

Memory Management

Polymorphism

Man Geni Addysg Uwch yng Nghymru
Where Higher Education Began in Wales

www.uwtsd.ac.uk



Introduction to Memory Management:

- DYNAMIC MEMORY ALLOCATION & DEALLOCATION (new & delete)
- C++ supports these functions, it defines two unary operators **new** and **delete** that perform the task of allocating and
- deallocating the memory in a better and easier way.
- A object can be created by using **new**, and destroyed by using **delete**.

syntax for using the new operator :

```
pointerVariable = new dataType;
```

syntax for delete operator:

```
delete pointerVariable;
```

C++ new Operator

```
// declare an int pointer
```

```
int* pointVar;
```

```
// dynamically allocate memory using the new keyword
```

```
pointVar = new int;
```

```
// assign value to allocated memory
```

```
*pointVar = 45;
```

- Consider the code

```
// declare an int pointer
```

```
int* pointVar;
```

```
// dynamically allocate memory for an int variable
```

```
pointVar = new int;
```

```
// assign value to the variable memory
```

```
*pointVar = 45;
```

```
// print the value stored in memory
```

```
cout << *pointVar; // Output: 45
```

```
// deallocate the memory
```

```
delete pointVar;
```

C++ Program to store GPA of n number of students and display it
// where n is the number of students entered by the user

```
int main() {  
    int num;  
  
    cout << "Enter total number of  
students: ";  
  
    cin >> num;  
  
    float* ptr;  
  
    ptr = new float[num];  
  
    cout << "Enter GPA of students." <<  
endl;  
  
    for (int i = 0; i < num; ++i) {  
        cout << "Student" << i + 1 << ": ";  
  
        cin >> *(ptr + i);  
  
    }
```

```
        cout << "\nDisplaying GPA of  
students." << endl;  
  
        for (int i = 0; i < num; ++i) {  
            cout << "Student" << i + 1 <<  
": " << *(ptr + i) << endl;  
        }  
  
        delete[] ptr;  
  
        return 0;  
    }
```

Exercise

write a program to find sum of list of integers

Note: get the list of numbers in an array

C++ new and delete Operator for Objects

```
class Student {  
    private:  
        int age;  
    public:  
        // constructor initializes age to 12  
        Student() : age(12) {}  
        void getAge() {  
            cout << "Age = " << age  
            << endl;  
        }  
};
```

```
int main() {  
    // dynamically declare Student object  
    Student* ptr = new Student();  
    // call getAge() function  
    ptr->getAge();  
    // ptr memory is released  
    delete ptr;  
    return 0;  
}
```

Output:
Age:12

```
class base
{
public:
int a;
void get_a(int x)
{
a=x;
}
void display_a()
{
cout<<"In base " <<
"\n"<<"a="<<a<<endl;
}
};
```

```
class derived : public base
{
int b;
public:
void get_ab(int x,int y)
{
a=x;
b=y;
}
void display_ab()
{
cout<<"In Derived " <<
"\n"<<"a="<<a<<"\n b="<<b<<endl;
}
};
```

```
int main()
{
    base b;
    base *bptr;
    bptr=&b;
    //points to the object of
    base class
    bptr->get_a(100);
    bptr->display_a();
    derived d;
    derived *dptr;
    dptr=&d;//points to the
    object of derived class
```

```
    dptr->get_a(400);
    dptr->display_a();
    dptr->get_ab(300,200);
    dptr->display_ab();
    bptr=&d;
    //points to the object of
    derived class
    bptr->get_a(400);
    bptr->display_a();
    return 0;
}
```

```
int main()
{
    base b;
    base *bptr;
    //points to the object of
    base class
    bptr=&b;
    bptr->get_a(100);
    bptr->display_a();
    derived d;
    derived *dptr;
    dptr=&d;//points to the
    object of derived class
```

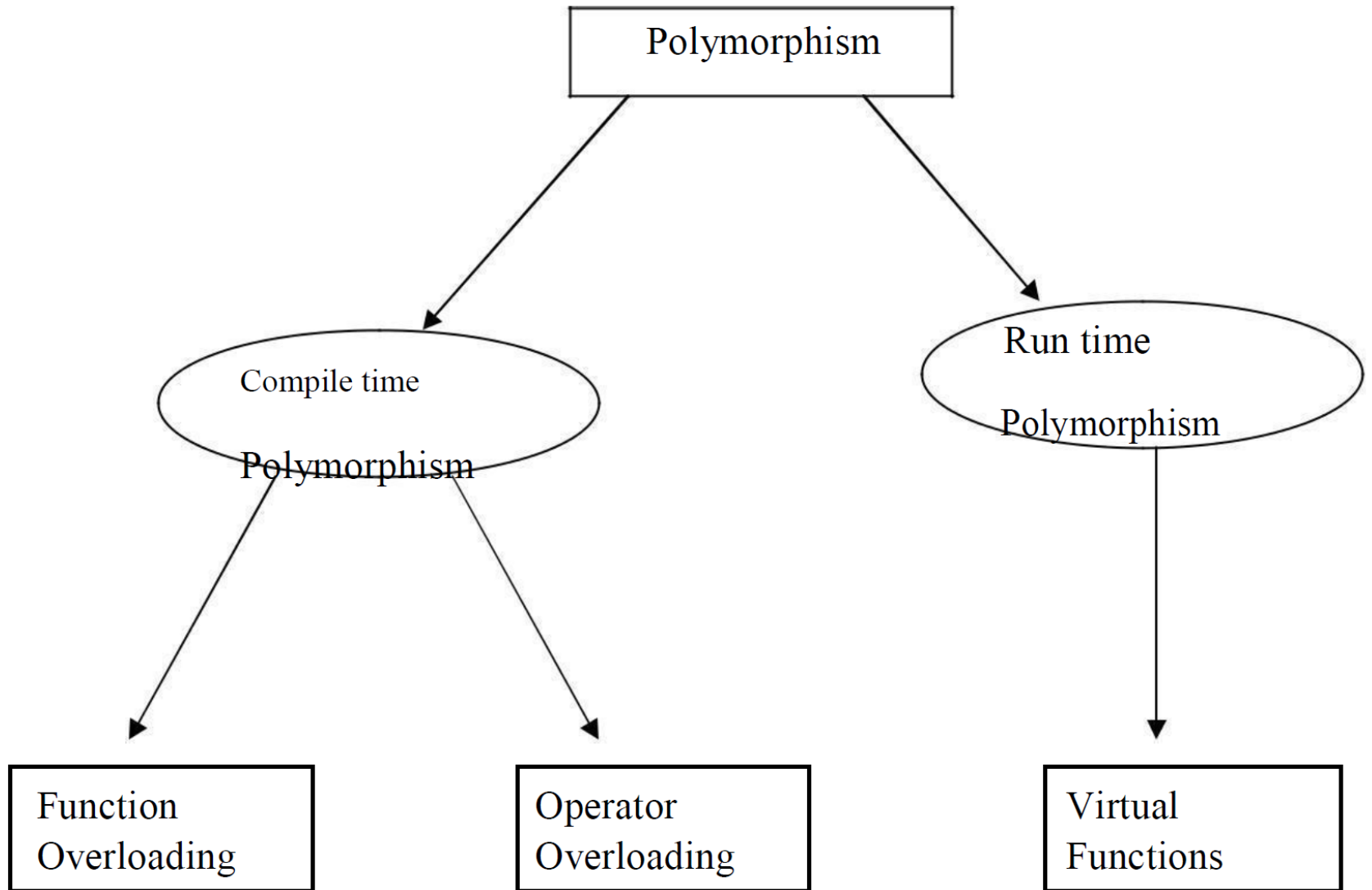
```
dptr->get_a(400);
dptr->display_a();
dptr->get_ab(300,200);
dptr->display_ab();
bptr=&d;//points to the
object of derived class
bptr->get_a(400);
bptr->display_a();
return 0;
}
```

Output:

In base
a=100
In base
a=400
In Derived
a=300
b=200
In base
a=400

Polymorphism

- “poly” means many and “morphism” means form i.e.. many forms. Polymorphism means the ability to take more than one form.
- access objects of different types through the same interface.
- For example, an operation have different behavior in different instances.
- Different ways to achieving polymorphism
 - * Function overloading
 - * Operator overloading
 - * Virtual functions



Compile time Polymorphism

- functions are „selected“ for invoking by matching arguments, both type and number
- compiler gets information at compile time and able to select the appropriate function for a particular call at the compile time.
- **Early Binding or Static Binding or Static Linking.**

Run time Polymorphism

- appropriate member function could also be selected while the program is running.
- C++ supports a mechanism known as virtual function to achieve run time polymorphism.

Early Binding or Static Binding or Static Linking.

- functions are „selected“ for invoking by matching arguments, both type and number
- compiler gets information at compile time and able to select the appropriate function for a particular call at the compile time.

Dynamic Binding

- appropriate member function could also be selected while the program is running.
- C++ supports a mechanism known as virtual function to achieve run time polymorphism.

VIRTUAL FUNCTIONS

- An essential requirement of polymorphism is the ability to refer to objects without any regard to their classes.
- When same function name is used in both base and derived classes, the function in the base class is declared as virtual using the keyword `virtual` preceding its normal declaration.
- Here, C++ determines which function to use at runtime based on the type of object rather than the type of the pointer.

Example

```
class Base
{
public:
void display()
{
cout<<"Display Base";
}
virtual void show()
{
cout<<"Show Base";
}
};
```

```
class Derived : public Base
{
public:
void display()
{
cout<<"Display Derived";
}
void show()
{
cout<<"show derived";
}
};
```

```
void main()
{
    Base b;
    Derived d;
    Base *ptr;
    cout<<"ptr points to Base";
    ptr=&b;
    ptr->display(); //calls Base
    ptr->show(); //calls Base
    cout<<"ptr points to derived";
    ptr=&d;
    ptr->display(); //calls Base
    ptr->show(); //class Derived
```

Output:

ptr points to Base

Display Base

Show Base

ptr points to Derived

Display Base

Show Derived

Rules For Virtual Functions:

When virtual functions are created for implementing late binding, observe some basic rules that satisfy the compiler requirements.

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer points to any type of the derived object, the reverse is not true. i.e. we cannot use a pointer to a derived class to access an object of the base class type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

OVERLOADING-OPERATOR OVERLOADING

We can overload all the operators except the following:

- Class member access operator (“.” And “.*”)
- Scope resolution operator “::”
- Size operator (sizeof)
- Conditional operator

OVERLOADING-OPERATOR OVERLOADING

The process of overloading involves following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function **operator op()** in the public part of the class. It may be a member function or a friend function.
3. Here op is the operator to be overloaded.
4. Define the operator function to implement the required operations.

General syntax:-

```
return-type classname :: operator op(arglist)
```

```
{
```

Function body

```
}
```

Concept of Operator Overloading

For example operator $+$ can be used as concatenate operator as well as additional operator.

$2+3$ means 5 (addition), where as

"2"+"3" means 23 (concatenation).

Operator overloading concept can be applied in following two major areas (Benefits)

1. Extension of usage of operators
2. Data conversions

Unary Operator Overloading

Syntax for Unary Operator (Inside a class)

```
return-type operator operatorsymbol( )  
{  
    //body of the function  
}
```

Example:

```
void operator-()  
{  
    real=-real;  
    img=-img;  
}
```

Syntax for Unary Operator (Outside a class)

```
return-type classname::operator operatorsymbol( )  
{  
    //body of the function  
}
```

Example 1	Example 2
<pre>void operator++() { counter++; }</pre>	<pre>void complex::operator-() { real=-real; img=-img; }</pre>

program to overload unary operator !

```
class fact {  
    int a;  
    public:  
    fact ()  
    {   a=0; }  
    fact (int i)  
    {  
        a=i;  
    }  
    fact operator!()  
    {  
        int f=1,i;  
        fact t;
```

```
        for (i=1;i<=a;i++)  
        {  
            f=f*i;  
        }  
        t.a=f;  
        return t;  
    }  
  
    void display()  
    {  
        cout<<"The factorial "<< a;  
    }  
};
```

program to overload unary operator !

```
void main()
{
    int x;
    cout<<"enter a number";
    cin>>x;
    fact s(x),p;
    p=!s;
    p.display();
}
```

Output :

Enter a number 5

The factorial of a given number

120

Exercise

- Write a program to negate the given values using unary operator –

Sample output:

x=10

y=-20

after negation

x=-10

y=20