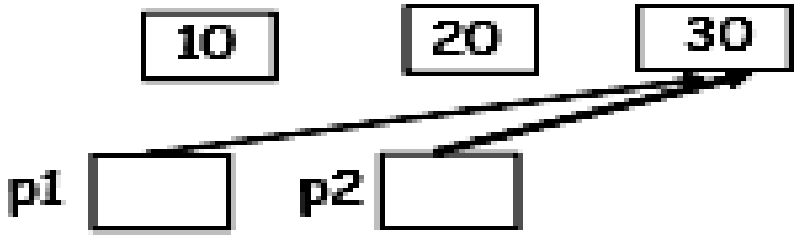
```
int v1=5,v2=15;
int *p1,*p2;
p1=&v1;
p2=&v2;
*p1=10;         //v1=10
*p2=*p1;        // v2=10
p1=p2;          // p1 addressed v2
*p1=20;
cout<<v1<<v2;   // 10     20
cout<<p1<<p2;   // same address of v2
```

- Exercise: What is the output produced by the following code?

```
int *p1, *p2;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;      // 10 20
p1 = p2;
cout << *p1 << " " << *p2 << endl;      // 20 20
*p1 = 30;
 cout << *p1 << " " << *p2 << endl;     // 30 30
```







Memory Allocation

Memory Allocation

- **Static Memory Allocation** (Compile Time Allocation): by OS at compile time, the allocation will be on the physical memory (RAM). **The stack** – All variables declared inside the function will take up memory from the stack.
- **Dynamic Memory Allocation**

1. Memory allocated "on the fly" during run time
2. dynamically allocated space usually placed in a program segment known as the ***heap***
3. Exact amount of space or number of items does not have to be known by the compiler in advance.
4. Use **new** and **delete** keywords.

Dynamic Memory Allocation (DMA)

- **Dynamic allocation requires two steps:**

1. Creating the dynamic space.
2. Storing its address in a pointer (so that the space can be accessed)

- To dynamically allocate memory in C++, we use the **new** operator.
- De-allocation:
- Deallocation is the "clean-up" of space being used for variables or other data storage
- Compile time variables are automatically deallocated based on their scope.
- It is the programmer's job to deallocate dynamically created space
- To de-allocate dynamic memory, we use the **delete** operator

What are applications of DMA?

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don’t need anymore. There are many cases where this flexibility helps

Memory Leaks

- Memory leaks refer to memory that has been allocated by an application, but not properly released back once that memory is no longer needed. Many systems have multiple applications running on their systems and programming errors usually result in “memory leaks”. Memory leaks may not affect the functionality of the application, but left unattended in a system, memory leaks can cause the machine to crash. This is why, servers restart often to avoid them to from going down. Programmers typically allocate memory and then somehow may lose the reference to that memory block.

When does the destructor get called?

A destructor is automatically called when:
1) The program finished execution.
2) When a scope (the { } parenthesis) containing local variable ends.
3) When you call the delete operator.

Example1

```
// declare an int pointer
int* pointVar;
// dynamically allocate memory
// using the new keyword
pointVar = new int;
// assign value to allocated memory
*pointVar = 45;
```

Example 2

```
int* a = NULL;              // Pointer to int, initialize to nothing.
int n;                      // Size needed for array
cin >> n;                   // Read in the size
a = new int[n];             // Allocate n ints and save ptr in a.
for (int i=0; i<n; i++)
{
   a[i] = 0;                // Initialize all elements to zero.
}
. . .  // Use a as a normal array

delete [] a;                // When done, free memory pointed to by a.
a = NULL;                   // Clear a to prevent using invalid memory reference.
```

NOTE: In this example, the user is allowed to specify any size for the array during run time. This means the array's size is determined during runtime, not the compile time.

Templates

Templates

- Templates are powerful features of C++ which allows you to write generic programs.
- In simple terms, you can create a single function or a class to work with different data types using templates.
- The concept of templates can be used in two different ways:

1. Function Templates
2. Class Template

Function Templates

```
#include <iostream>
using namespace std;

template <class T>
T add (T num1, T num2)
{
    return (num1 + num2);
}

void main()
 {
    int result1;
    double result2;
    // calling with int parameters
    result1 = add <int> (2, 3);
    cout << "2 + 3 = " << result1 << endl;

    // calling with double parameters
    result2 = add <double> (2.2, 3.3);
    cout << "2.2 + 3.3 = " << result2 << endl;
}
```

Class Template

```
template <class T>
class myclass
{
  T a, b;
  public:
  myclass (T first, T second)
    {
    a=first; b=second;
    }
  T getMaxval ();
};
template <class T>
T myclass<T>::getMaxval ()
{
  return (a>b? a : b);
}
```

```
void main ()
{
  myclass <int> m (100, 75);
  cout<<"Maximum of 100 and 75 = "<<m.getMaxval()<<endl;

  myclass<char> m('A','a');
  cout<<"Maximum of 'A' and 'a' = "<<m.getMaxval()<<endl;
}
```

Output:

Maximum of 100 and 75 = 100
Maximum of ‘A’ and ‘a’ = a

End