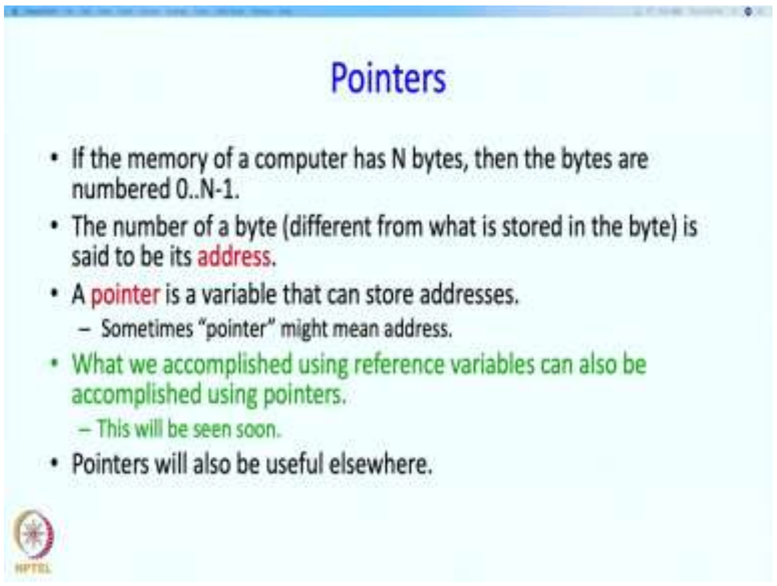
An Introduction to Programing through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture-10
Part – 4
Functions
Pointers

Hello and welcome back. In the last segment we discussed reference parameters. In this segment we are going to talk about pointers which do something similar.

(Refer Slide Time: 00:34)



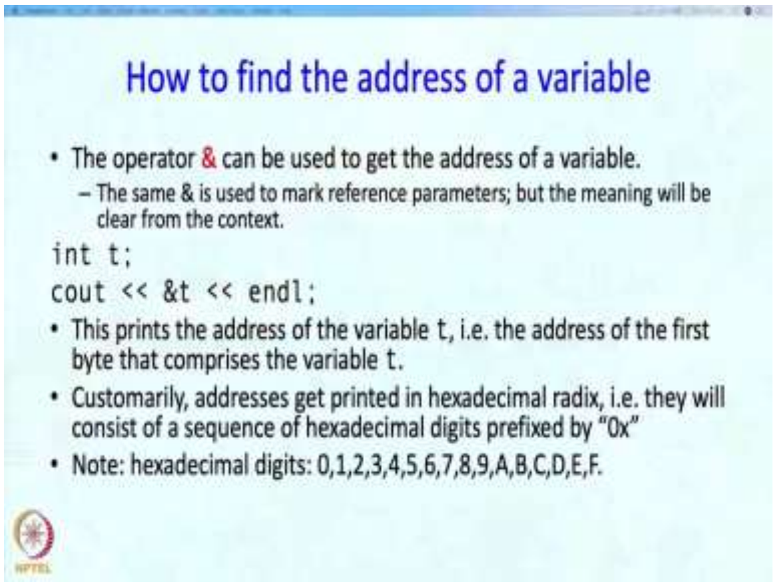
Pointers

- If the memory of a computer has N bytes, then the bytes are numbered 0..N-1.
- The number of a byte (different from what is stored in the byte) is said to be its address.
- A pointer is a variable that can store addresses.
  - Sometimes "pointer" might mean address.
- What we accomplished using reference variables can also be accomplished using pointers.
  - This will be seen soon.
- Pointers will also be useful elsewhere.

So, if the memory of a computer has N bytes, then you may recall that the bytes are numbered 0 through N minus 1 and this number of a byte is different from what is stored in the byte. So, the number of byte is said to be its address. So, if you remember our analogy was the thing of the bytes as sitting on the long street and you give them the number say left to right along the street. So, from one end of the street to the other end of the street and each byte is given a number according to its position on the street and, of course, in the bytes you can store some other values so that is different, so that is different from the address or the number that has been given which is the positon of the byte along that street. So, the number is called the address and, of course, we can store anything in every byte.

The term pointer is used to denote a variable that can store an address. So, an address is also a number but we are going to make a distinction, because addresses are going to be used for very different purposes than ordinary numbers and therefore, we really do not want to mix up addresses and ordinary numerical values. So, in any case you know that everything on a computer is a sequence of bits, so an address is also a sequence of bits, a number whether it a floating point number or a fixed point integer is also a sequence of bits but we have to treat then differently. And so an address is a variable that can store addresses or some times you might use the term pointer to refer to addresses themselves. What we accomplished using reference variables can also be accomplished using the pointers. So, that is the motivation for studying pointer right now and pointers will also be useful elsewhere, so we will study that later.

(Refer Slide Time: 02:55)



How to find the address of a variable

- The operator & can be used to get the address of a variable.
  - The same & is used to mark reference parameters; but the meaning will be clear from the context.

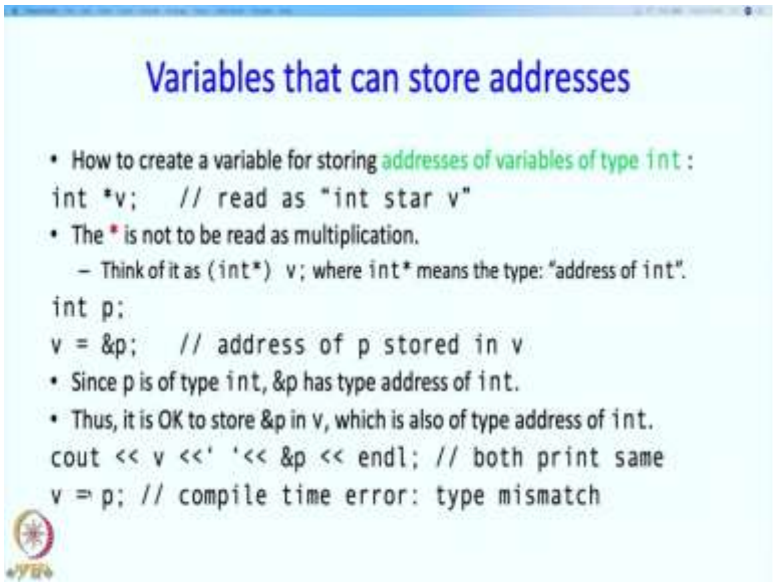
```
int t;
cout << &t << endl;
```

- This prints the address of the variable t, i.e. the address of the first byte that comprises the variable t.
- Customarily, addresses get printed in hexadecimal radix, i.e. they will consist of a sequence of hexadecimal digits prefixed by "0x"
- Note: hexadecimal digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

We said that pointers are variables to store the address of a variable, but let us now see, how do you first get to the address of a variable. So, the operator & can be used to get the address of a variable. Now, the same & is used to mark reference parameters, but which one we are thinking of will became clear from the context. In fact, if you remember we also had an and to denote the conjunction of two conditions. Again it is that same character but in this context we are using a single && so do not confuse it with that and or either the & that is used to mark reference parameters. So, here is a statement which creates a variable t and t. Now, if I write cout<<&t then this is going to print out the address of t not what is contained, what is stored in t that it is going to print out the address of t. So, in particular, so it is an int variable, so you remember that an int variable takes 4 bytes, so this is going to be the address of the very first byte amongst those four bytes. Now, when you do printing like this and you can try it out customarily the radix used for printing addresses is the hexadecimal radix and, in fact, so your value will consist of a sequence of hexadecimal digits which are 0 through 9 and then

A, B, C, D, E and F. And they are also prefixed by 0x, 0x sort of stands for hexadecimal, so if you print this you will see some string looking like that.

(Refer Slide Time: 05:14)



Variables that can store addresses

- How to create a variable for storing addresses of variables of type `int`:

```
int *v;   // read as "int star v"
```

- The * is not to be read as multiplication.
  - Think of it as `(int*) v;` where `int*` means the type: "address of `int`".

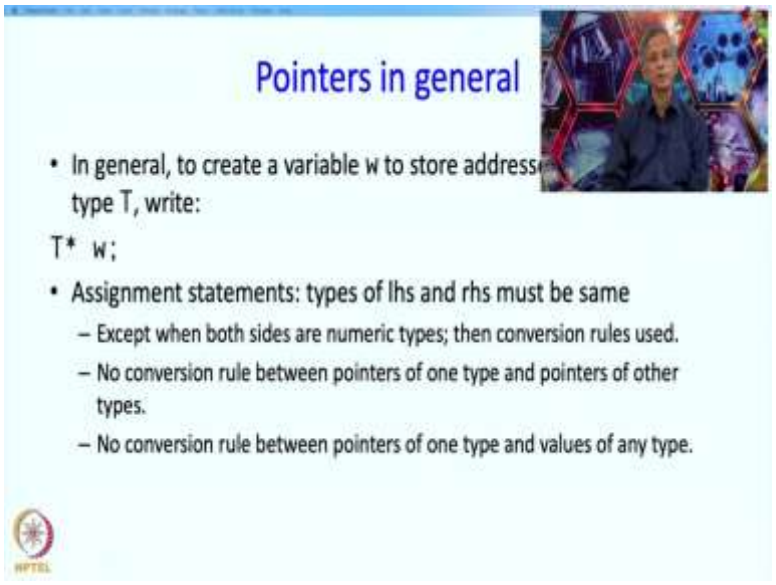
```
int p;
v = &p;   // address of p stored in v
```

- Since p is of type `int`, &p has type address of `int`.
- Thus, it is OK to store &p in v, which is also of type address of `int`.

```
cout << v <<' '<< &p << endl; // both print same
v = p; // compile time error: type mismatch
```

And now we know how to print out address or how to get to address, now we see how we are going to construct variables which can store addresses. So, how do we create a variable for storing addresses of variables of type int, so this is how, so, we are going to write int *v as I just pronounced it this is going to be read or pronounced as int-star-v. And, of course, again there is a confusing usage, the star when used in the manner is not to be read as multiplication, it is meant to be read as something else. So, in fact, we are going to think of the star as joining with int, so we can think of the entire int star as the type of v and you can think of int star to together meaning address of int, so it is a value and the type of that value is address of int. So, suppose we have int p define later on, now I can write v=&p, so what does this do? So v is a variable see supposed it contain addresses and p is an address of int, this is supposed to contain addresses of int and &p is an address of int and therefore, we can store such values inside v. So addresses can be stored, addresses of int can be stored in v. So, in fact, C++ will check whether in fact the values you are storing are of the right type otherwise the compiler will flag an error. Now, if I print out v and print out &p you will see that exactly the same value will get print, so v now is the name of a variable but if I say print me the value of print me v, what gets printed is its value. So, v is not an integer so the value that v contains is an address and so that is, that gets printed, but here I am asking the address itself to get to be printed, so both will really print the same thing. Here is an interesting statement what if I right v is equal to p? Now, v is a variable of type address of p but p is not and address of type address of end, but p is not an address of int, p is an int. So, C++ says that look both of them may be bit strings but I do not think you really want to do this, so there is no occasion where we really want to mix values and addresses and therefore, C++ will flag it as an error during compilation. So, in general we can create pointer to arbitrary types of variables, so not just ints may be doubles, may be chars whatever we want.

(Refer Slide Time: 08:51)



Pointers in general

- In general, to create a variable w to store address
  type T, write:

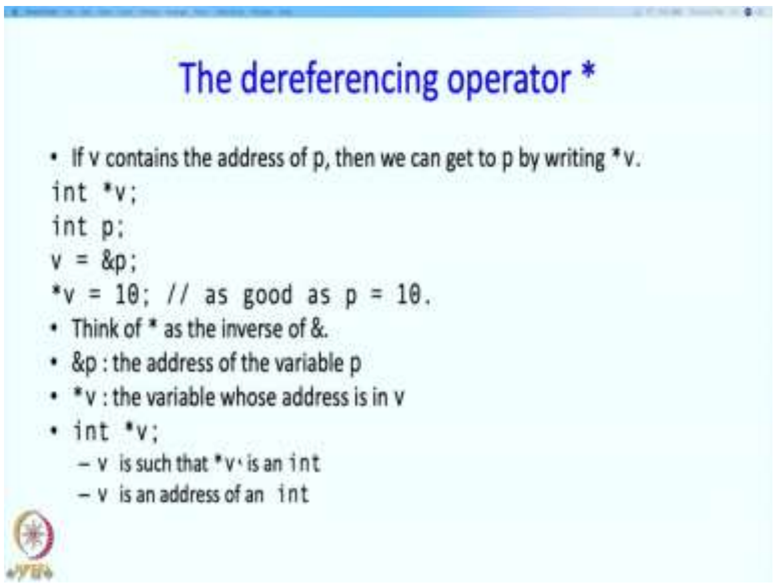
```
T* w;
```

- Assignment statements: types of lhs and rhs must be same
  - Except when both sides are numeric types; then conversion rules used.
  - No conversion rule between pointers of one type and pointers of other types.
  - No conversion rule between pointers of one type and values of any type.

So, to create a variable w to store addresses of variables of type T, we will write T *w, so just as we wrote int star w we can write double *w, double *y or char *x whatever it is, whatever if you want. An assignment statements in general require the types of the left hand side of the assignment and the right hand side assignment to match, so if you create something which has type T * and you assign something to it then the value on the right hand side must also be of type T *.

Now, this type matching rule is not quite applicable if both sides you have numeric types, so we saw that earlier, right, so you can take a float value or a double value and store it into an int or vice versa, but that is an exception made to the very general rule of type matching on both sides of the assignment statement when both sides are numeric, but if both sides are not numeric then there is no automatic conversion made from one type to another. And, in fact, the compiler will flag an error, so there is no conversion rule between pointers of one type to values of another type or pointers of one type and pointers of other types.

(Refer Slide Time: 10:23)



The dereferencing operator *

- If v contains the address of p, then we can get to p by writing *v.

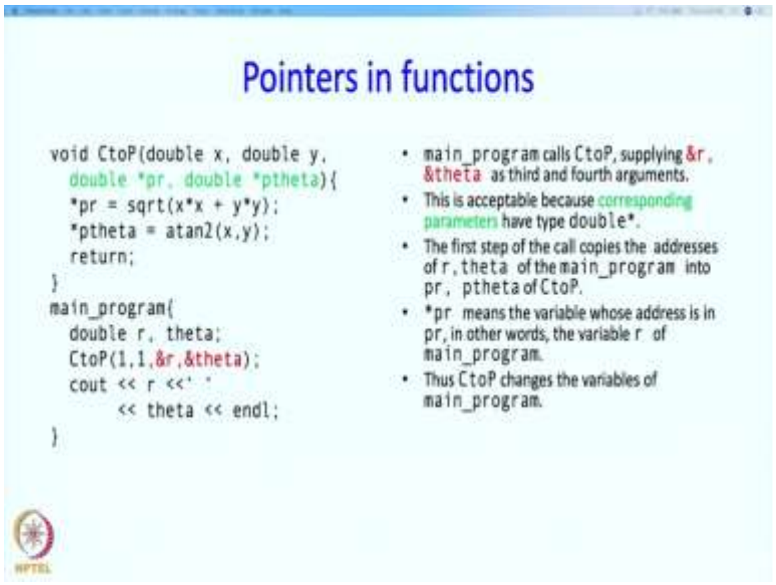
```
int *v;
int p;
v = &p;
*v = 10; // as good as p = 10.
```

- Think of * as the inverse of &.
- &p : the address of the variable p
- *v : the variable whose address is in v
- `int *v;`
  - v is such that *v is an `int`
  - v is an address of an `int`

So, we have the address of operator and we also have the dereferencing operator and the dereferencing operator is *, so again they should not be confused with multiplication. So, if v contains the address of p, then we can get to p by writing *p, so here is an example, so int *v declares v to be a variable of type address of int, then int p says p is just a variable of type int, then we can write the assignment v=&p, so v now contains the address of p. And, now, if I write *v, so * is the dereferencing operator. What that says is this entire expression should be thought of as referring to the variable whose address is contained in p, well so which variables address contained in p? p itself, in v the address of p and therefore, star v should be thought of as p and therefore, this entire statement should be thought of as p is equal to 10.

So these stars and ands are really very simple, very straight forward, but they are not the same, they are, in fact, inverses of each other. So, one says, so and says give me the address of this variable, * says give me the variable whose address is in this variable, so and p is the variable of p and *v is the variable whose address is in v. So, this gives us a slightly different way of thinking about the definition of v, so the definition of v is in *v. So, sometime ago we said that we can think of this definition as v is an address of an int, so this is what we said earlier, the second line, v is an address of int because we think of int * as an atomic term and int * we think of as an address of int, so here we are declaring v to be address of int. But there is a different way of thinking about this as well which says that, “This is not really a declaration of v directly, it is a declaration of what is contained in v.” So, what is contained in v? ints and therefore, v is of type address of int, perhaps the more accurate way of viewing this is that this is a declaration of *v or whatever is contain inside v. So, sometimes this view is little bit more important and we will see a use for that, but both are sort of good but this view that, the first view written over here is slightly better for a reason. So, now we are going to see a use of pointers when we write functions.

(Refer Slide Time: 14:00)



Pointers in functions

```
void CtoP(double x, double y,
  double *pr, double *ptheta){
  *pr = sqrt(x*x + y*y);
  *ptheta = atan2(x,y);
  return;
}
main_program{
  double r, theta;
  CtoP(1,1,&r,&theta);
  cout << r <<' '
       << theta << endl;
}
```

- main_program calls CtoP, supplying &r, &theta as third and fourth arguments.
- This is acceptable because corresponding parameters have type double*.
- The first step of the call copies the addresses of r, theta of the main_program into pr, ptheta of CtoP.
- *pr means the variable whose address is in pr, in other words, the variable r of main_program.
- Thus CtoP changes the variables of main_program.

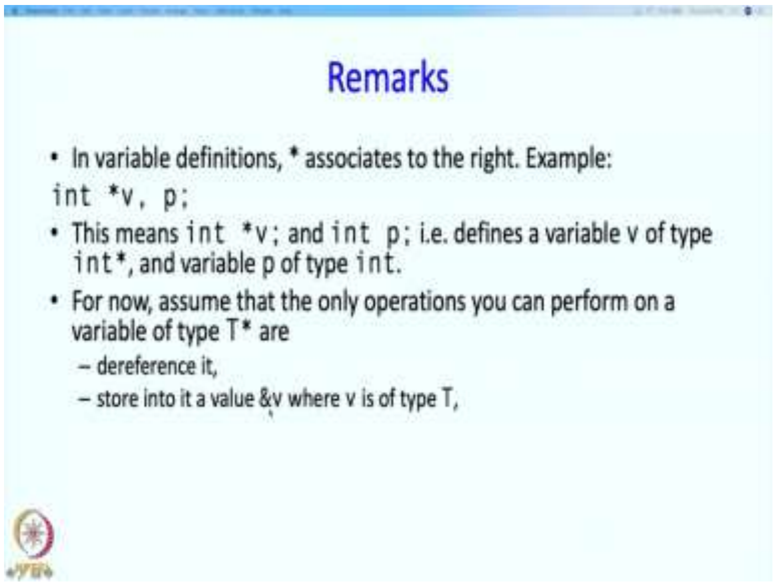
So we are going to write the function which converts Cartesian to polar, so it is going to receive the Cartesian coordinates by value in this parameters x and y, but in addition it is going to have two parameters pr and ptheta so pr has type double * or it is an address of double value and similarly, ptheta is the address of a double value. So, let me just write down the function first and then I will explain.

So, what happens over here, so the main program contains the statement which has address of r being passed and address of theta also being passed, so what is this? So, first of all C++ requires that whatever type you have over here had better match the type of the parameter over here, so what is the type of this parameter pr? So, it address of double, so that is what this double stars say that pr has type address of double. And this, in fact, is address of double, so this type matches, this says ptheta is address of double and, in fact, and theta is the address of theta which is a double variable, so the pointers are the types of the arguments are matching. So, what happens, so the types are matching, so the call will copy the values 1 into x and 1 into y but it will copy the address of r into pr and address of theta into ptheta. So, now something interesting has happened, CtoP which is executing in its own activation frame has received addresses of variables defined in some other activation frame, in the activation frame associated with the main program. So, when I execute this what is going to happen, what does this mean? The rule is as before, so *pr means that variable whose address is in pr.

Now, when this when I do this, this variable is not living in the activation frame of C to P, but I have its address and I can dereference, I can use the dereferencing operator and therefore, this entire expression really now will mean the r variable which is appearing in the main program in the activation frame of the main program, so it will refer to this variable. So whatever value we calculate over here will get stored, actually stored in this variable of the main program.

Similarly, ptheta is the address of some variable well it's the address of this variable theta because that is what we passed over here and when we do star of ptheta this expression is the same as this variable and therefore, atan2 of x, y is going to get stored in this variable theta. So, as a result, so in this case 1, 1 so square root of two in the in r because of this statement and atan2 is going to be placed in theta because of the statement. So, when the call finishes what you are going to get is one is going to get square root 2 over here and pi by 4 over here or 45 degrees over here, as we got when we used references parameters. So, again CtoP is going to change the variables of the main program but the mechanism is slightly different here we have been we have send them the address and there we sort of say that look you treat your r and theta to mean my variables. Where as over here we have actually send then the address, it like saying to somebody look I am not going to send you the value, I am not going to send you the money, I am going to send you the address of the bank from where you can get the money or you can, I am going to send you the address of the bank where you can deposit the money, so that is what is happening in this function and that is what pointers enable you to do. So this will exactly do what we wanted.

(Refer Slide Time: 19:20)



Remarks

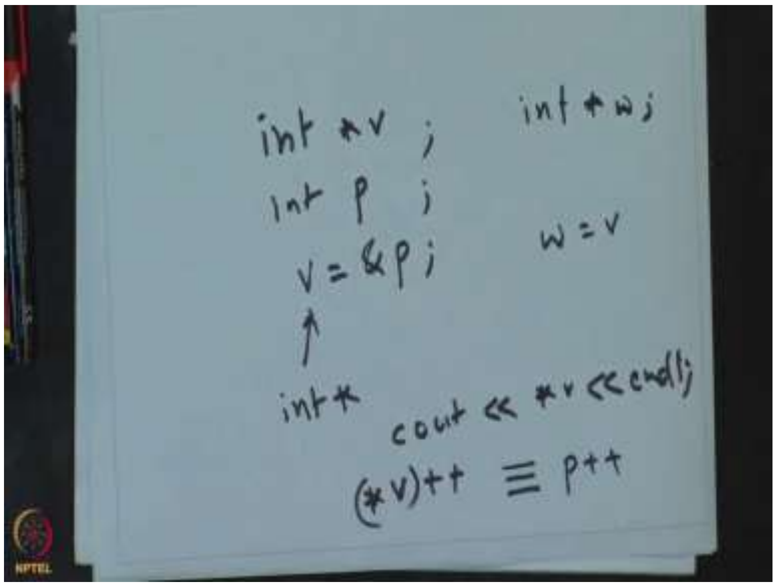
- In variable definitions, * associates to the right. Example:

```
int *v, p;
```

- This means `int *v;` and `int p;` i.e. defines a variable v of type `int*`, and variable p of type `int`.
- For now, assume that the only operations you can perform on a variable of type T* are
  - dereference it,
  - store into it a value &v where v is of type T,

So there is a remark to be made which is that star in definitions has a slightly unexpected behaviour, so the star actually associates to the right, so suppose a right int *v, p and inclined to read this as that the star sort of I read it as int * and then v is int * and p is also int *. That could be one way C++ could have define this but that is not C++ ha defined it and the definition that C++ has is perhaps slightly unfortunate but it is there. So what is the definition? So it says that we think of this as is the type of the variable which are contained in v, not the values that are contained in the v but is the type of the variable that contained in v, so basically it does say that v contains addresses of int variables but it is saying that in a roundabout way. So int is, this whole thing is an int and similarly when I write this, this p is an int, so this means int *v and int p, not int *v and int *p because this star sort of binds to v and not to int. And for now assume that the only operation you can perform on a variables of type T * or an address of something are that we can dereference it, so I can write *v if I have a variable of type address of int, or I can store into it a value address of v. So, you an example of this earlier.

(Refer Slide Time: 21:46)

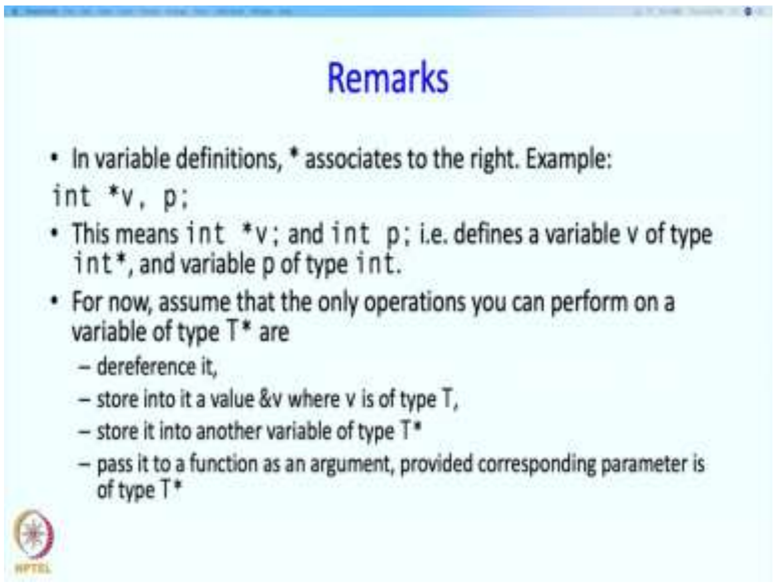


```
int *v ;      int *w;
int p ;
 v = &p;      w = v
 ↑
int*
       cout << *v << endl;
       (*v)++  ≡  p++
```

So, I can have int *v and int p and I can say v=&p, so this is a variable of type int star and what I can do with it is, I can store a value into it which is address of some int variable, so this is what I can so with it or I can say cout<<*v or I can say *v++, what would this do? If v has being said to be address of p this is going to be the same thing as p++.

I can so one more thing, I might have int *w in addition to int *v, so in that case I can write w is equal to v, so whatever variable address was in v now is also stored in w. So this is completely consistent but you may find it unusual, simply because you are not accustomed to dealing with pointers and addresses and dereferencing addresses. But think of it one line at a time and it should be quite clear to you, so look at every expression, look at what the definition that is and if you do this a few times it will be became second nature.

(Refer Slide Time: 23:39)



Remarks

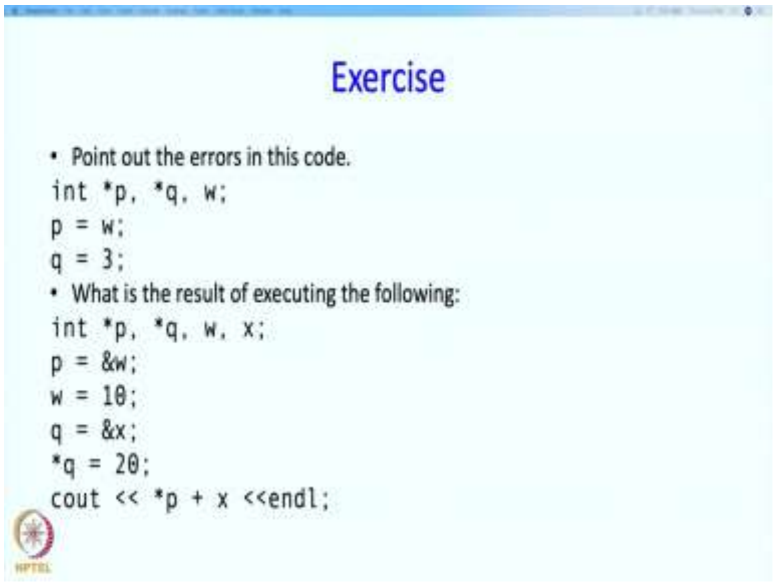
- In variable definitions, * associates to the right. Example:

```
int *v, p;
```

- This means `int *v;` and `int p;` i.e. defines a variable v of type `int*`, and variable p of type `int`.
- For now, assume that the only operations you can perform on a variable of type T* are
  - dereference it,
  - store into it a value &v where v is of type T,
  - store it into another variable of type T*
  - pass it to a function as an argument, provided corresponding parameter is of type T*

You can pass it to a function as an argument provide the corresponding parameter is of the right type that T *, so if you have a variable of T *, then you can pass it as a parameter to a function in which as an argument to a function if the corresponding parameter is of type T *.

(Refer Slide Time: 24:05)



Exercise

- Point out the errors in this code.

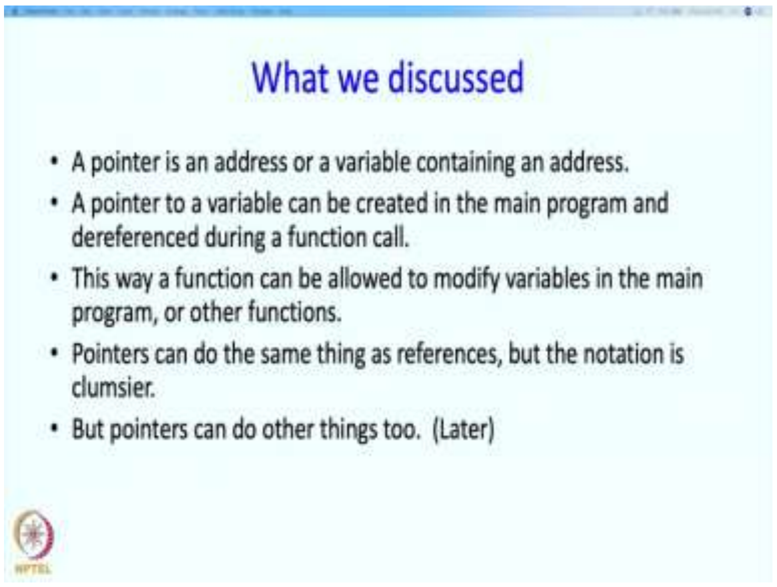
```
int *p, *q, w;
p = w;
q = 3;
```

- What is the result of executing the following:

```
int *p, *q, w, x;
p = &w;
w = 10;
q = &x;
*q = 20;
cout << *p + x <<endl;
```

So, now here are some exercises for you. You have to point out the errors in this code and you are also to tell what gets printed as a result of executing this code. So what did we discuss in this segment?

(Refer Slide Time: 24:21)



What we discussed

- A pointer is an address or a variable containing an address.
- A pointer to a variable can be created in the main program and dereferenced during a function call.
- This way a function can be allowed to modify variables in the main program, or other functions.
- Pointers can do the same thing as references, but the notation is clumsier.
- But pointers can do other things too. (Later)

So we said that a pointer is an address or a variable containing an address. A pointer to a variable can be created in the main program and dereferenced during a function call. So, this way a function can be allowed to modify variables in the main program, or the variables in other function as well however, wherever we create the pointers and if we pass it, pass to other functions.

Pointers in some sense are doing the same thing as references, but the notation is slightly more means is clumsier in the sense you have to explicitly say that now dereference it, whereas in the case of a reference you declared at the beginning that 'Oh this name is just going to be in this other name,' whereas here the name does not really mean the other name, this name is means the address of this name and then you have to say that, "Ok, I want to get to the name, the variable itself whose address I have and therefore, you have to use that dereferencing operator."

So, pointers can do what references can do, but they can do other thing as well and we will see all of that later. So, we will take a break here.