



Lecture Four

Arrays, Pointers and References

Ref: Herbert Schildt, Teach Yourself C++, Third Edⁿ (Chapter 3)

© Dr. M. Mahfuzul Islam
Professor, Dept. of CSE, BUET



Arrays of Objects

- An array of objects is **declared** and **accessed** exactly the same way of other type of variables.
- If a class include a **constructor**, an array of objects can be **initialized**.

```
#include <iostream>
using namespace std;
class samp {
    int i;
public:
    void set_i(int n) {i= n;}
    int get_i() { return i;}
};
```

```
int main() {
    samp ob[4];
    int i;

    for(i=0; i<4; i++) ob.set[i].set_i(i);

    for(i=0; i<4; i++) cout << ob.set[i].get_i();

    return 0;
}
```

An array of objects can be **initialized** in two ways.



Arrays of Objects

(1) For single argument:

```
#include <iostream>
using namespace std;

class samp {
    int i;
public:
    samp(int n) { i = n;};
    int get_i() { return i;}
};
```

(a) For 1D array:

```
int main() {
    samp ob[4] = {-1, -2, -3, -4};

    for( int i= 0; i < 4; i++)
        cout << ob[i].get_i() << ' ';
    cout << "\n";
    return 0;
}
```

(b) For 1D array alternative:

```
int main() {
    samp ob[4] = { samp(-1), samp(-2),
                  samp(-3), samp(-4)};

    for( int i= 0; i < 4; i++)
        cout << ob[i].get_i() << ' ';
    cout << "\n";
    return 0;
}
```

(c) For 2D array:

```
int main() {
    samp ob[4][2] = {1, 2, 3, 4, 5, 6, 7, 8};

    for( int i= 0; i < 4; i++) {
        cout << ob[i][0].get_i() << ' ';
        cout << ob[i][1].get_i() << ' ';
    }
    return 0;
}
```



Arrays of Objects

(2) For multiple arguments:

```
#include <iostream>
using namespace std;

class myclass {
    int a, b;
public:
    void myclass(int i, int j) { a = i; b = j;}
    void show() { cout << a << ' ' <<
                  b << '\n'; }
};
```

```
int main() {
    myclass ob[4] = {myclass(1, 2), myclass(3, 4),
                    myclass(2, 4), myclass(4, 5)};

    for( int i= 0; i < 4; i++)
        cout << ob[i].show();

    return 0;
}
```



Using Pointers to Objects

- When a pointer to object is used, the object's members are referenced using **the arrow (->) operator** instead of the dot (.) operator.
- **Pointer arithmetic** using an object pointer is the same as it for any other data type.

```
#include <iostream>
using namespace std;

class myclass {
    int a, b;
public:
    void myclass(int i, int j) { a = i; b = j;}
    int get_a() { return a;}
    int get_b() { return b;}
};
```

```
int main() {
    myclass ob[4] = {myclass(1, 2), myclass(3, 4),
                     myclass(2, 4), myclass(4, 5)};
    myclass *p;

    p = ob;
    for( int i= 0; i < 4; i++) {
        cout << p->get_a() << " ";
        cout << p->get_b() << '\n';
        p++; //advance to next object
    }
    return 0;
}
```



The **this** Pointer

- C++ contains a special pointer called **this** that is automatically passed to any member function when it is called.
- No C++ programmer uses the **this** pointer to access a class member because the shorthand form is much easier.

```
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
    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;
    cout << " : $" << this->cost;
    cout << "    On hand: " << this->on_hand
         << "\n";
}
```

```
int main() {
    inventory ob("wrench", 4.95, 4);

    ob.show();
    return 0;
}
```



Using **new** AND **delete**

- C++ uses **new** operator for dynamically allocating memory (C uses **malloc()**).
- The general form of new operator: **p-var = new type;**
- If there is insufficient available memory, **new** responses varies ways-
 - When C++ was first invented, **new** returned **null** on failure.
 - Later, **new** causes an **exception** on failure.
 - Microsoft visual C++, returns a **null** pointer when **new** fails.
 - Borland C++ generates an exception when **new** fails.
- **Advantages of new:**
 1. Automatically allocate enough memory, no need of **sizeof()**.
 2. No explicit type cast is required.
 3. Both **new** and **delete** can be overloaded.
 4. It is possible to initialize the dynamically allocated memory.
- **General form of initializing dynamic memory:**
p-var = new type (initial value);
- **General form of allocating 1D array:**
p-var = new type [size];



Using **new** AND **delete**

- C++ uses **delete** operator for releasing dynamically allocating memory (C uses **free()**).
- The general form of delete operator: **delete p-var;**
- **General form of releasing dynamically allocated array:**
delete [] p-var;

```
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    samp(int a, int b) { i = a; j = b;};
    int get_product() { return i*j;};
};
```

```
int main() {
    samp *p;

    p = new samp (6, 5);
    if (!p){
        cout << "Allocation error\n";
        return 1;
    }
    cout << "Product is: " << p->get_product()
        << "\n";
    delete p;

    return 0;
}
```



Using new AND delete

Dynamic allocation of array

```
#include <iostream>
using namespace std;
```

```
class samp {
    int i, j;
public:
    void set_ij(int a, int b) { i = a; j = b;};
    int get_product() { return i*j;};
};
```

```
int main() {
    samp *p;

    p = new samp [10];
    if (!p){
        cout << "Allocation error\n";
        return 1;
    }
    for(int i = 0; i < 10; ++i){
        p[i].set_ij(i, 2*i);
    }

    for(int i = 0; i < 10; ++i){
        cout << "Product {" << i << "} is: ";
        cout << p[i].get_product() << "\n";
    }

    delete [] p;

    return 0;
}
```



References

- 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:
 - ✓ A reference can be **passed to a function** (most important)
 - ✓ A reference can be **returned by a function**
 - ✓ An **independent reference** can be created.

Using Pointer, not Reference (only way used in C for call by reference)

```
#include <iostream>
using namespace std;

void f( int *n);

int main(){
    int i = 0;

    f(&i);
    cout << "value of i:" << i << "\n";
    return 0;
}

void f( int *n){
    *n = 100;
}
```

Using References

```
#include <iostream>
using namespace std;

void f(int &n);

int main(){
    int i = 0;

    f(i);
    cout << "value of i:" << i << "\n";
    return 0;
}

void f( int &n){
    n = 100;
}
```



References

- When a **reference parameter** is used, the compiler automatically passes the **address of the variable** as the argument.
- There is no need to manually generate the address of the argument by preceding it with an **&** (**in fact, it is not allowed**).
- Within the function, the compiler automatically uses the variable pointed to by the reference parameter, no need to employ *****.
- A reference parameter **fully automates** the **call-by-reference** parameter passing mechanism.

```
void f( int &n){
    n = 100;
    n++;
}
```

- In the above example, instead of incrementing **n**, this statement increments the value of the variable being referenced (in this case, **i**).



Advantages of Reference Parameters

There are several advantages of using **reference parameters** over their equivalent **pointer** alternatives:

1. No longer need to **remember** to pass the **address of an argument**.
2. Reference parameters offer a **cleaner, more elegant interface** than the rather clumsy explicit pointer mechanism.
3. When an object is passed to a function as a reference, no copy is made.

➤ When an object is passed to a function by using **call by value**, constructor is called only **once** and destructor is called **several times**, causing **serious problem**.

➤ Two solutions: to pass an object using **call by reference** and using **copy constructor**.



Passing References to Object

- When an object is passed to a function by **reference**, no copy is made and therefore its destructor function is not called when the function returns.

```
#include <iostream>
using namespace std;

class myclass {
    int who;
public:
    myclass (int n){
        who = n;
        cout << "Constructing....\n";
    }
    ~myclass() {
        cout << "destructing...\n";
    }
    int id() { return who; }
};
```

```
void f(myclass &o) {
    cout << "Received: " << o.id() << '\n';
}

int main(){
    myclass x(1);

    f(x);
    return 0;
}
```

Note that, a **reference is not a pointer**. Therefore, when an object is passed by reference, the member access operator remains **the dot (.)**, not the arrow (->).



Returning Reference

- Very useful for **overloading** certain types of operator.
- Allow a function to be used on **the left side** of an assignment statement.

```
#include <iostream>
using namespace std;

int &f();
int x;

int main() {
    f() = 100;
    cout << x << '\n';
    return 0;
}

int &f(){
    return x;
}
```

BUT

```
int &f(){
    int x;
    return x;
}
```



Independent Reference

- An independent reference is a reference variable that in all effects is simply **another name for another variable**.
- Because reference cannot be assigned new values, an **independent reference must be initialized when it is declared**.

```
#include <iostream>
using namespace std;

int main() {
    int x;
    int &ref = x;

    x = 10;
    ref = 100;
    cout << x << " " << ref << '\n';

    return 0;
}
```

The independent reference `ref` serves as a different name for `x`.

Independent reference cannot be a constant like
`const int &ref = 10;`