References and pointers

- Pointers are variables whose value is a reference, i.e. an address of a store location.
- Early languages (Fortran, COBOL, Algol 60) had no pointers; added to Fortran 90.
- In Pascal, Ada, C and C++, we can declare pointer variables.
- In C and C++, we can also ***do arithmetic*** on pointer variables.
- In Java, objects are always accessed via references (pointers), but we do not explicitly declare pointer variables.

Chapter 3

References and pointers

With pointer (reference) variables, there are two declaration-reference bindings to deal with, the allocation of space to the:

- pointer variable,
- the item the pointer variable is pointing at.

References and pointers

Consider the effect of the following declarations:

```
int i;        //Java, C, C++
int* ipoint;  // C, C++
```

What about these statements?

```
ipoint = new int; // C++
int* ipoint = new int;
```

References and pointers

What is the effect of this statement?

```
*ipoint = 27;   //C
```

When we leave the method in which **`ipoint`** is declared, the location it is pointing at becomes inaccessible.

It is now ***garbage***.

Why?

References and pointers

C and C++ allow more complex assignments (assuming that i is an integer variable):

```
ipoint = &i;
```

There is no equivalent of the & operator in Java.

What is the value of i after execution of:

```
i = 34;
*ipoint = 27;
```

What is ipoint pointing to after

```
ipoint = ipoint + 1;
// C and C++
```

Garbage

When space for variables is allocated dynamically, garbage can be created.

Suppose we have:

```
int* ipoint = new int;
*ipoint = 34;
…
ipoint = new int;
```

What has happened to the location containing **`34`** ?

Dangling reference

Suppose we have:

```
int* ipoint = new int;
int* another = ipoint;
```

Both `another` and `ipoint` now point to the same store location.

This is known as ***aliasing***.

Dangling reference

```
delete ipoint; // C ++
ipoint = new int;
```

What is `another` pointing to?

We have a ***dangling reference***.

Dangling reference

Dangling references are a ***MAJOR*** problem.

Leads to programs giving ***wrong*** results.

Why?

Example of the problem with ***aliases***.

Dangling references are much more serious than garbage.

Garbage Collection

Space for dynamic variables organised as a ***free space list*** in what is known as the ***heap***.

Call of **new** takes the required amount of space from the free space list, while a call of **`delete`** in C++ returns it so that it can be re-used later.

In Java, a ***garbage collector*** automatically determines store locations which have been allocated, but which are no longer accessible.

It then collects them together and returns them to the free space list - no need for **`delete`**.

This removes the problem of dangling references.

Pointers to objects

In Pascal, Ada, C and C++ we can declare pointers to any type.

Eg.

```
Pair* p = new Pair(43, 5);
```

// C ++

What does the * tell us?

In Java, the equivalent declaration is:

```
Pair p  = new Pair(43, 5);
```

Objects

In C++, we can declare a **`Pair`** object.

```
Pair obj(43, 5);
```

Space for the **`Pair`** object is allocated on block entry.

Suppose that class **`Pair`** had a void method **`mkBigger`**:

```
obj.mkBigger();
```

Calling methods

Recall **`p`** is a reference to a **`Pair`** object.

In Java:

```
p.mkBigger();
```

In C++ :

```
(*p).mkBigger();
```

This is rather messy and so C++ has the syntactic sugar:

```
p->mkBigger();
```

Arrays

Let us look again at arrays.

Example:

```
int[] a = new int[10]; //Java
int* a = new int[10];  //C, C++
```

Space allocated on the stack or the heap?

We access the elements of the array as:

`a[0]`, **`a[1]`**, **`a[j]`** etc.

Arrays

What about:

```
int aa[10]; //C, C++
```

What is the effect of:

```
int* b;
b = aa;
```

We could instead have written: `b = &aa[0]`

Arrays

We can now access the array elements either as: **`aa[0]`**, **`aa[1]`**, ...

or as: **`b[0]`**, **`b[1]`**, ...

Also, instead of writing:

`aa[0] = 17;` or **`b[0] = 17;`**

we can write:

`*aa = 17;` or **`*b = 17;`**

i.e. the element pointed to by b is given the value 17.

Arithmetic on pointers

C and C++ allow us to do arithmetic on pointers.

Hence, instead of:

```
for (int i = 0; i < 10; i++)
     b[i] = 0;
```

we can write:

```
for (int i = 0; i < 10; i++)
    *b++ = 0;
```

Each time round the loop, the **++** causes **b** to point to the next element.

This gives very fast access to arrays.

Pitfalls

When using pointers, we must ensure that they are pointing to allocated space.

Consider:

```
int aa[10];
int* b = aa;
int* c = new int[10];
```

Where is the storage allocated? How much is allocated? What happens when we leave the block?

Pitfalls

There are some common, but subtle pitfalls.

Consider:

```
int* fun() {
    int aa[10];
    ...
    return aa; // wrong!!
}
```

The function returns a *pointer to an* **`int`** and so this is syntactically OK.

Why might this go wrong?

Reserving space

Better:

```
int* fun() {
    int* aa = new int[10];
    ...
    return aa;
}
```

As space for the array is now allocated on the heap, it remains in existence after we leave **`fun`**.

Parameters

Why use parameters to methods?

How do ***pure functions*** differ?

Chapter 6

Object-oriented languages

Remember our **`BankAccount`** example.

In Java, we had:

```
bk1.deposit(6);
```

We pass the value **`6`** to be deposited in our **`bk1`** object.

In Pascal, we had:

```
deposit(bk1, 6);
```

We pass over the variable **`bk1`** and the value **`6`** and an ***updated*** version of **`bk1`** is passed back to us.

In Java, **`bk1`** was an implicit extra parameter.

Passing information in

The mechanism used in Algol 60, Pascal, C, C++ and Java is ***call by value***.

Suppose we have the C++ or Java declaration:

```
void ex(int par) { ... }
```

and the call:

```
target.ex(actual);
```

What values are copied, linked, or new storage locations created?

Passing information in

The mechanism used in Ada is called ***call by constant-value***.

This is like call-by-value except that **`par`** is like a ***local constant***.

The advantage of call by value and call by constant-value is that we have a guarantee that the ***actual*** parameter is not modified by the call.

The drawback is that a copy of the actual parameter has to be made. Consider what happens if the parameter is a large array.

Call by reference

What happens when information is to be passed back?

In ***call by reference***, the ***address*** of the actual parameter is passed over. Within the subprogram, the formal parameter is an indirect reference to the actual parameter. Any change in the formal parameter immediately affects the actual parameter.

This is the way that **`var`** parameters in Pascal and ***reference parameters*** in C++ are implemented .

Call by reference

In C++, we have:

```
void swap(int& first, int& second) {
    int temp;
    temp = first;
    first = second;
    second = temp;
} // swap
```

Assuming **`dee`** and **`dum`** are declared as integers, the call is:

```
swap(dee, dum);
```

Java does not have reference parameters.

Call by reference in C

In C, ***all*** parameter passing is done using call-by-value.

Call-by-reference is achieved using pointers:

```
void swap(int* first, int* second) {
    int temp;
    temp = *first;
    *first = *second; // what??
    *second = temp;
}
```

Method call:

```
swap(&dee, &dum);// recall & means address of
```

The syntax is different, but the C and the C++ versions of **`swap`** are implemented in exactly the same way.

Call by reference

Parameters in Java are always passed by value.

Primitive types and references are therefore handled differently.

Example:

```
Someclass p;
```

Can values in the object referred to by **`p`** be changed?

Call by reference

Consider what happens with the method:

```
void update(Someclass f) {
    ...
    f.someChange(); ...
} // fun
```

and the method call:

```
target.update(p);
```

Call by reference

The advantage of call by reference with structured types is that less space is needed. In call by value with an array (or indeed any large data structure), a copy of the complete array is passed over. In call by reference, only a pointer to the array is copied over. Hence, many Pascal programmers use this method with arrays, even when the array is not going to be changed.

Call by reference

However there is then the disadvantage that we do not have a ***guarantee*** that the actual parameter will remain unchanged. Also accessing a variable through a pointer is slower than accessing a variable directly (because we need more steps to get to the value). Also, aliasing - two references point to the same object/value.

Call by value-result

The value of the actual parameter is passed over.

Within the procedure, changes affect only the local copy. On procedure exit, the actual parameter is updated.

Advantages might include clarity. But can be expensive when used with structured variables.

Needed in distributed systems that do not have shared memory as references are not then possible.

Call by value result

In Ada, scalar objects are passed by value-result.

The syntax is:

```
procedure ex(par : in out integer);
```

However, structured objects in Ada ***may*** by passed by reference.

Functions

No major difference between languages.

In Java, value returning methods correspond to functions.

A function returns a value and is used in expressions, while a procedure is a statement in its own right.

Functions should ***not*** have side-effects:

```
if (f1(e) && f2(c)) ...
```

Here, the effect of calling **`f1`** should not affect the execution of **`f2`**.

Functions should return a value and have no other effect.