Pointers and References

Ajit Rajwade

Refer: Chapter 7 of the **SCHAUM SERIES book**

Address of a Variable

- We saw before: any C++ variable has a name (identifier), an address and a data-type.
- The address of a variable can be obtained via the `&` operator (not to be confused with `&&`), prefixed to the variable's name. The `&` operator is called the **address operator**.
- For example, consider:

```
main(){

int n;

cout << n << endl; // prints the value of n

cout << &n; // prints the address of n

}
```

References

- A **reference** is an *alias* for another variable. It is declared using the reference operator `&` appended to the datatype of the variable.

```
main () {
    int n=3;
    // reference variable below - needs to be initialized
    // during declaration
    int& r = n;
    cout << n << “ “ << r << endl;
    r = 2; // assigns this value to r and n both
    cout << n << “ “ << r << endl;
}
```

Both `r` and `n` have the same address. Any operation on `r` will change the value of `n`. Note that `int r = n;` will make a fresh copy but `r` will then have a different address than `n`.

- We have encountered references earlier in **function declarations by reference**.

Pointers

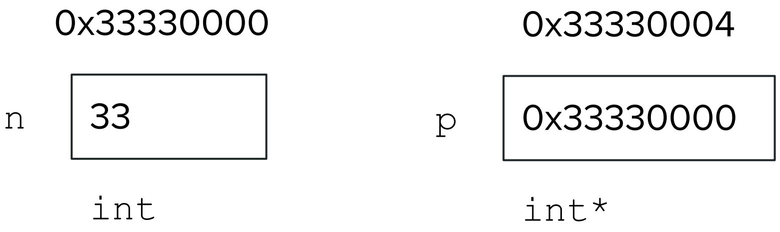
- We can store the address of one variable in another variable which is called a **pointer**.
- If the original variable is of a data-type `D`, then the pointer variable must be declared to be of type `D*`, or pointer to variable of type `D`. `*` stands for “pointer” here.

```
main(){
int n = 33; int *p = &n;
cout << “n = “ << n << “ &n = “ << &n << “ p = “ << p;
}
```

Output: 33 followed by address of `n` printed twice

`p` will contain the address of `n`. Thus when you print `p` and `&n`, the outputs will be the same. The address of `p` itself is different. `p` is called a pointer as its **points** to the address/location of another variable. See next slide.

Pointers



0x33330000

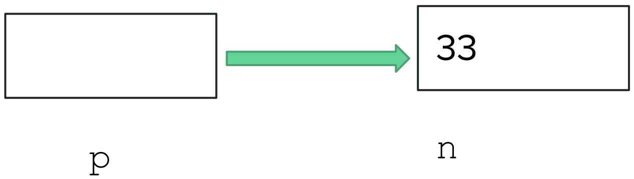
0x33330004

n 33

p 0x33330000

int

int*



33

p

n

Dereferencing a pointer

- In the earlier example, `p` contains the address of `n`.
- `*p` will peek into that address (contained in `p`) and give you the value from the latter address, as shown below:

```
main(){

int n = 33; int *p = &n;

cout << “*p = “ << *p;

}
```

Output: 33

- `*p` is thus an alias for `n`. That is whenever `p = &n`, then `*p` is equal to `n`. In other words, `n` is equal to `*&n` and `p` is equal to `&*p`.
- Thus `&` (address operator) and `*` (dereference operator) are inverses.

Declaring Pointers

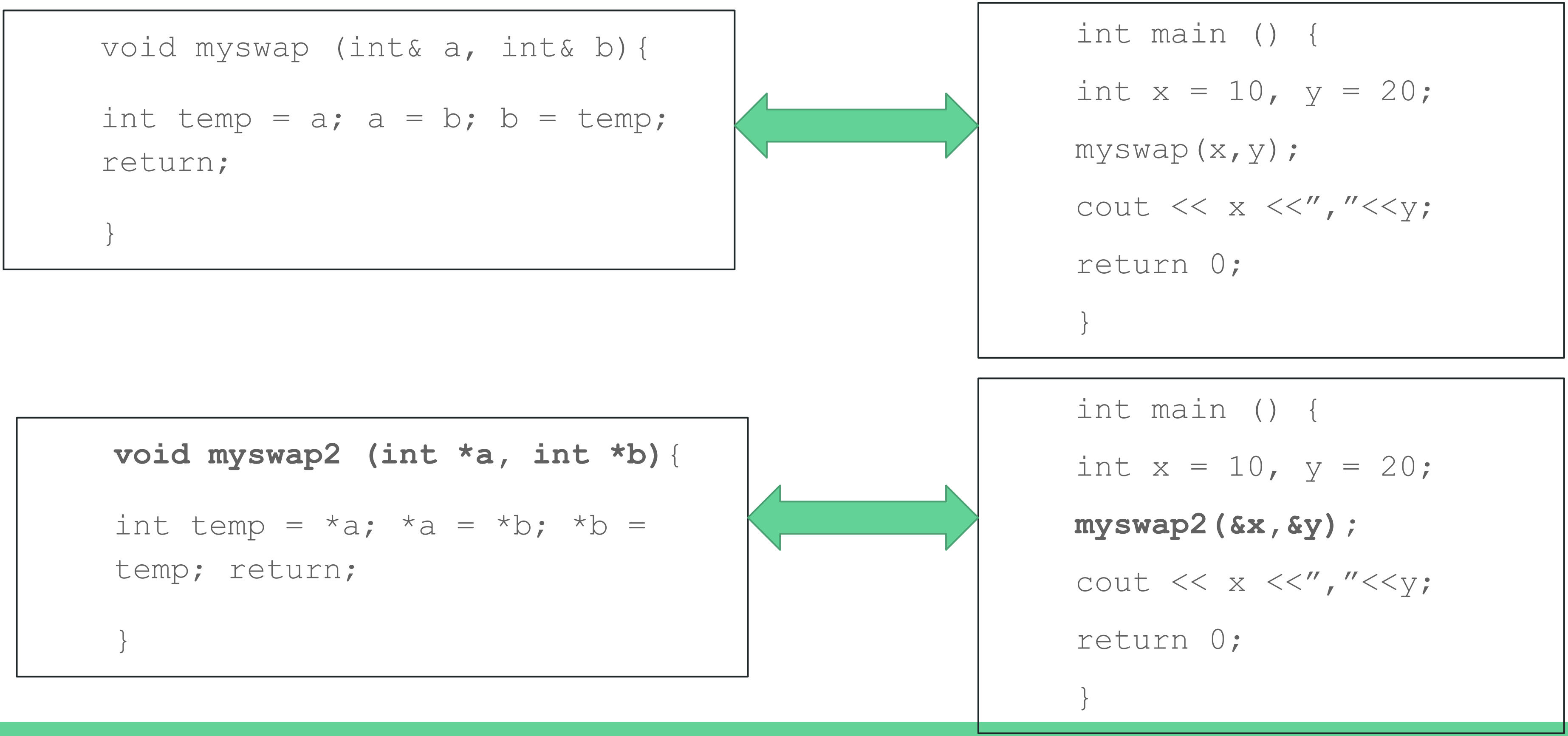
- We looked at int* so far, but equivalently we have float*, double*, char* and so on.
- But note that float* and int* are different data-types and the following code snippet will produce a compilation error:

```
main(){

double q;

int *p = &q;

}
```

- In the declaration int *p, q; note that only p is of type int*, whereas q is of type int. If you want both p and q to be pointers, then the declaration is int *p, *q;

Use of Pointers in Functions

- Recall the swap function from last class: we will rewrite it with pointers



```
void myswap (int& a, int& b){

int temp = a; a = b; b = temp;
return;

}
```

```
int main () {

int x = 10, y = 20;

myswap(x,y);

cout << x <<”,”<<y;

return 0;

}
```

```
void myswap2 (int *a, int *b){

int temp = *a; *a = *b; *b =
temp; return;

}
```

```
int main () {

int x = 10, y = 20;

myswap2(&x,&y);

cout << x <<”,”<<y;

return 0;

}
```

Use of Pointers in Functions

- When `myswap2` is called, the addresses of `x` and `y` are copied into `a` and `b` respectively.
- The body of `myswap2` is executed, during which the values of `*a` and `*b` are exchanged.
- Note that `*a` is the same as `*(&x)`. So when you change the value of `a` via `*a = *b`, you are effectively changing the value of `x` from the main function. Likewise for `*b` and `y`.
- This is because `&x` referred to the address of `x` in the main function.

Use of Pointers in Functions

- We will rewrite the earlier `cartesianToPolar` function using pointers (compared with previous lecture slides)

```
void cartesianToPolar2 (double x, double y, double
*r, double *theta){
    *r = sqrt(x*x + y*y);
    *theta = atan2(y,x);
}
int main (){
double x = 1.0, y = 1.0, theta, r;
cartesianToPolar2 (x,y,&r,&theta);
}
```

The return statement allows you to typically return just a single variable. But pointers give a function the ability to “return” multiple variables!

Pointers to Pointers

- A pointer may point to another pointer. See the following example:

```
main(){
    int a = 10;
    int *pa = &a;
    int **ppa = &pa;
    **ppa = 50;
    cout << “a = “ << a << “ *pa = “ << *pa << “ **ppa = “
<< **ppa;
}
```

Output: all values will be printed as 50.

Be careful with pointer assignments

- You cannot store the address of an `int` variable into an `int` variable.
- You cannot store an `int` value into an `int*` variable.

```
int *v, p;

v = p; // not allowed: compilation error

p = v ; // not allowed: compilation error
```

- The following are not allowed:

```
int &r = 22; // you cannot have a reference to a constant

int *p = &44; // reference operator & cannot be applied to a constant

int w, v; &w = &v; // not allowed, you cannot change the address of a
variable
```

Be careful with pointer assignments

- When you declare a pointer variable, always initialize to the address of some other well-defined variable.
- For example consider: `int *z; *z = 10;`
- This is a dangerous statement that will produce a run-time error because of the following reasons:
    - `z` is intended to contain the address of another variable. But due to lack of initialization, it could contain some arbitrary value.
    - The statement `*z = 10;` attempts to write the value 10 into this arbitrary location. This could produce an error called "segmentation fault".
    - So when you declare `int *z`, just assign the address of another variable (say `q`) to it: `int q; int*z = &q;.`

Uses of Pointers and Referencing

- A function can return a pointer or a reference.
- But we have to exercise care while doing so.
- We will see examples of this later on in this course.
- There are also many uses of pointers in a concept called **dynamic memory allocation** which you will see later in this course.