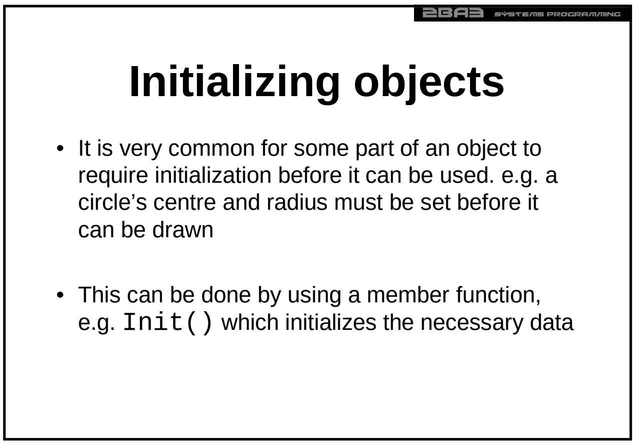
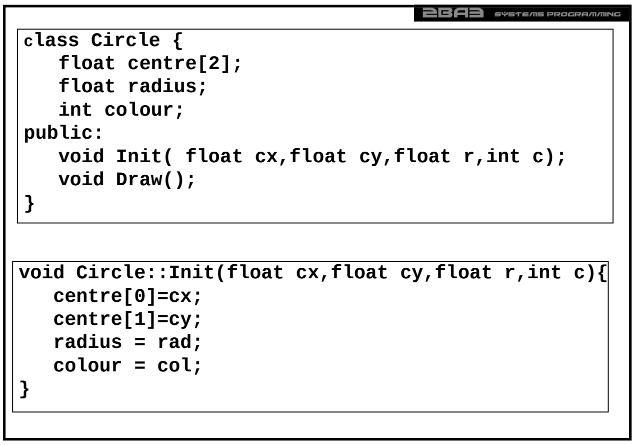


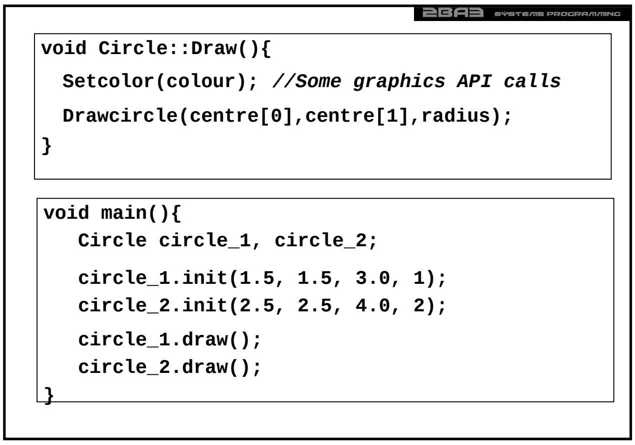
Initializing objects

- It is very common for some part of an object to require initialization before it can be used. e.g. a circle's centre and radius must be set before it can be drawn
- This can be done by using a member function, e.g. `Init()` which initializes the necessary data



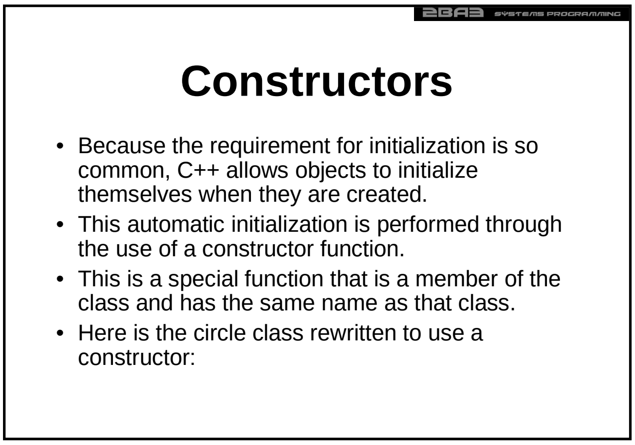
```
class Circle {
   float centre[2];
   float radius;
   int colour;
public:
   void Init( float cx,float cy,float r,int c);
   void Draw();
}
```

```
void Circle::Init(float cx,float cy,float r,int c){
   centre[0]=cx;
   centre[1]=cy;
   radius = rad;
   colour = col;
}
```



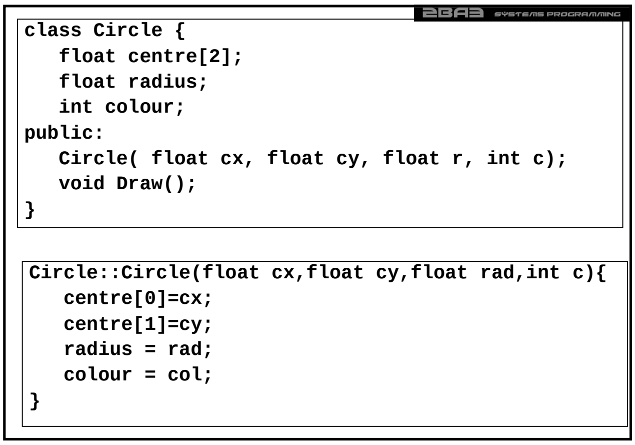
```
void Circle::Draw(){
  Setcolor(colour); //Some graphics API calls
  Drawcircle(centre[0],centre[1],radius);
}
```

```
void main(){
   Circle circle_1, circle_2;

   circle_1.init(1.5, 1.5, 3.0, 1);
   circle_2.init(2.5, 2.5, 4.0, 2);

   circle_1.draw();
   circle_2.draw();
}
```



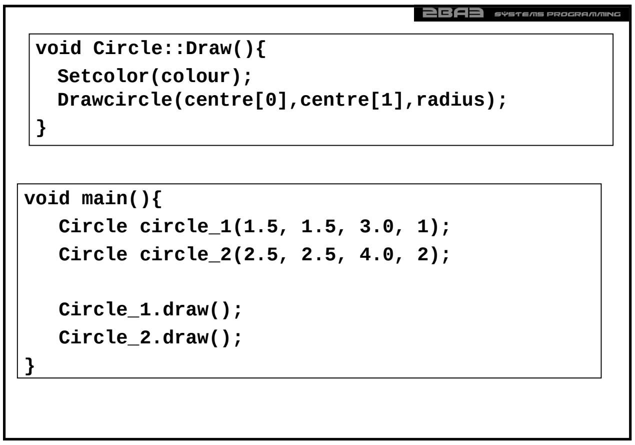
Constructors

- Because the requirement for initialization is so common, C++ allows objects to initialize themselves when they are created.
- This automatic initialization is performed through the use of a constructor function.
- This is a special function that is a member of the class and has the same name as that class.
- Here is the circle class rewritten to use a constructor:



```
class Circle {
   float centre[2];
   float radius;
   int colour;
public:
   Circle( float cx, float cy, float r, int c);
   void Draw();
}
```

```
Circle::Circle(float cx,float cy,float rad,int c){
   centre[0]=cx;
   centre[1]=cy;
   radius = rad;
   colour = col;
}
```



```
void Circle::Draw(){
  Setcolor(colour);
  Drawcircle(centre[0],centre[1],radius);
}
```

```
void main(){
   Circle circle_1(1.5, 1.5, 3.0, 1);
   Circle circle_2(2.5, 2.5, 4.0, 2);

   Circle_1.draw();
   Circle_2.draw();
}
```

Constructors

- An object’s constructor is called when the object is created, i.e. it is called when the object’s declaration is executed.
- Some C++ variable declarations are passive, and resolved mainly at compile time, e.g. int x;
- However, C++ declarations can also be active statements that are actually executed at run time
- This is to allow the declaration of a object to call a constructor, thus making it an executable statement.
- Constructor functions do not have return values.

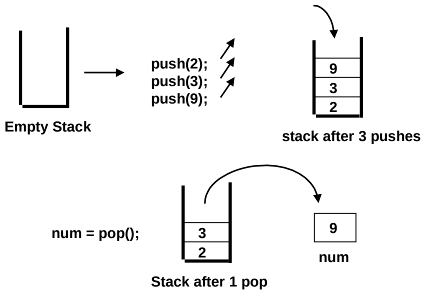
Destructors

- In many cases, and object will need to perform some action or actions when it is destroyed.
- For example, an object may allocate memory for some purpose, and must then deallocate it on destruction
- Local objects are created when their function is entered, and destroyed when the function is left, Global objects are destroyed when the program terminates
- The destructor has the same name as the constructor, but is preceded by a ~
- Destructor functions also have no return value
- e.g. If we needed a destructor for our circle class, (which we don’t) it would be defined as:

```
~circle();
```

Example: Stacks

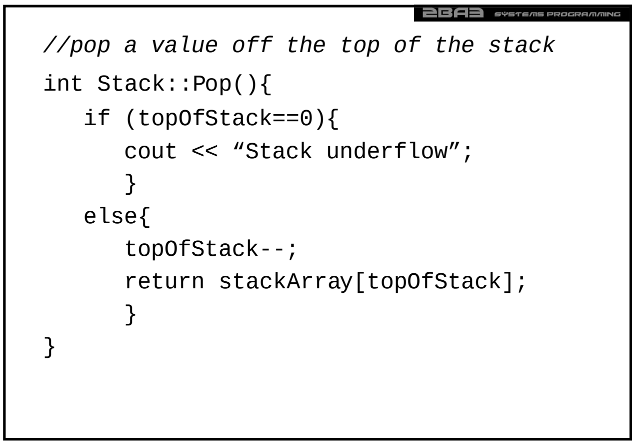
Stacks are a **LIFO** data structure, i.e. **L**ast **I**n **F**irst **O**ut



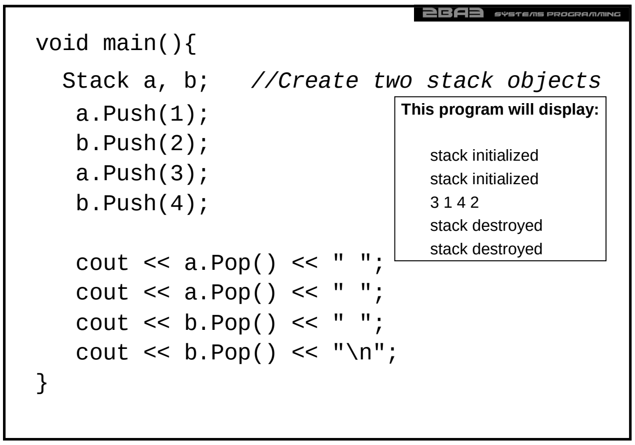
```
#define SIZE 100

//Class definition
class Stack {
   int stackArray[SIZE];
   int topOfStack;
public:
   Stack();          //constructor
   ~Stack();         //destructor
   void Push(int i);
   int Pop();
};
```

```
//The stack’s constructor function
Stack::Stack(){
   topOfStack = 0;
   cout << "stack initialized.\n";
}

//The stack’s destructor function
Stack::~Stack(){
cout << "stack destroyed.\n";
}
```

```
//push a value onto the top of the stack
void Stack::Push(int i){
if (topOfStack==SIZE){
   cout << “Stack is full.\n”;
   }
else {
   stackArray[topOfStack] = i;
   topOfStack++;
   }
```



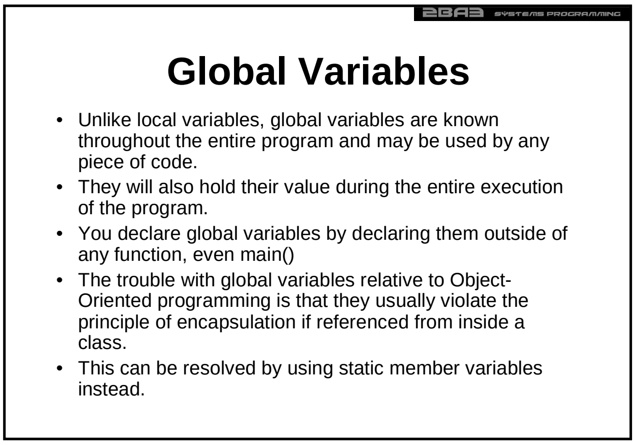
```
//pop a value off the top of the stack
int Stack::Pop(){
   if (topOfStack==0){
      cout << “Stack underflow”;
      }
   else{
      topOfStack--;
      return stackArray[topOfStack];
      }
}
```



```
void main(){
  Stack a, b;   //Create two stack objects
   a.Push(1);
   b.Push(2);
   a.Push(3);
   b.Push(4);

   cout << a.Pop() << " ";
   cout << a.Pop() << " ";
   cout << b.Pop() << " ";
   cout << b.Pop() << "\n";
}
```

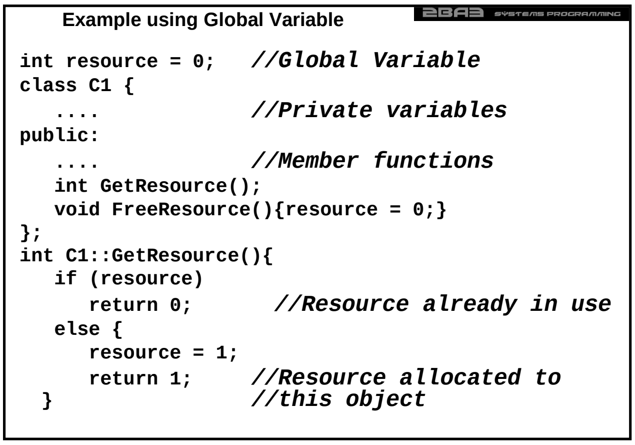
This program will display:

```
stack initialized
stack initialized
3 1 4 2
stack destroyed
stack destroyed
```



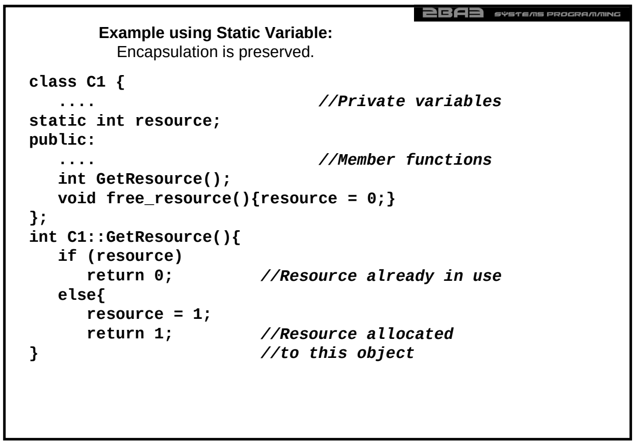
Global Variables

- Unlike local variables, global variables are known throughout the entire program and may be used by any piece of code.
- They will also hold their value during the entire execution of the program.
- You declare global variables by declaring them outside of any function, even main()
- The trouble with global variables relative to Object-Oriented programming is that they usually violate the principle of encapsulation if referenced from inside a class.
- This can be resolved by using static member variables instead.



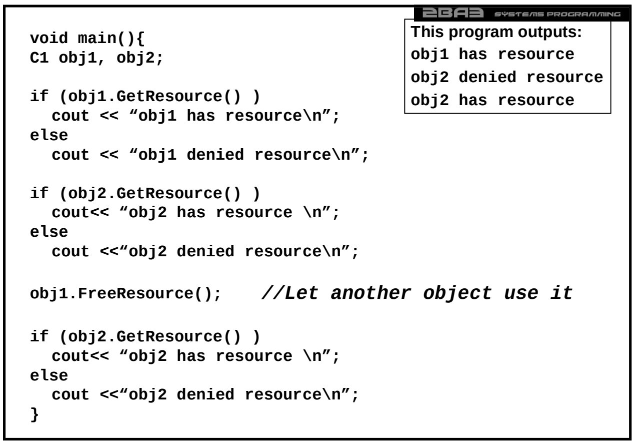
Example using Global Variable

```
int resource = 0;   //Global Variable
class C1 {
   ....             //Private variables
public:
   ....             //Member functions
   int GetResource();
   void FreeResource(){resource = 0;}
};
int C1::GetResource(){
   if (resource)
      return 0;      //Resource already in use
   else {
      resource = 1;
      return 1;     //Resource allocated to
  }                 //this object
```



Example using Static Variable:
Encapsulation is preserved.

```
class C1 {
   ....                        //Private variables
static int resource;
public:
   ....                        //Member functions
   int GetResource();
   void free_resource(){resource = 0;}
};
int C1::GetResource(){
   if (resource)
      return 0;          //Resource already in use
   else{
      resource = 1;
      return 1;          //Resource allocated
}                        //to this object
```



```
void main(){
C1 obj1, obj2;

if (obj1.GetResource() )
  cout << “obj1 has resource\n”;
else
  cout << “obj1 denied resource\n”;

if (obj2.GetResource() )
  cout<< “obj2 has resource \n”;
else
  cout <<“obj2 denied resource\n”;

obj1.FreeResource();    //Let another object use it

if (obj2.GetResource() )
  cout<< “obj2 has resource \n”;
else
  cout <<“obj2 denied resource\n”;
}
```

This program outputs:

```
obj1 has resource
obj2 denied resource
obj2 has resource
```

Static Data Members

- When you precede a member variable’s declaration with static, you are telling the compiler that:
  - only one copy of that variable will exist
  - all objects of the class will share that variable
- Unlike regular data members, individual copies of a static member variable are not made for each obect.
- No matter how many objects of a class are created, only one copy of a static data member exists.
- All static variables are initialized to zero (or 0.0 in the case of floating point numbers) when the first object is created. Other initializations are complicated,

```
#include <iostream.h>
class Shared {
   static int a;
   int b;
public:
   void Set(int i, int j) {a=i; b=j;}
   void Show();
}
void Shared::Show(){
   cout << "This is static a: " <<a;
   cout << "This is non-static b: " <<b;
   cout << "\n";
}
```

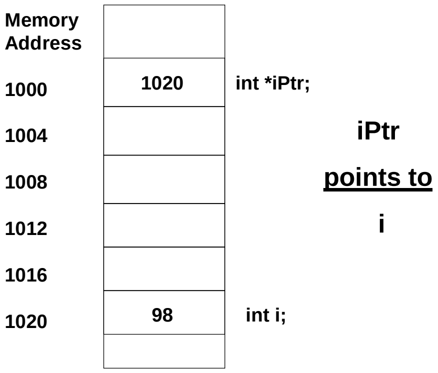
This program displays:

```
This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

```
void main(){
   Shared x, y;

   x.Set(1, 1);       //Set a to 1
   x.Show();

   y.Set(2,2);        //Change a to 2
   y.Show();

   x.Show();          //a has been changed
}                     //for both objects
```

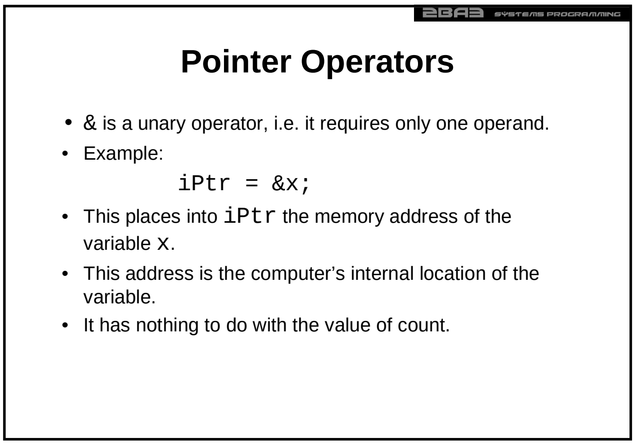
Pointers

- A pointer is a variable that holds a memory address
- Most commonly, this address is the location of another variable in memory.
- If one variable contains the address of another variable, the first variable is said to point to the second.
- If a variable is going to hold a pointer, it must be declared as such.
- A pointer declaration consists of a base type, and *, and the variable name. i.e: `type *name;`
- The type may be any valid type or class name, and name is the name of the pointer variable.



Pointer Operators

- The base type of the pointer defines what type of variables the pointer can point to.
- Technically, any type of pointer an point anywhere in memory, but pointer arithmetic is done relative to its base type, so it is important to declare the pointer correctly.
- There are two special pointer operators: * and &
  - & returns the memory address of a variable
  - * is its complement, i.e. it returns the contents of the memory located at the given address.

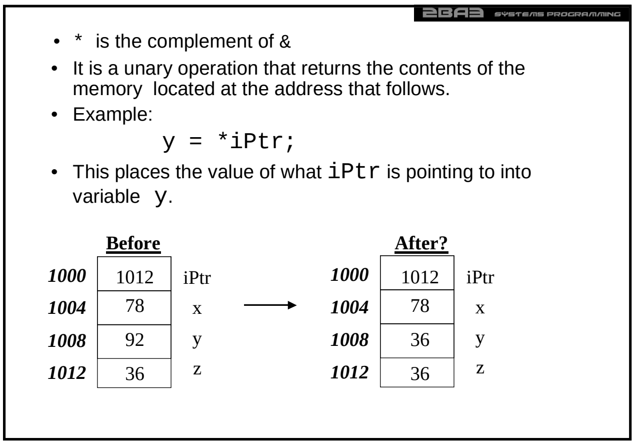


Pointer Operators

- & is a unary operator, i.e. it requires only one operand.
- Example:

```
iPtr = &x;
```

- This places into `iPtr` the memory address of the variable `x`.
- This address is the computer's internal location of the variable.
- It has nothing to do with the value of count.

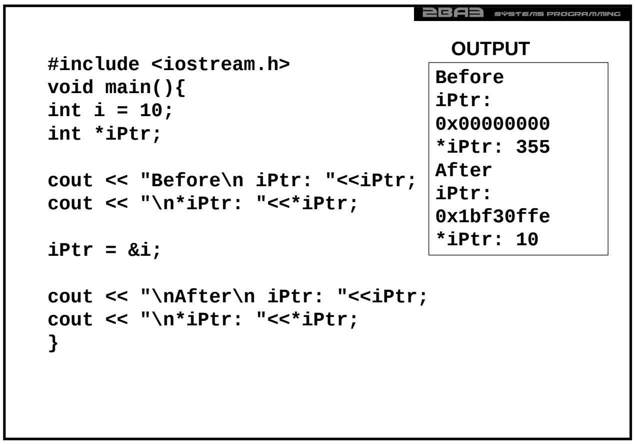


- `*` is the complement of &
- It is a unary operation that returns the contents of the memory located at the address that follows.
- Example:

```
y = *iPtr;
```

- This places the value of what `iPtr` is pointing to into variable `y`.

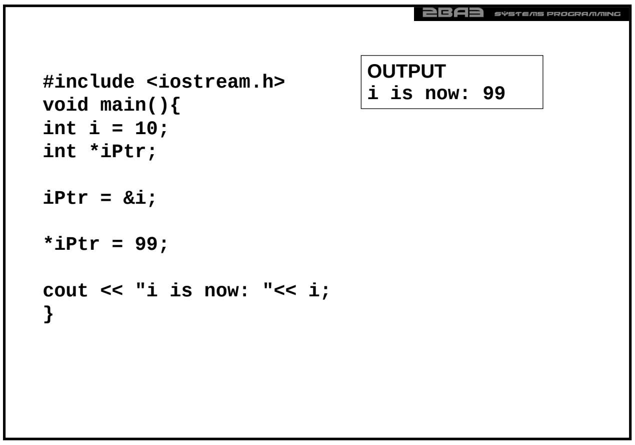
Before				After?		
1000	1012	iPtr	→	***1000***	1012	iPtr
1004	78	x		***1004***	78	x
1008	92	y		***1008***	36	y
1012	36	z		***1012***	36	z



```
#include <iostream.h>
void main(){
int i = 10;
int *iPtr;

cout << "Before\n iPtr: "<<iPtr;
cout << "\n*iPtr: "<<*iPtr;

iPtr = &i;

cout << "\nAfter\n iPtr: "<<iPtr;
cout << "\n*iPtr: "<<*iPtr;
}
```

OUTPUT

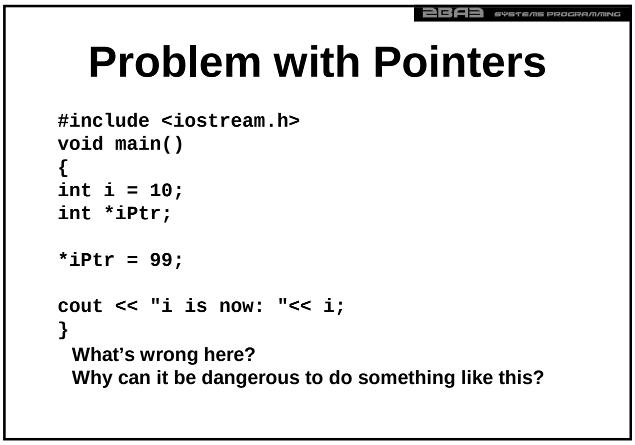
```
Before
iPtr:
0x00000000
*iPtr: 355
After
iPtr:
0x1bf30ffe
*iPtr: 10
```



```
#include <iostream.h>
void main(){
int i = 10;
int *iPtr;

iPtr = &i;

*iPtr = 99;

cout << "i is now: "<< i;
}
```

OUTPUT

```
i is now: 99
```



Problem with Pointers

```
#include <iostream.h>
void main()
{
int i = 10;
int *iPtr;

*iPtr = 99;

cout << "i is now: "<< i;
}
```

What's wrong here?
Why can it be dangerous to do something like this?