Polymorphism

Polymorphism

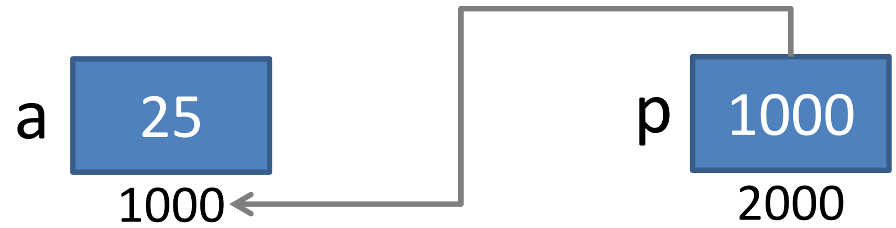
- Pointers in C++
- Pointers and Objects
- this pointer
- virtual and pure virtual functions
- Implementing polymorphism

Pointers in C++

Pointer variable

- Pointer is a variable that holds a memory address, of another variable.

```
int a = 25;
int *p;
p = &a;
```




```
cout<<"&a:"<<&a;
cout<<"p:"<<p;
cout<<"&p:"<<&p;
cout<<"*p:"<<*p;
cout<<"*(&a):"<<*(&a);
(*p)++;
cout<<"*p:"<<*p;
cout<<"a:"<<a;
```

&a:1000

p:1000

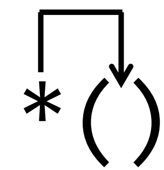
&p:2000

*p:25

*(&a):25

*p:26

a:26



* Indicates value at address

Pointer to arrays

```
int main ()
{
    int arr[5] = {10,20,30,40,50};
    int *ptr;

    ptr = &arr[0];
    for ( int i = 0; i < 5; i++ )
    {
        cout <<"*(ptr+"  << i <<"):";
        cout <<*(ptr + i) << endl;
    }
    return 0;
}
```

Also, written as

```
ptr = arr;
```

0	10	1000	← ptr
1	20	1002	
2	30	1004	
3	40	1006	
4	50	1008	

Output:

*(ptr + 0) : 10

*(ptr + 1) : 20

*(ptr + 2) : 30

*(ptr + 3) : 40

*(ptr + 4) : 50

Pointers and objects

- Just like pointers to normal variables and functions, we can have **pointers to class members** (variables and methods).

```
class ABC
{
  public:
  int a=50;
};
int main()
{
  ABC ob1;
  ABC *ptr;
  ptr = &ob1;
  cout << ob1.a;
  cout << ptr->a; // Accessing member with pointer
}
```

When accessing members of a class given a pointer to an object, use the **arrow (–>) operator** instead of the dot operator.

Pointers and objects (Cont…)

```
class demo{
   int i;
   public:
     demo(int x)
     {
       i=x;
     }
     int getdata(){
       return i;}
};
int main()
{
   demo d(55),*ptr;
   ptr=&d;
   cout<<ptr->getdata();
}
```

Pointers and objects (Cont…)

```
class demo{
   int i;
   public:
     demo(int x){
       i=x; }
     int getdata(){
       return i;}
};
int main()
{
   demo d[3]={55,66,77};
   demo *ptr=d; //similar to *ptr=&d[0]
   for(int i=0;i<3;i++)
   {
      cout<<ptr->getdata()<<endl;
      ptr++;
   }
}
```

- When a pointer incremented it points to next element of its type.
- An integer pointer will point to the next integer.
- The same is true for pointer to objects

`this` pointer

this pointer

```
class Test
{
   int mark;
   float spi;
   public:
    void SetData(){
      mark = 70;
      spi = 6.5;
  this->mark = 70;
   this->spi = 6.5 ;
    }
    void DisplayData(){
      cout << "Mark= "<<mark;
      cout << "spi= "<<spi;
    }
} ;
int main()
{
   Test o1;
   o1.SetData();
   o1.DisplayData();
}
```

- Within member function, the members can be accessed directly, without any object or class qualification.
- But implicitly members are being accessed using **this** pointer

- When a member function is called, it automatically passes a **pointer** to invoking object.

this pointer(Cont…)

- ‘**this**’ pointer represent an object that invoke or call a member function.
- It will point to the object for which member function is called.
- It is automatically passed to a member function when it is called.
- It is also called as implicit argument to all member function.

Note:

✓ Friend functions can not be accessed using **this** pointer, because friends are not members of a class.

✓ Only member functions have a **this** pointer.

✓ A **static** member function does not have **this** pointer.

this pointer (Cont…)

```
class sample
{
  int a,b;
  public:
    void input(int a,int b){
      this->a = a + b;
      this->b = a - b;
    }
    void output(){
     cout<<"a = "<<a;
     cout<<"b = "<<b;
    }
};
int main()
{
    sample ob1;
    int a=5,b=8;
    ob1.input(a,b);
    ob1.output();
}
```

this pointer is used when local variable's name is same as member's name.

this pointer (Cont…)

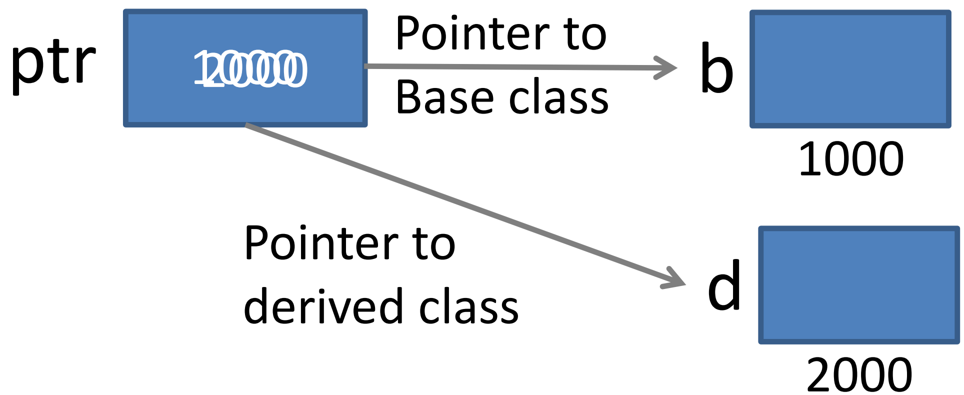
```
class Test
{
  int x; int y;
public:
  Test& setX(int a) { x = a; return *this; }
  Test& setY(int b) { y = b; return *this; }
  void print() {
  cout << "x = " << x ;
  cout << " y = " << y;
  }
};

int main()
{
  Test obj1;
  obj1.setX(10).setY(20);
  obj1.print();
}
```

this pointer is used to return reference to the calling object

Pointer to Derived Class

Pointer to derived class

- We can use pointers not only to the base objects but also to the objects of derived classes.
- A single pointer variable of **base type** can be made to point to objects belonging to **base** as well as **derived classes**.



For example:

```
Base *ptr;
Base b;
Derived d;
ptr = &b; //points to base object
      //We can make ptr to point to the object d as follows
ptr = &d; //base pointer point to derived object
```

```
class Base {
public:
void showBase(){
  cout << "Base\n"; }
};
class Derv1 : public Base {
public:
void showDerived(){
  cout << "Derv1\n"; }
};
int main(){
   Derv1 dv1;
   Base* ptr;
   ptr = &dv1;
   ptr->showBase();
   ptr->showDerived(); //error
   ((Derv1 *)ptr)->show();
}
```

Derived type casted to base type

Base pointer explicitly casted into derived type

Output:
Base
Derv1

Pointer to derived class (Cont…)

- We can access those members of derived class which are **inherited from base class** by **base class** pointer.
- But we cannot access original member of derived class which are **not inherited** from base class using base class pointer.

- We can access original member of **derived class** using pointer of **derived class**.

Program

```
class base
{
   public:
   int b;
   void show()
   {
     cout<<"\nb="<<b;
   }
};
class derived : public base
{
   public:
   int d;
   void show()
   {
     cout<<"\n b="<<b<<"\n d="<<d;
   }
};
```

Program (Cont...)

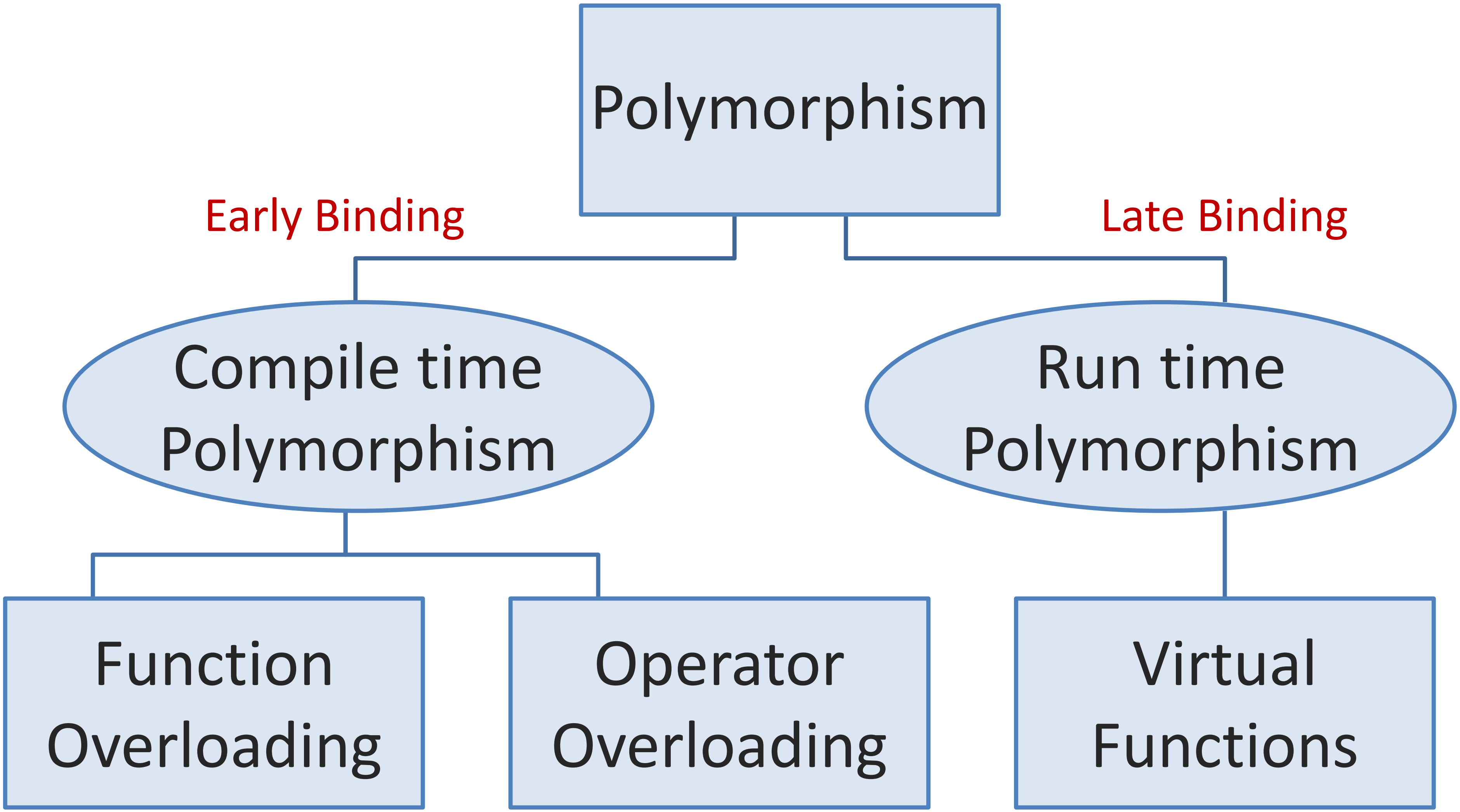
```
int main(){
base B1;
derived D1;
base *bptr;
bptr=&B1;
cout<<"\nBase class pointer assign address of base class object";
bptr->b=100;
bptr->show();
bptr=&D1;
bptr->b=200;
cout<<"\nBase class pointer assign address of derived class object";
bptr->show();
derived *dptr;
dptr=&D1;
cout<<"\nDerived class pointer assign address of derived class
object";
dptr->d=300;
dptr->show();
}
```

Virtual Function

Virtual Function

- A **virtual function** is a member function that is declared within a base class and redefined by a derived class.
- To create a **virtual function**, precede the function's declaration in the base class with the keyword `virtual`.

Compile time and Run time Polymorphism



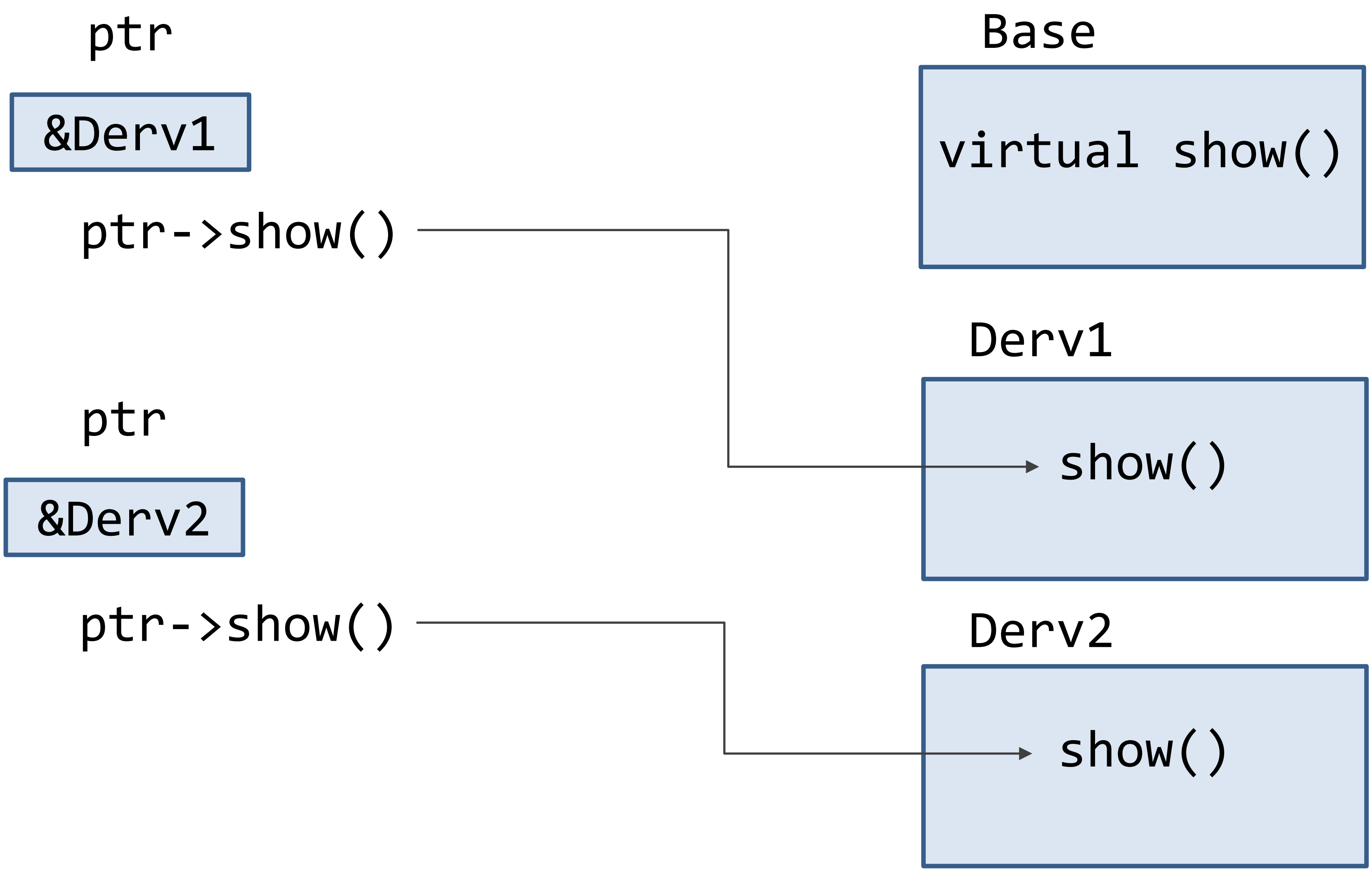
Virtual Function

- When **virtual function** accessed "normally," it behave just like any other type of class member function.
- But when it is accessed via a **pointer** it supports **run time polymorphism**.
- Base class and derived class have **same function name** and base class pointer is assigned address of derived class object then also pointer will execute base class function.
- After making virtual function, the compiler will determine which function to execute at run time on the basis of assigned address to pointer of base class.

```cpp
class Base {
public:
virtual void show(){
  cout << "Base\n"; }
};
class Derv1 : public Base {
public:
void show(){
  cout << "Derv1\n"; }
};
class Derv2 : public Base {
public:
void show(){
  cout << "Derv2\n"; }
};
```

```cpp
int main()
{
   Derv1 dv1;
   Derv2 dv2;
   Base* ptr;
   ptr = &dv1;
   ptr->show();
   ptr = &dv2;
   ptr->show();
}
```

Output:
Derv1
Derv2



- When a function is made virtual, C++ determines which function to use at **run time** based on the type of object pointed by the base pointer, rather than the type of pointer .

Rules for virtual base function

1. The virtual functions must be member of any 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.

Pure Virtual Function

Pure Virtual Function

- A pure virtual function is virtual function that has no **definition** within the base class.

```
class shape
{
 public:
   virtual void draw()
    {
        ...
    }
 };
```

```
class rectangle
{
 public:
    void draw()
    {
        ...
    }
 };
```

```
class square
{
 public:
    void draw()
    {
        ...
    }
 };
```

```
class circle
{
 public:
    void draw()
    {
        ...
    }
 };
```

Pure virtual functions

- A **pure virtual function** means ‘do nothing’ function.
- We can say empty function. A **pure virtual function** has no definition relative to the base class.
- Programmers have to redefine **pure virtual function** in derived class, because it has no definition in base class.
- A class containing **pure virtual function** cannot be used to create any direct objects of its own.
- This type of class is also called as **abstract class**.

Syntax:

```
virtual void display() = 0;
            OR
virtual void display() {}
```

Program

```
class Shape{
    protected:
        float x;
    public:
        void getData(){cin >> x;}
        virtual float calculateArea() = 0;
};
class Square : public Shape
{
    public:
        float calculateArea()
        {   return x*x;   }
};
class Circle : public Shape
{
    public:
        float calculateArea()
        { return 3.14*x*x; }
};
```

This is called abstract class

Program

```cpp
int main()
{
    Square s;
    Circle c;
    cout << "Enter length to calculate the area of a square:";
    s.getData();
    cout<<"Area of square: " << s.calculateArea();
    cout<<"Enter radius to calculate the area of a circle: ";
    c.getData();
    cout << "Area of circle: " << c.calculateArea();

}
```

Output:
Enter length to calculate the area of a square: 10
Area of square: 100
Enter radius to calculate the area of a circle: 9
Area of circle: 254.34

Abstract Class

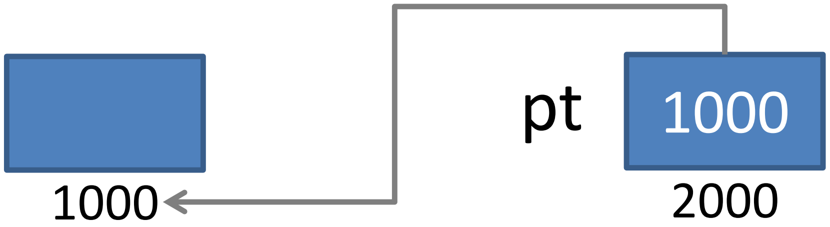
- A class that contains at least one **pure virtual function** is called **abstract** class.
- You can not create objects of an **abstract class**, you can create **pointers and references** to an abstract class.

new and **delete** Operator

Memory allocation using **new** operator

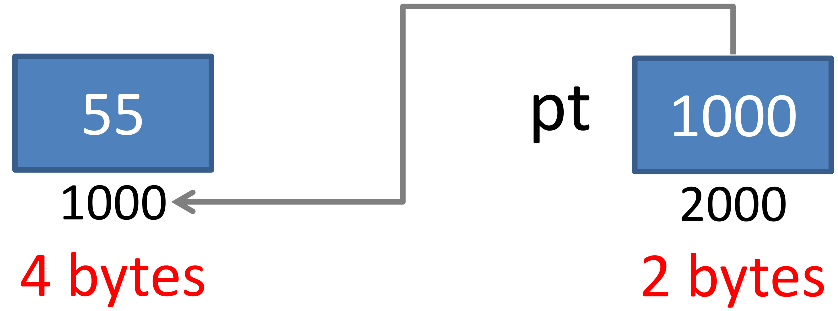
- new is used to dynamically allocate memory
- new finds a block of the correct size and returns the address of the block.
- Assign this address to a pointer.

```
int *pt = new int;
```



- new int part tells the program you want some new storage of size int.
- Then it finds the memory and returns the address.
- Next, assign the address to *pt.
- Now pt is the address and *pt is the value stored there.

Program




```
int main ()
{
    float *pt = new float;
    *pt = 55;
    cout<<"value="<<*pt;
    cout<<"\naddress="<<pt;
    cout<<"\nsize="<<sizeof (*pt);
    cout<<"\nsize ptr="<<sizeof pt;
}
```

value=55

address=1000

size=4

size=2

Free memory using **delete** operator

- `delete` operator frees memory allocated by `new`.

```
int * ps = new int; // allocate memory with new
. . . // use the memory
delete ps; // free memory with delete when done
```

- it doesn’t remove the pointer ps itself. You can reuse ps, to point to another new allocation.

```
int * ps = new int; // valid
delete ps; // valid
delete ps; // not valid now
int jugs = 5; // valid
int * pi = &jugs; // valid
delete pi;//not allowed, memory not allocated by new
```

Thank You