

More on Classes and Object

- This Pointer
- Array of Objects
- Dynamic allocation of objects
- De-allocation of memory
- Destructor
- C++ Functions with the default argument

What is “this” Pointer

What is “*this*” pointer?

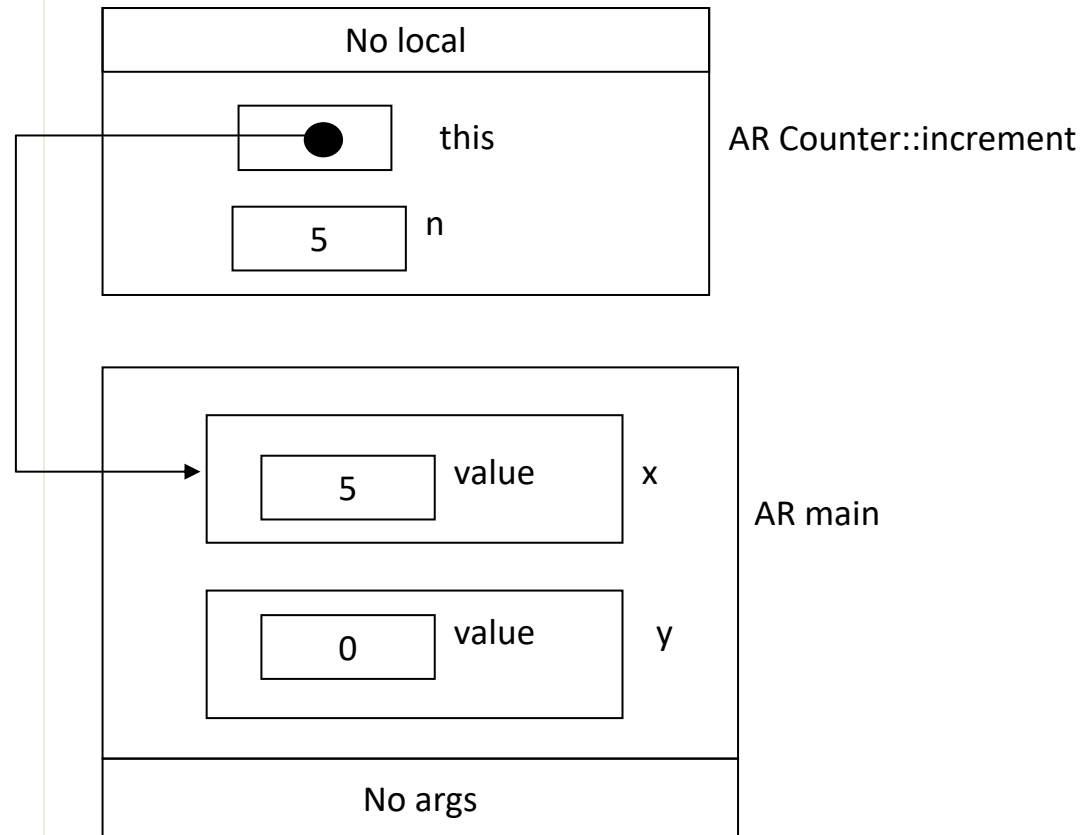
- Each class object maintains its own copy of the class data members, but there is only one copy of each member function in the program.
 - How does a member function know a data member within the function belongs to which object?
 - The answer is : this pointer
- Every member function except static members has a hidden pointer, as its first parameter, pointing to the object that invokes the member function.

Draw AR Diagram for Point ONE

```
class Counter {
public:
    Counter() :value(0) {}
    void increment(int n);
private:
    int value;
};

void Counter::increment(int n)
{
    value += n;
    // it is in fact: this->value +=n;
    // Point ONE
}

int main(void)
{
    Counter x;
    Counter y;
    x.increment(5);
}
```



Array of Objects

Array of objects

- To declare an array of objects, the class definition must have a default (No arg) constructor.

```
#define SIZE 20
```

```
class Car
```

```
{
```

```
    public:
```

```
    Car (const char* m, int y, double p);
```

```
    Car ();
```



Default (no arg) constructor

```
    const char* getMake() const;
```

```
    void setMake(const char* n);
```

```
    ... // other member functions
```

```
    private:
```

```
    char make[SIZE];
```

```
    int year ;
```


```
    double price;
```

```
};
```

// Constructor

```
Car::Car(const char* m, int y, double p) :  
    year(y), price (p)  
{  
    assert (strlen (m) < SIZE);  
    strcpy (make, m);  
}
```

// Constructor (No Argument)



```
Car::Car( ) : year(0), price (0)  
{  
    for (int j =0; j < SIZE; j++)  
        make[j] = '\0';  
}
```

```
const char* Car::getMake() const
```

```
{  
    return make;  
}
```

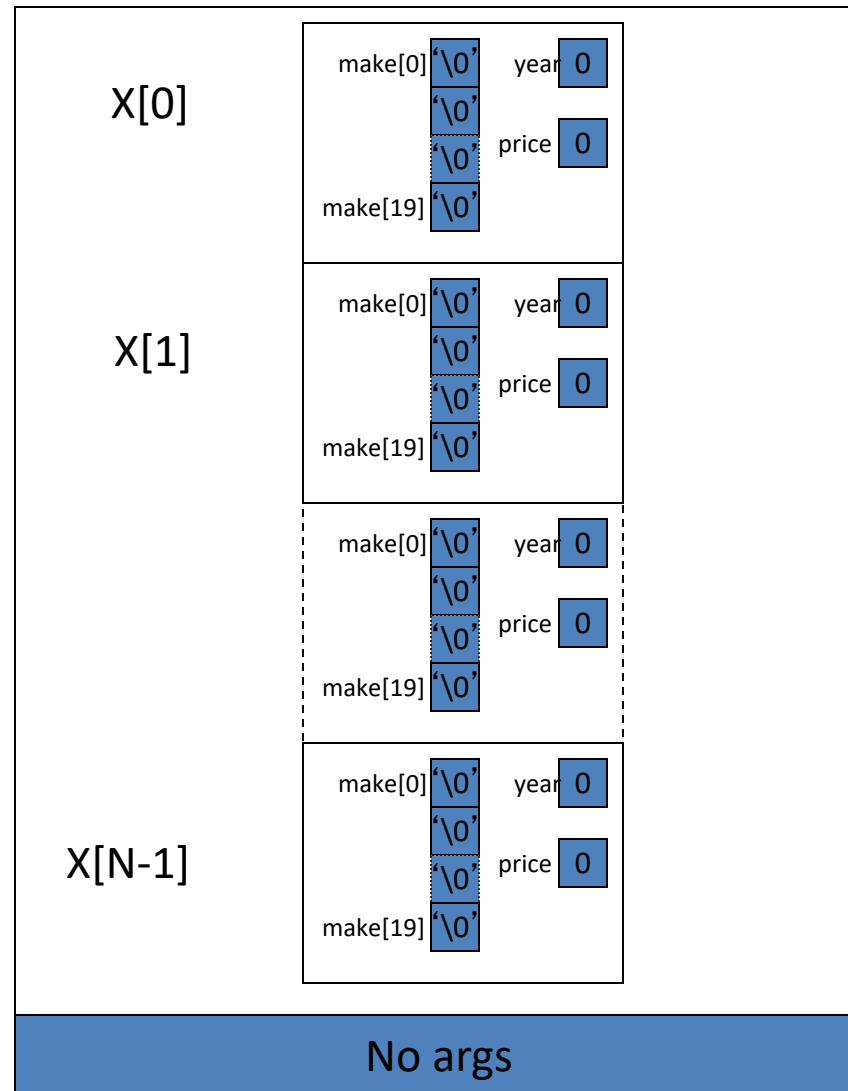
```
void Car::setMake(const char* m)
```

```
{  
    assert (strlen (m) < SIZE);  
    strcpy(make, m);  
    // POINT ONE  
}
```

Declaration of Array of Objects

```
int main()
{
    Car x[N];
    // Point ONE
}
```

It means that the constructor of class Car will be called N times



AR main

Access to the Data Members in An Array Element

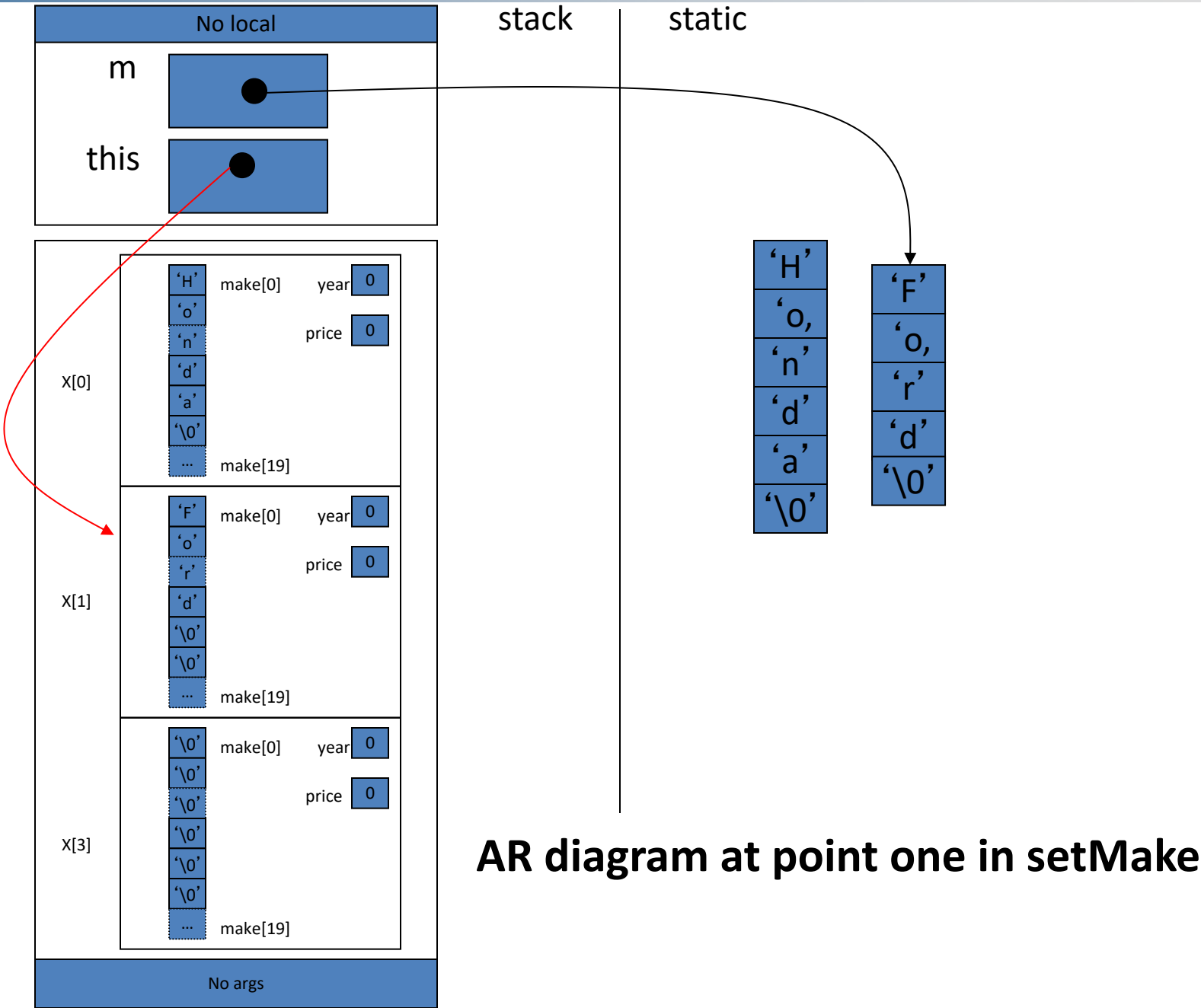
- Access to the private data members in an element of an array, follows the rules that applies to ordinary object. See the following example:

```
int main(void)
{
    Car x[3];
    x[0].setMake("Honda");
    x[1].setMake("Ford");
    cout << x[0].getMake();    // displays: Honda
    cout << x[1].getMake()[0]; // displays: F
}
```

- Draw the AR diagram at point ONE, for the second call to the function setMake().

AR
setMake

AR
main



Arrays and Array Elements as Function Arguments

- Arrays of objects are passed to a function similar to other basic data type.

```
int main(void)
{
    Car x[3];

    x[0].setMake("Honda");

    x[1].setMake("Ford");

    displayAll(x , 2);

    swap (&x[0], &x[1]);
}
```

Arrays and Array Elements as Function Arguments

```
void displayAll (Car x[], int n)
{
    for(int j=0; j<n; j++)
        cout << x[j].getMake();
}
```

```
void swap (Car *x, Car *y)
{
    Car temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Dynamic Allocation and De-allocation of Memory

Dynamic Allocation of Memory

- Creates a block of memory at the run time.
- This type of allocation can be used for any legal C++ built-in or user-defined data type.
- *new* is a C++ operator for dynamic allocation of memory on the free storage (heap).
- *delete* is a C++ operator for deallocation of the memory from free storage (heap).

Dynamic Allocation of Memory

- Example of allocation of memory on the free storage (heap) for an array of integers with 5 elements

```
int main()
```

```
    int *array;
```

```
    array = new int [5] ;
```

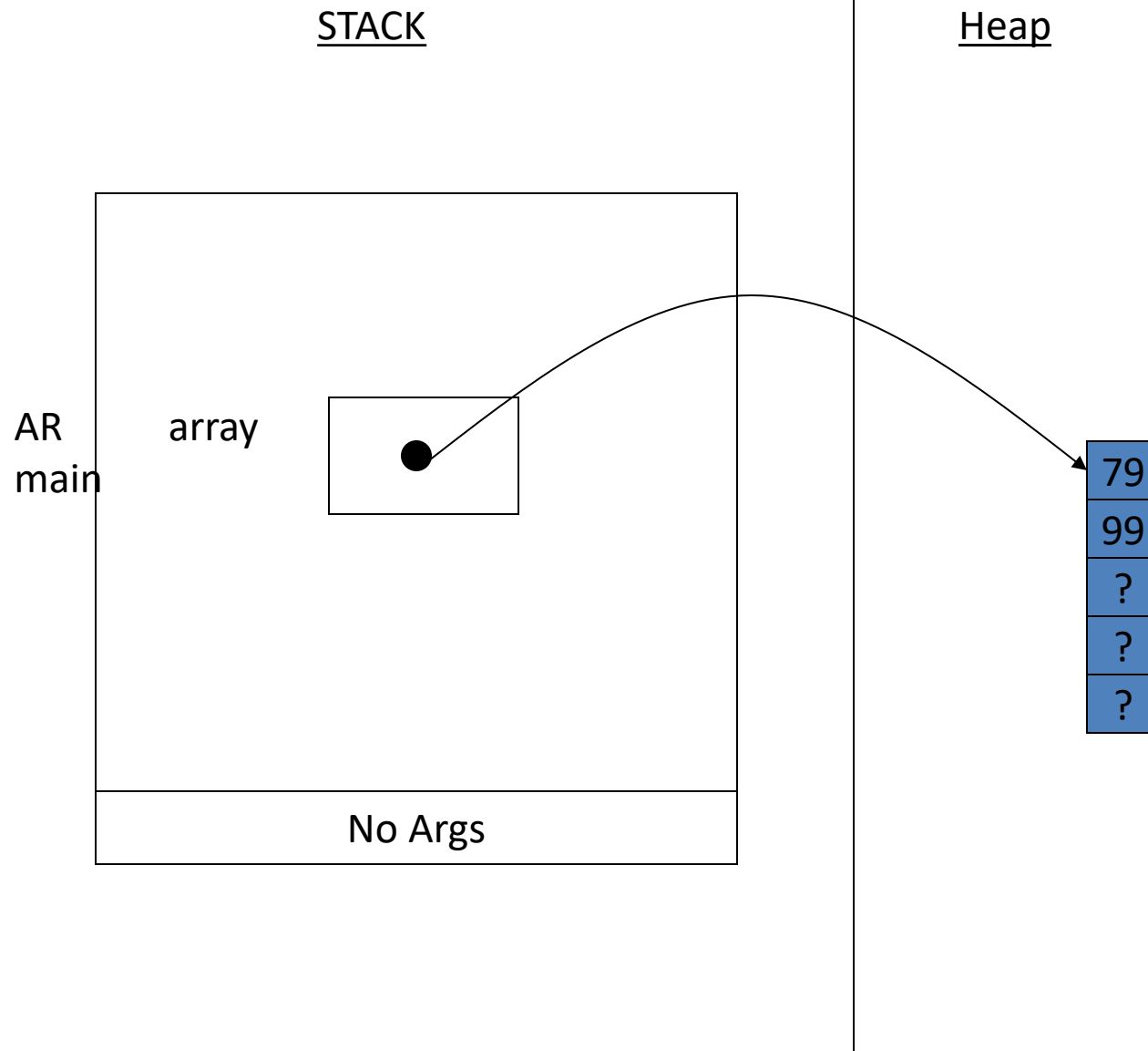
```
    array[0] = 79;
```

```
    array[1] =99;
```

```
    ...
```

```
    }
```

AR Diagram



Delete allocated memory from heap

- Use delete operator to delete allocated memory by new.

`delete [] array;`

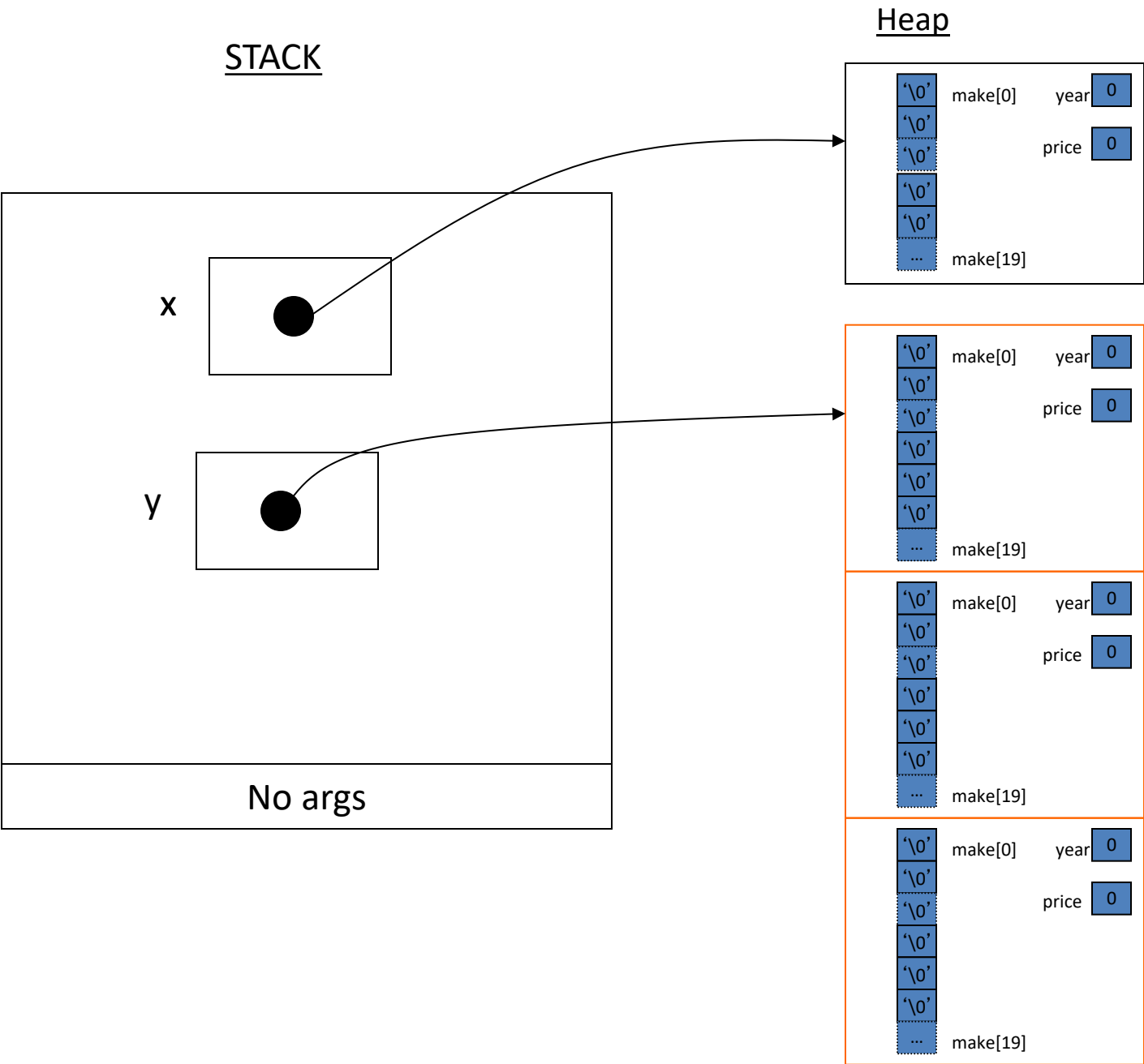
- Use [] whenever referring to an array of built-in or user defined types.
- Do not use *delete* for variables that their space is not allocated by new.
- Do not use *delete* if space is not allocated or if space has been already deallocated.

Dynamic allocation of class objects

- Class objects can also be allocated dynamically:

```
int main()
{
    Car *x;
    Car *y;
    x = new Car;
    y = new Car[3 ];
    ...
}
```

- Dynamically allocated spaces will not be deallocated automatically by the compiler. Therefore, a dynamically allocated space, should be to deallocated by *delete* operator:
 - Deallocation of a single objects:
delete x;
 - Deallocation of an array of objects:
delete [] y;



What is a destructor and when do we need it?

Destructor – Automatic Deletion of Object

- Destructor is responsible for resetting the data member values or de-allocation of memory , if necessary.
- A destructor doesn't return a value and does not receive any parameter
- Destructor will be called automatically when an object goes out of scope. It can be also called explicitly
- Destructor can NOT be overloaded

Class Person Example

```
// File: person.h
```

```
#ifndef PERSON
```

```
#define PERSON
```

```
Class Person{
```

```
    private:
```

```
        int age;
```

```
        char* name;
```

```
    public:
```

```
        Person(const char* n, int a);
```

```
        ~Person();
```

```
        void setAge(int a) { age = a;}
```

```
        int getAge() const{return age;}
```

```
        const char* getName ()const;
```

```
};
```

```
#endif
```

Constructor and Destructor Definition

```
// File: person.cpp
// Definition of the Constructor
// Constructor's name is the same as the class name
#include "person.h"
#include <string.h>

Person::Person(const char* n, int a)
{
    age = a;
    name = new char [strlen(n)+1];
    assert (name != 0);
    strcpy(name,n);
}
```

```
// Definition of the Destructor
// Destructors name is also same as class name
// with the exception of having tilde character
// '~' after the scope resolution operator

Person::~~Person()
{
    delete [] name;
    name = NULL;
}
```

What is a Default Argument

C++ functions - default arguments

- A global or member function argument in C++ may have a default value:

```
char fun (int h = 24, int w = 80, char ch = ' ');  
    fun();  
    fun(2);  
    fun(2, 8);  
    fun(2, 8, 'B' );
```

- The right most not initialized argument must be supplied with a default initializer before any argument to its left may be supplied.
- You should initialize arguments either at the function prototype or function definition, not both.

Another Example

```
Person::Person(const char* n = NULL, int a = 0)
{
    age = a;
    name = new char [strlen(n)+1];
    assert (n != 0);
    strcpy(name, n);
}
```

Member Functions with Reference Return Type

Functions with Reference Return Type

- C++ function can also return a reference type
- Examples:

```
class MyString {  
    public:  
        MyString();  
        MyString(const char *s);  
        const char& at(int i) const;  
        char& at (int i);  
        ...  
        ...  
    private:  
        char * storageM ;  
        int length;  
};
```

```
const char& MyString::at(int i) const  
{  
    assert(i >= 0 && i < length);  
    return storageM[i];  
}  
  
char& MyString::at(int i)  
{  
    assert(i >= 0 && i < length);  
    return storageM[i];  
}
```

- What do these at functions do, what are their differences, and why do we need two of them?

Member Functions with const Return Type

Member Functions with `const` Return Type

- Sometimes, it is necessary to protect the values returned from member functions. In particular, if you return a pointer or reference to a member variable, the user has the ability to change the value of your member variable, even if it is private. Lets see what may go wrong with the following example:

```
class Student
{
public:
    Student();
    Student(const char* &name, const int id);
    char* get_name() const;
    // other member function prototypes here
private:
    char nameM[50];
    int idM;
};
```

Protecting Data Members

```
Student::Student()
```

```
{  
    strcpy(nameM, "None");  
    idM = 0;  
}
```

```
Student::Student(const char* name, const int id)
```

```
{  
    strcpy(nameM, name);  
    idM = id;  
}
```

Protecting Data Members

```
char* Student::get_name() const
{
    return nameM; // returns address of "s_name"
}
```

- Assuming that the above compiles, consider the following (legal) code segment:

```
char name[] = "Jane";
Student One(name, 123456);

char* trouble = One.get_name();

trouble[0] = 'P'; // One.nameM is now "Pane"
```

- What is wrong or bad about this code?
 - This code defeats the purpose of information hiding
 - Even const member function didn't help
 - What could the solution be?

Protecting Data Members

- To get around this problem, we must specify the return type to be `const`. In the above example, we would write:

```
const char* Student::get_name() const
{
    return nameM;
}
```

- The first `const`, (in red) prevents the user from modifying the return value.
- The second `const` prevents us from modifying any member variables inside the `get_name()` function.
- If your function returns a `const` pointer, you must also make the object in your calling function a `const`. For instance you will need to replace:

```
char* trouble = One.get_name();
```

- with

```
const char* trouble = One.get_name();
```

- Accordingly the prototype of the function `get_name` in the `Student` class must be changed to:

```
const char* Student::get_name() const;
```

Inline Member Functions and Inline Modifier

Inline Functions

- If the function body is very short, the overhead of invoking a function call may be too high for a trivial amount of work done.
- In these situations we can use *inline functions*, by defining it within the body of the class:

```
class Counter
{
    public:
        ...

        int get_value() { return value; }
        void increment(int n) { value += n; }
    private:
        int value;
};
```

- Or, you make it inline by preceding the entire implementation by the `inline` keyword (the prototype is the same as before). For example:

```
inline void decrement( int n) // implementation
{
    value -= n;
}
```

- Note: global functions can be also defined as inline function

Inline Functions

Some notes on inline functions:

- The compiler may ignore your ***request*** to inline a function
- Any change to an inline function will require that all functions that call it be recompiled. This may be significant for large programs.
- Inlining can be done implicitly or explicitly.
- Inline functions cannot contain a loop, switch.
- Inline functions cannot be recursive.
- Inline functions cannot contain an array.
- Inline functions improve execution times, but increase program size.

Closer Look at Standard I/O in C++

Input from keyboard

- To read any character, including white spaces, use `get()` member function.

```
char ch;  
cout << "\nEnter a character: ";  
ch = cin.get();
```

- to read strings, including white spaces use `getline` function:

```
string name;  
cout << "\nEnter your name: ";  
getline (cin, name);
```

- In this example `string` is a predefined library class that creates objects that hold a string of characters. For students who have programming background with Processing language, string object in C++ works slightly like String object in Processing.
- We will discuss this data type in more detail later in this course.

Formatting your Output

- To format your output you should include header file `<iomanip>`
- You can set the format flag either to fixed point or scientific by:

```
cout << setiosflags( ios::fixed );  
cout << setiosflags( ios::scientific );
```
- Another option to set the format flag to fixed is:

```
cout << fixed;
```
- Now you can set the precision (number of digits after the decimal point) by:

```
cout << setprecision(2); // sets the precision to 2.
```

Example

- Consider the following statements:

```
double x = 61110.56673
```

```
cout << setiosflags(ios::fixed) << setprecision(2) << x;
```

- What is the output?

61110.57

- Consider the following statements:

```
cout << setiosflags(ios::scientific) << setprecision(2) << x;
```

- What is the output?

6.1e+04

Set the Field Size

- You may use `setw` to indicate the field size. The following statement displays the value of `x` in a field of 10.

```
double x = 61110.56673
cout << setiosflags(ios::fixed) << setprecision(2)
      << " x = " << setw(10) << x "." ;
```

- Assuming Δ is one space, the output is:

x = 61110.56 $\Delta\Delta$.

- Another example:

```
int a = 3, b = 5;
cout << setw(7) << a << setw(7) << b;
```

- Assuming each Δ represents one space the output is:

$\Delta\Delta\Delta\Delta\Delta\Delta$ 3 $\Delta\Delta\Delta\Delta\Delta\Delta$ 5

- Each of the number are printed in a field of 7.

Cleaning Buffer

Input Stream

- A C++ programs sees input as a stream of characters, not as a sequence of lines of text. `cin` will consume just enough characters to do its job, or to find out that it can't do its job.
- Remaining characters on the input stream are not consumed--instead they remain in the stream, waiting for the next input operation.
- If you type and run this program, you see that the second `cin` in the program that is supposed to read a character does not work! Why?

```
#include <iostream>
using namespace std;
void main( void )
{
    int i;
    char ch;

    cout << "Please enter an integer ";
    cin >> i;

    cout << "Please enter a character ";
    cin >> ch;
}
```

Solution:

- To solve the problem in the previous slide, the input stream (input buffer) must be cleaned up.
- The following code segment cleans up the standard input buffer (what ever is leftover from last read

```
char ch;  
cin.clear(); // clear the cin failure status  
do  
{  
    ch = cin.get(); // read a character  
} while (ch != '\n' && ch != EOF);
```

Example

```
#include <iostream>
#include <string>
using namespace std;
void cleanStandardInputStream (void){
    cin.clear();
    int leftover;
    do
    {
        leftover = cin.get();
    } while (leftover != '\n' && leftover != EOF);

} // end of function

int main(void){
    int a, b, c;
    cout << "enter a value for a: " ;
    cin >> a;
    cleanStandardInputStream();

    cout << "enter a value for b: " ;
    cin >> b;
    cleanStandardInputStream();
    return 0;
} // end of main
```