Chapter-4
Arrays, Pointers and References

Arrays of Objects

Example:

```
#include <iostream>
using namespace std;
class samp {
private:
int a;
public:
void set_a(int n)  {a=n;}
int get_a() {return a;}
};
```

```
int main()
{
samp object_1[4];
int i;
for (i=0; i<4 i++) object_1[i].set_a(i);
for (i=0; i<4 i++) cout<<object_1[i].get_a()<<”  “;
cout<<endl;
return 0;
}
```

If a class type includes constructor, an array of objects can be initialized.

Example:

```
#include <iostream>
using namespace std;
class samp {
private:  int a;
public:
samp(int n)  {a=n;}
int get_a() {return a;}
};
int main(){
samp object_1[4]={-1,-2,-3,-4};
int i;
for (i=0; i<4 i++) cout<<object_1[i].get_a()<<”  “;
cout<<endl;
return 0;
}
```

You can also have multidimensional arrays of objects.

Example:

```
#include <iostream>
using namespace std;
class samp {
private:  int a;
public:
samp(int n)  {a=n;}
int get_a() {return a;}
};
int main() {
samp object_1[4][2]={1,2,3,4,5,6,7,8};
int i;
for (i=0; i<4 i++) {
cout<<object_1[i][0].get_a()<<”  “;
cout<<object_1[i][1].get_a()<<”  “;
cout<<endl;}
return 0;}
```

When initializing an array of objects whose constructor takes more than one argument, you must use the alternative form of initialization.

Example:

```
#include <iostream>
using namespace std;
class samp {
private:  int a,b;
public:
samp(int n, int m)  {a=n; b=m;}
int get_a() {return a;}
int get_b() {return b;}
};
```

```
int main()
{
samp object_1[4][2]={
samp(1,2),samp(3,4),samp(5,6),samp(7,8),samp(9,10),
samp(11,12),samp(13,14),samp(15,16)
};
int i;
for (i=0; i<4 i++)
{
cout<<object_1[i][0].get_a()<<”  “;
cout<<object_1[i][0].get_b()<<”  “<<endl;
cout<<object_1[i][1].get_a()<<”  “;
cout<<object_1[i][1].get_b()<<”  “<<endl;
}
cout<<endl;
return 0;
}
```

Using Pointers to Objects

Objects can be accessed via pointers. Pointer arithmetic using an object is the same as it is for any other data type.

Example:

```
#include <iostream>
using namespace std;
class samp {
private:  int a,b;
public:
samp(int n, int m)  {a=n; b=m;}
int get_a() {return a;}
int get_b() {return b;}
};
```

```
int main()
{
samp object_1[4]={
samp(1,2),samp(3,4),samp(5,6),samp(7,8)};
int i;
samp *p;
p=object_1; // Get starting address of array.
for (i=0; i<4 i++)
{
cout<<p->get_a()<<”  “;
cout<<p->get_b()<<”  “<<endl;
p++; //Advance to next object.
}
cout<<endl;
return 0;
}
```

The this pointer

C++ contains a special pointer that is called **this**. **this** is a pointer that is automatically passed to any member function when it is called, and it is a pointer to the object that generates the call.

Example:

```
#include <iostream>
#include <cstring>
using namespace std;
class inventory {
private: char item[20];
double cost;
int on_hand;
```

```
public:
inventory(char * i,double c,int o)
{
strcpy(item,i);
cost=c; on_hand=o;
}
void show();
};
void inventory::show()  {
cout<<item<<": $"<<cost<<" On hand: "<<on_hand<<endl;}

int main()
{
inventory object_1("wrench",4.95,4);
object_1.show();
return 0;
}
```

The preceding program could be written as shown here:

```
#include <iostream>
#include <cstring>
using namespace std;
class inventory {
private: char item[20];
double cost;
int on_hand;
public:
inventory(char i,double c,int o){
strcpy(this->item,i);
this->cost=c; this->on_hand=o;
}
void show();
};
void inventory::show()  {cout<<this->item<<": $"<< this->cost<<"
On hand: "<<this->on_hand<<endl;}
```

```
int main()
{
inventory object_1(“wrench”,4.95,4);
object_1.show();
return 0;
}
```

Using **new** and **delete**

C++ provides a safer and more convenient way to allocate and free memory compared to **malloc()** and **free()** functions. In C++ you can allocate memory using **new** and release it using **delete.** These operators takes these general forms:

p-var=**new** type;
delete p-var;

type: Type specifier of the object (or variable).
p-var: A pointer to the object (or variable).

- **new** automatically allocates enough memory to hold an object of the specified type. You do not need to use **sizeof** function.
- **new** automatically returns a pointer of the specified type.

Example:

```
#include <iostream>
using namespace std;
int main ()
{
int *p
p=new int; //allocate room for an integer.
if(!p) {
cout <<”Allocation error”<<endl;
return 1;
}
*p=1000;
cout<<”Here is integer at p : “<<*p<<endl;
delete p //release memory
return 0;
};
```

Here is an example that allocates an object dynamically:

```
#include <iostream>
using namespace std;
class samp { private:  int a,b;
public:
int set_ab(int i, int j) {a= i; b= j;}
int get_product() {return a*b;}
};
int main;
{ samp *p;
p=new samp; //allocate object
if(!p) {
cout <<”Allocation error”<<endl;
return 1;
}
p->set_ab(4,5);
cout<<”Product is : “<<p->get_product()<<endl;
delete p; return 0;}
```

More about New and Delete

p-var=**new** type (initial value); //You can give dynamically allocated object an initial value by using this form.

p-var=**new** type[size]; //To dynamically allocate a one dimensional array, use this form of **new**.

delete [] p-var; //To delate dynamically allocated array, use this form of **delete**.

Example:

```
#include <iostream>
using namespace std;
int main ()
{
int *p
p=new int [5]; //5 allocate room for 5 integers.
if(!p) {
cout <<”Allocation error”<<endl;
return 1;
}
for (i=0;i<5;i++)  p[i]=i;
for (i=0;i<5;i++)  {
cout<<”Here is the integer at p[”<<i<<”] : “<<p[i]<<endl;
};
delete  [] p //Release memory.
return 0;}
```

Example: The program creates a dynamic array of objects.

```
#include <iostream>
using namespace std;
class samp { private:  int a,b;
public:
int set_ab(int i, int j) {a= i; b= j;}
int get_product() {return a*b;}   };
int main()
{ samp *p;int i;
p=new samp[10]; //allocate object array.
if(!p) {
cout <<”Allocation error”<<endl;
return 1;}
for (i=0;i<10;i++) p[i].set_ab(i,i);
for (i=0;i<10;i++)
cout<<”Product[“<<i<<”]: ”<<p[i].get_product()<<endl;
delete [] p; return 0;}}
```

References

C++ contains a feature that is related to the pointer: the *reference*. A reference is not a pointer. A reference is an implicit pointer that for all intents and purposes acts like another name for a variable. There are three ways that a reference can be used. First, a reference can be passed to a function. Second, a reference can be returned by a function. Finally, an independent reference can be created.

An example in C:

```
#include <iostream>
using namespace std;
void f(int *n);
int main()
{ int i=0;
f(&i);
cout<<”Here is i‘s new value : “<<i<<endl;
return 0;}
void f(int *n){*n=100;} //put 100 into the argument pointed to by n.
```

The same program in C++:

```
#include <iostream>
using namespace std;
void f(int &n); //declare a reference parameter
int main()
{ int i=0;
f(i);
cout<<”Here is i‘s new value : “<<i<<endl;
return 0;
}
void f(int &n)
{
n=100; //put 100 into the argument used to call f1()
}
```

Passing References to Functions

When you pass the object by reference, no copy is made, and therefor its dectructor function is not called when the function returns.

Example:

```
#include<iostream>
using namespace std;
class myclass {
private: int who;
public:
myclass (int n) {
who=n;
cout<<”Constructing the object “<<who<<endl; }
~myclass () {cout<<”Destructing the object “<<who<<endl;}
int id() {return who;}
};
```

```
//o is passed by value.
void f(myclass o)
{
cout<< “Received ”<< o.id()<< endl;
}
int main() {
myclass x(1);
f(x);
return 0;}
```

This function displays the following

```
Constructing the object  1
Received 1
Destructing the object  1
Destructing the object  1
```

Referans parametresi kullanırsak aşağıdaki gibi olur.

Example: The object is passed by reference.

```
#include<iostream>
using namespace std;
class myclass {
private: int who;
public:
myclass (int n) {
who=n;
cout<<" Constructing the object "<<who<<endl; }
~myclass () {cout<<" Destructing the object "<<who<<endl;}
int id() {return who;}
};
//o is passed by reference.
void f(myclass &o)
{
cout<<"Received "<<o.id() <<endl;
}
```

```
int main()
{
myclass x(1);
f(x);
return 0;
}
```

This version displays the following output.

```
Constructing the object  1
Received 1
Destructing the object  1
```

Returning References

A function can return a reference. Returning a reference can be very usufel when you are overloading certain types of operators. It also can be employed to allow a function to be used on the left side of an assignment statement.

Example:

```
#include<iostream>
using namespace std;
int &f(); //return a reference.
int x;
int main()
{ f()=100;  //assign 100 to reference returned by f().
cout<<x<<endl;
return 0;}
int &f() //return an int reference.
{
return x;// return a reference to x.
}
```

Example:

```
int &f()                  //Return an int reference.
{
int x ;                  //x is a local variable
return x;                 // returns a reference to x.
}                         //
```

X is local to f() and will go out of scope when f() returns. This effectively means that the reference returned by f() is useless.

Example:

```
#include <iostream>
#include <cstdlib>
using namespace std;
class array {
int size;
char *p;
public:
array (int num);
~array() {delete [] p;}
char &put(int i);
char get(int i);
};
array::array(int num) {
p=new char [num];
if(!p) { cout <<”Allocation error.”<<endl;exit (1);}
size=num;
}
```

```
char &array::put(int i)  //Put somthing into the array.
{ if (i<0 || i>=size) { cout<<"sınır hatası"<<endl;exit(1)}
return  p[i];} // return reference to p[i]
char array::get(int i)  //Get something from the array.
{ if (i<0 || i>=size) { cout<<"Bounds error"<<endl;exit(1)}
return p[i];} //return character
int main ()
{ array a(10);
a.put(3)='X';
a.put(2)='R';
cout<<a.get(3)<<"  "<<a.get(2)<<endl;
a.put(11)='!'; //generate run-time boundary error.
return 0;
}
```

Independent References and Restrictions

An independent reference is a reference variable that is another name for the variable. An independent reference must be initialized when it is declared.

Restrictions:

- You cannot reference another reference.
- You cannot obtain address of a reference.
- You cannot create arrays of references.
- You cannot reference a bit-field.
- References must be initialized unless they are members of a class, or return values or are function parameters.

Example:

```
#include <iostream>
using namespace std;
int main (){
int x;
int &ref=x;
x=10;
ref=100; //x=ref=100;
cout<<x<<”   “<<ref<<endl; //prints the number 100 twice.
return 0;}
```

Chapter 5
Function Overloading

Overloading Constructor Functions

Example

```
#include<iostream>
using namespace std;
class myclass {
private: int x;
public:
myclass () {x=0;} //no initializer
myclass (int n) {x=n;}// initializer
int getx() {return x;}
};
```

```
int main ()
{
myclass object_1(10);//declare with initial value.
myclass object_2;// declare without initializer
cout<<"object_1: "<<object_1.getx()<<endl;
cout<<"object_2: "<<object_2.getx()<<endl;
return 0;
}
```

Example:

```
int main ()
{
 myclass object_1[10];////declare array without initializers.
//declare array with initializers.
myclass object_2[10]={1,2,3,4,5,6,7,8,9,10};
for (i=1;i<10;i++){
cout<<"object_1[ "<<i<<"]:"object_1.getx()<<"  "<<endl;
cout<<"object_2[ "<<i<<"]:"object_2.getx()<<"  "<<endl;}
return 0;
 }
```

Example:

```
#include <iostream>
#include <cstdio>
using namespace std;
class date{
int day, month,year;
public:
date(char *str);
date(int m, int d, int y) {day=d;month=m;year=y}
void
show(){cout<<month<<'/'<<day<<'/'<<year<<endl;}
};
date::date(char *str)
{sscanf(str,”%d*c%d*c%d”,&month,&day;&year);}
```

```
int main()
{
date sdate(“8/23/2010”);
date idate(8,23,2010);
sdate.show();
idate.show();
return 0;
}
```

Example:

```
#include<iostream>
using namespace std;
class myclass
{
private: int x;
public:
myclass () {x=0;} //no initializer
myclass (int n) {x=n;} //initializer
int getx() {return x;}
void setx(int n) {x=n;}
};
```

```
int main()
{myclass *p;
myclass object_1(10); //initialize single variable
p=new myclass[10];//can not use initializer here..
if(!p)
{
cout<<”allocation error”<<endl;return 1;
}
int i;
for (i=0;i<10;i++)  p[i]=object_1;
for (i=0;i<10;i++)
{
cout<<”p[ ”<<i<<”]:”p[i].getx()<<”  “<<endl;
}
delete []p;
return 0;
}
```

Creating and Using a Copy Constructor

- Problems can occur when an object is passed to or returned from a function. One way to avoid these problems is to define a copy constructor.
- When an object is passed to a function, a bitwise (exact) copy of that object is made and given to the function parameter that receives the object. However, there are cases in which this identical copy is not desirable.
- For example, if the object contains a pointer to allocated memory, the copy will point to the same memory as does the original object. Therefore if the copy makes a change to the contents of this memory, it will be changed for the original object too.

•Copy constructor is called (automatically) in the following three situations.
•Whe an object is used to initialize another in a declaration statement.
•When an object is passed as a parameter to a function.
•When a temporary object is created for use as a return value by a function

The most common form of a copy constructor is shown here:
classname (const classname &obj)
{ //body of constructor.}

Here *obj* is reference to an object that is being used to initialize another object. For example, assuming a class called **myclass**, and that y is an object of type **myclass**, the following statements would invoke the **myclass** copy constructor:

```
myclass x=y; // y explicitly initializing x
func1(y);       // y passed as a parameter.
z=func2();      // z receiving a returned object.
```

In the first two cases, reference to **y** would be passed to the copy constructor. In the third, a reference to the object returnd by **func2()** is passed to the copy constructor.

Example: (copy constructor.)

```
#include <iostream>
#include <cstdlib>
using namespace std;
class array {int size;int *p;
public:
array (int sz) {//constructor
p=new int[sz];
if (!p) exit(1);
size=sz;
cout<<"Using "normal" constructor"<<endl;}
~array() {delete [] p;}
array(const array &ob1) //Copy conctructor
int get(int i) {return p[i]};
void put(int i, int j) {if (i>=0 && i<size p[i]=j}; };
```

```
array::array(const array &a ){ int i;
size=a.size;
p=new int[a.size];//Allocate memory.
if (!p) {cout<<“Allocation error”<<endl; exit(1);}
for (i=0:i<a.size;i++) p[i]=a.p[i]; //copy contents
cout <<”using copy constructor”<<endl;}
int main (){ array num(10); //this calls normal constructor.
array num_1(20); // this calls normal constructor.
int i;
for (i=0:i<num.size;i++) num.put(i,i);
//put some values into the array
for (i=0:i<num.size;i++) cout<< num.get(i); cout<<endl;
//display num
array x=num; //this invokes copy constructor
for (i=0:i<10;i++) cout<<x.get(i); //display x
num_1=num; //does not call copy constructor.
return 0;
}
```

Example: This program does not use copy constructor (incorrect)

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class strtype {
private:
char *p;
public:
strtype(char s);//constructor
~strtype() {delete []p;}//destructor
char *get() {return p;}
};
strtype::strtype(char *s);{int l;
l=strlen(s)+1;
p=new char [l];
if (!p) {cout<<”Allocation error”<<endl; exit (1);}
strcpy(p,s);}
```

```
void show(strtype x)
{char *s;
s=x.get();
cout<<s<<endl;}
int main ()
{
strtype a(“Hello”), b(“There”);
show (a);
show (b);
return 0;
}
```

// memories allocated for objects ‘a’ and ‘b’ will be released twice (one by object one by the copy of object). This will cause error. To solve the problem, we must use copy constructor.

Example: This program uses copy constructor (correct)

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class strtype {private: char *p;
public:
strtype(char s);//normal constructor
strtype(const strtype &o);// copy constructor
~strtype() {delete []p;}//destructor
char *get() {return p;}
};
strtype::strtype(char *s)
{int l;
l=strlen(s)+1;
p=new char [l];
If (!p) {cout<<”Allocation error”<<endl; exit (1);}
strcpy(p,s);}
```

```
strtype::strtype(strtype &o)
{int l;
l=strlen(o.p)+1;
p=new char [l];
If (!p) {cout<<”Allocation error”<<endl; exit (1);}
strcpy(p,o.p);} //copy string into copy. (copy o.p to p)
void show(strtype x)
{char *s;
s=x.get();
cout<<s<<endl;}
int main ()
{
strtype a(“Hello”), b(“There”);
show (a);
show (b);
return 0;
}//
```

Using Default Arguments

•There is a feature of C++ that is related to function overloading. This feature is called default argument, and it allows you to give a parameter a default value when no corresponding argument is specified when the function is called.

•Using default arguments is essentially a shorthand form of function overloading.

Example:

```
#include <iostream>
using namespace std;
void f(int a=0, int b=0)
{cout <<"a: "<<a<<",  b: "<<b<<endl;}
int main (){
f();
f(10);
f(10,20);
}
```

Output a:0, b:0
        a:10, b:0
        a:10, b:20

Example:

```
#include<iostream>
using namespace std;
class myclass {
private: int x;
public:
myclass (int n=0) {x=n;} //use default argument instead of overloading
int getx() {return x;}
};
int main()
{
myclass o1(10);// deeclare with initial value.
myclass o2;//declare without initializer.
cout<<"o1: "<<o1.getx()<<endl;
cout<<"o2: "<<o2.getx()<<endl;
}
```

Overloading and Ambiguity

- When you are overloading functions, it is possible to introduce ambiguity into your program.
- Overloading-caused ambiguity can be introduced through type conversion, reference parameters, and default arguments.
- Ambiguity must be removed before your program will compile without error.

Example:

```
#include <iostream>
using namespace std;
float f(float i) {return i/2.0;}
double f(double i) {return i/3.0;}
int main (){
float x=10.09;double y=10.09;
cout<<f(x)<<endl; //unambiguous
cout<<f(y)<<endl; // unambiguous
cout<<f(10)<<endl; //ambiguous, convert 10 to double or float.
return 0;}
```

Example:

```
#include <iostream>
using namespace std;
void f(unsigned char c) {cout<<c<<endl;}
void f( char c) {cout<<c<<endl;}
int main ()
{
f('c'); // unambiguous.
f(86);    // ambiguous. Which f() is called.
return 0;
}
```

Example:

```
#include <iostream>
using namespace std;
int f(int a, int b) {return a+b;}
int f(int a, int &b) {return a-b;}
int main ()
{
Int x=1, y=2;
cout<<f(x,y);    //Belirsiz. Hangi f() çağrılacak?
return 0;
}
```

Finding the Address of an Overloaded Function

Let zap() be a function and p be a pointer. In C, The address of zap() can be assigned to p as:

p=zap;

- In C, any type of pointer can be used to point to a function because there is only one function that it can point to.
- In C++, the situation is a bit more complex because a function can be overloaded.Thus, there must be some mechanism that determins which function's address is obtained.
- When obtaining the address of an overloaded function, it is the way the pointer is declared that determines which overloaded function's address will be optained.
- The pointer's declaration is matched againest those of the overloaded functions.

Example: Illustrate assigned function pointers to overloaded functions.

```
#include <iostream>
using namespace std;
void space(int count)
{for (;count;count--) cout<<' '; cout<<“end”<<endl;}
void space(int count,char ch)
{for (;count;count--) cout<<ch;cout<<“end”<<endl;}
int main ()
{
void (*fp1) (int);/*create a pointer to void function with one int
parameter. */
void (*fp2) (int,char);/* create a pointer to void function with one
int parameter and one character parameter.*/
fp1=space; //gets address of space(int).
fp2=space; // gets address of space(int, char).
fp1(22);//output 22 spaces.
fp2(30,'x');//output 30 x’s.
return 0;
}
```