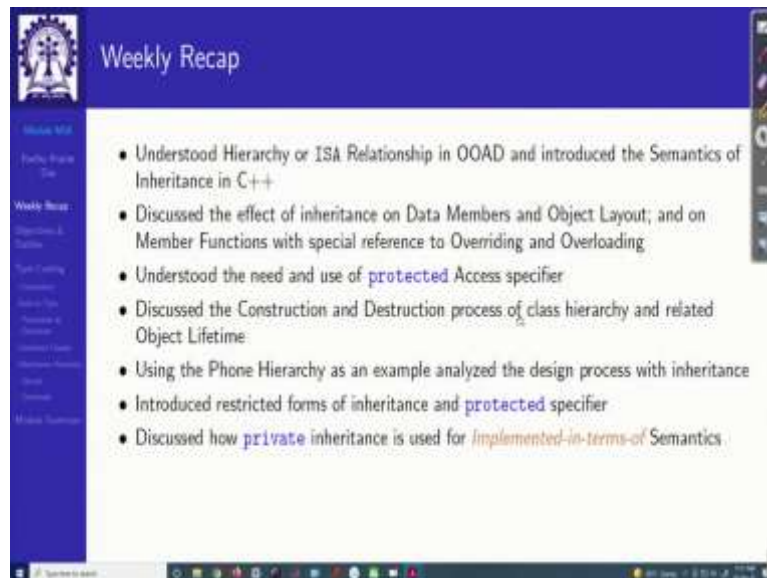
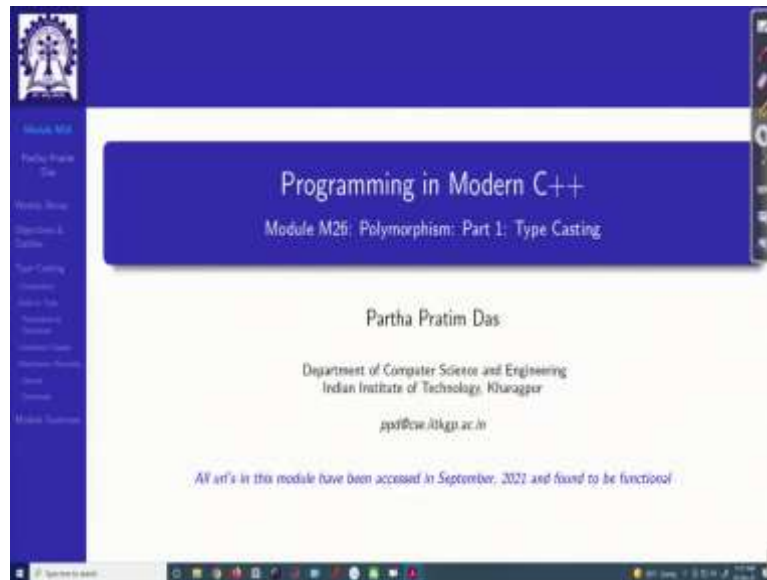


**Programming in Modern C++**  
**Professor Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture - 26**  
**Polymorphism: Part 1: Type Casting**

(Refer Slide Time: 00:37)

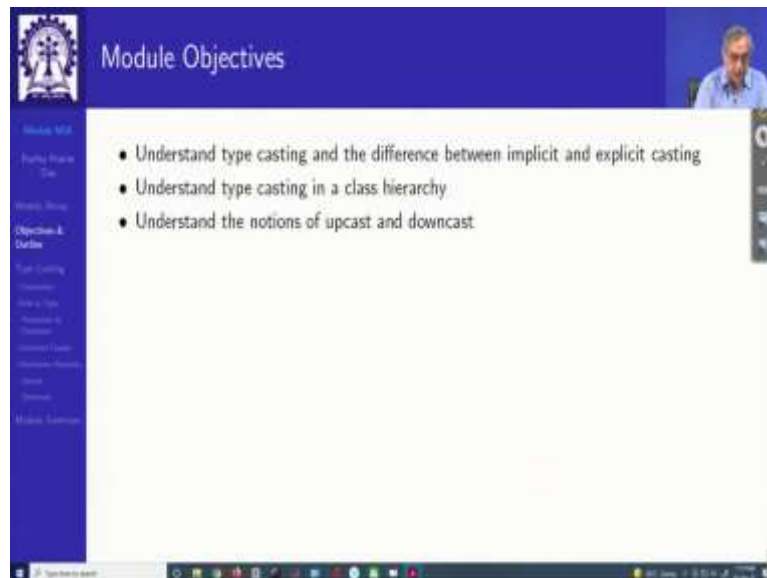


Welcome to Programming in Modern C++. We are starting week 6, with module 26. In the previous week, we took a look at the inheritance mechanism of C++ which is the way to realize is a relationship or a hierarchy of generalization specialization in C++.

So, we have considered different aspects of that how to define the hierarchy. What happens to the objects when they are constructed, they are destructed, their lifetime. Particularly taken look at method overriding and method overloading processes and so on. So, this gives a solid

foundation to the object oriented, second object oriented aspect of C++ which is about inheritance beyond the encapsulation that we had seen earlier.

(Refer Slide Time: 1:38)

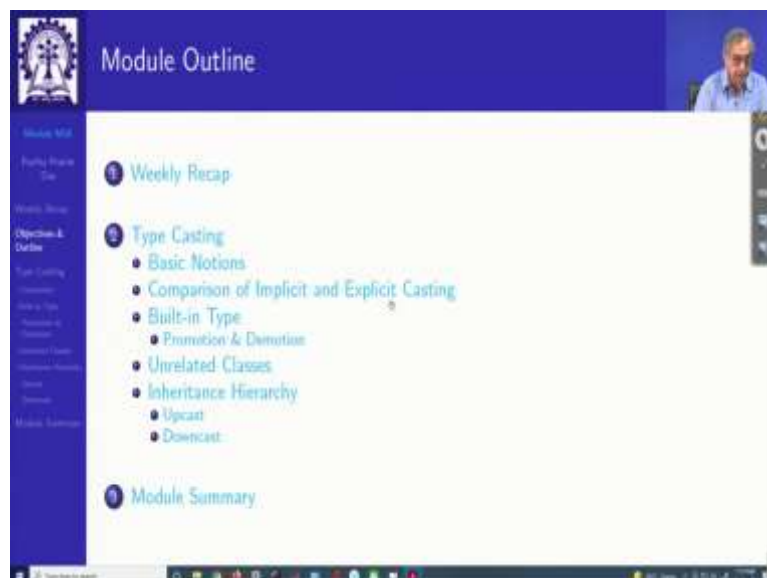


The screenshot shows a presentation slide titled "Module Objectives" with a blue header and a white content area. A small video feed of a speaker is visible in the top right corner. The slide lists three bullet points:

- Understand type casting and the difference between implicit and explicit casting
- Understand type casting in a class hierarchy
- Understand the notions of upcast and downcast

In this week, starting with the current module we will primarily take a look at polymorphism or as it is said static and dynamic binding on a hierarchy or even outside of that. This will be the third major aspect of object oriented programming which gives the foundation to the efficient use of C++ as an object oriented language. Starting with that, we will first take a look at what is type casting? What is the difference between implicit and explicit casting? How casting can work on a hierarchy and will introduce the notions of up cast and down cast on a hierarchy.

(Refer Slide Time: 2:30)



The screenshot shows a presentation slide titled "Module Outline" with a blue header and a white content area. A small video feed of a speaker is visible in the top right corner. The slide lists the following topics:

- 1 Weekly Recap
- 2 Type Casting
  - Basic Notions
  - Comparison of Implicit and Explicit Casting
  - Built-in Type
    - Promotion & Demotion
  - Unrelated Classes
  - Inheritance Hierarchy
    - Upcast
    - Downcast
- 3 Module Summary

# Type Casting

## Type Casting: Basic Notions

- Casting is performed when a value (variable) of one type is used in place of some other type. Converting an expression of a given type into another type is known as **type-casting**.

```

int i = 3;
double d = 2.5;
double result = d / i; // 1 is cast to double and used
  
```

*Handwritten notes:* "mixed mode" with arrows pointing to the variables in the code above. "Convert to double" with an arrow pointing to the division operation.

- Casting can be **implicit** or **explicit**

```

d = i; // implicit: int to double
i = d; // implicit: warning: loss of precision from 'double' to 'int': possible loss of data
d = (double)i; // explicit: int to double
i = (int)d; // explicit: double to int
  
```

- Casting Rules can be grossly classified for:
  - Built-in types
  - Unrelated types
  - Inheritance hierarchy (static)
  - Inheritance hierarchy (dynamic)

## Type Casting: Basic Notions

- Casting is performed when a value (variable) of one type is used in place of some other type. Converting an expression of a given type into another type is known as **type-casting**.

```

int i = 3;
double d = 2.5;
double result = d / i; // 1 is cast to double and used
  
```

- Casting can be **implicit** or **explicit**

```

d = i; // implicit: int to double
i = d; // implicit: warning: loss of precision from 'double' to 'int': possible loss of data
d = (double)i; // explicit: int to double
i = (int)d; // explicit: double to int
  
```

*Handwritten notes:* "int double" with arrows pointing to the variables in the code above. "<type>" with an arrow pointing to the cast operator in the code above.

- Casting Rules can be grossly classified for:
  - Built-in types
  - Unrelated types
  - Inheritance hierarchy (static)
  - Inheritance hierarchy (dynamic)

So, this is the outline which will be available on the left panel as usual. So, talking about type casting, I am sure all of you know type casting from C in a very, I should say compact way. So, casting is performed when a value or variable of one type is used in place of some other type. So, when we want to do that then naturally we need to convert an expression in general which could be a simple value literal, a variable or an expression involving operators which is of one type and we take it to another type, we say that is typecasting.

So, here is an example there is an int variable and there is a double variable initialized and we are trying to define a result which is the division of d by i, that is a double value by an integer value. You know in C this is called mixed mode operation. We will not use those terms frequently now because we will generalize to a much bigger context.

So, the basic issue here is this is double and this is integer. Since the division in the system can either be of integer type that is an integer divided by another integer or of double type, floating point type where a floating point value is divided by another floating point value. This is what turns out to be as we say is a mixed mode. That is it involves a floating point value with an integer value.

So, what we will need to do, what the compiler needs to do? Since it cannot do this division, it will have to convert this integer value to double. Now obviously we will ask the question as to why i is converted to double for the division? Why not d is converted to int for the division? That is to preserve as much of information as possible and will look at those rules later. Now as you can see, as you have known in the C language that such conversions are implicit in the sense that you did not have to say that you convert the integer to the double, the mixed mode automatically supports that.

So, that brings us to the first basic notion about casting that it can be implicit or it can be explicit, consider a few examples. In the context of the above definition, suppose I try to assign i to d that is an integer to a double, this is implicit because if I assign i to d the value of i which is an integer will have to be converted to the value in a double format.

So, if the value of i is 3, it will have to become 3.0, it has to take the format of the floating point number and so on and this is allowed implicitly, I am not explicitly saying that the compiler knows that the int is being assigned to a double and it will allow that.

I can do the reverse also. Assigning d to i that is trying to convert double to int, this also implicitly be converted but the compiler will give a warning. GCC will give a warning

because when you convert a double value to an integer, you may not have a correct representation. For example if you convert 2.5 to a integer, it will possibly become 2. So, you are losing information. So, there is possible loss of data. The compiler allows you but not silently, it will give you a warning.

In contrast, you can do these things if you do not want the compiler to advise you or if you really know what you are doing, then you can write explicitly that I want the integer `i` to be converted to double and this is the type of casting that you have seen in C or you can do the reverse in either case the compiler will allow it and will allow it silently because it knows that as a programmer you know what you are doing. That is the basic assumption.

So, when we write like this where inside this pair of parentheses there is a type given it is considered that it is an instruction to the compiler to convert the value from the given type which can be obtained from `i` or from `d`. This is `int` or this is `double` to the type that you have specified and this is known as a cast operator. This is more specifically known as C style casting.

So, we will in C++, this will get into lot more of depth. So, what we will do we will look at the different casting rules in certain groups. First we will look at the casting for the built in types, then we will look at casting for unrelated types including classes, you know C++ deals primarily with classes. And then we will talk about what happens to casting when we have an inheritance hierarchy. What happens at the compiled time and what can happen at the runtime? So, there is casting all over. So, these different aspects are what you will see in the next couple of slides.

(Refer Slide Time: 8:26)

Implicit Casting	Explicit Casting
<ul style="list-style-type: none"><li>• Done automatically ✓</li><li>• No data loss, for promotion ✓ Compiler will be <i>silent</i></li><li>• Possible data loss, for demotion ✓ Compiler will issue <i>warning</i></li><li>• No throwing of exception – is <i>type safe</i></li><li>• Requires no special syntax</li></ul>	<ul style="list-style-type: none"><li>• Done programmatically ✓</li><li>• Data loss <i>may or may not take place</i> Compiler will be <i>silent</i></li><li>• May throw for wrong type casting ✓</li><li>• Requires cast operator for conversion C style operator: (&lt;type &gt;) ✓ C++ style operators: const_cast, static_cast, dynamic_cast, and reinterpret_cast</li><li>• Avoid C style cast ✓ Use C++ style cast</li><li>• Possible in static as well as dynamic time ✓</li><li>• May be defined for User-Defined Types</li></ul>

Now before we proceed further, I would just like to highlight the basic differences between implicit and explicit casting, you may find this, you may want to revisit this slide after you have understood the casting well. So, in implicit casting as you have seen it is done automatically by the compiler but explicit casting has to be done programmatically by the developer.

In implicit casting there is no loss of data, if you are promoting that is taking a smaller sized data type to a bigger size data type and compiler will be silent. But if you do the reverse as you have seen, so for the case of int to double compiler was silent. If you do the reverse, if you do a demotion that is from a bigger size to a smaller size naturally in the smaller size data type you will not be able to fit all the information when you convert a double to an int, the compiler will issue a warning.

For explicit casting, there may or may not be loss of data int to double. Possibly there will be, there will not be loss of data, double to int there will be lots of data in most cases compiler will always be silent because it knows that the developer is knowingly doing this. Implicit casting does not throw an exception you would still have not, we have not done exception so you will see that more when we do that and therefore it is type safe whereas explicit casting may throw exception.

In implicit casting as you have seen, there is nothing specific that you are writing down so there is no specific syntax but for explicit casting it requires cast operator either C style as we have just discussed or as we will see that there are four specific operators provided in C++ const\_cast, static\_cast, dynamic\_cast and reinterpret\_cast which allow us to do a casting in

the proper semantic manner in any of the, between any two different types, it will let you do the casting or will through exception, error compile time error and so on to let you know what you are actually doing and whether it is acceptable.

Implicit casting it is advisable that you avoid because it cannot be easily figured out from the program you will have to really think what is happening like when we do divided by i, it is not written, nothing is written. So, if you avoid that do explicit casting when you need to do then anybody reading the program would find it easier to follow and even in that I would advise and that's what the community strongly says that in C++ avoid C style casting, do not do it unless it is absolutely necessary but such situations will not occur if you are using C++ casting properly always use C++ style casting.

Implicit casting is possible only at the static time, compile time as we will see explicit casting can be both at the static as well as dynamic time and implicit casting may be disallowed for user defined types. There are ways to do that, we will see that but it cannot be disallowed for the built in types. For explicit casting you can define the way you want the casting to happen for the user defined types. So, these are kind of the basic differences between the casting two forms of casting that we have.

(Refer Slide Time: 11:56)

**Type Casting Rules: Built-in Type**

- Various type castings are possible between built-in types

```
int i = 3;
double d = 2.5;
double result = d / i; // i is cast to double and used
```

- Casting rules are defined between numerical types, between numerical types and pointers, and between pointers to different numerical types and void
- Casting can be **implicit** or **explicit**

```
int i = 3;
double d = 2.5, *p = &d;

d = i; // implicit: int to double
i = d; // implicit: casting: '*' ; conversion from 'double' to 'int' - possible loss of data

d = (double)i; // explicit: int to double
i = (int)d; // explicit: double to int

i = 0;
i = (int)*p; // explicit: double * to int
```

So, let us move on to the, discussing the rules for the built-in type. This is the example you have already seen. We have seen this, these are implicit casting, these are explicit casting but not everything can be explicitly cast either. For example, I have a pointer p, which is a pointer to a double.

Now, if I want to assign that to an integer, I will get an error, there will be an error because a pointer is not implicitly allowed to be treated as an integer but explicitly I will be able to do this kind of a conversion. Why we, at all we will do that is something we will discuss subsequently but this is the basic rules of casting in terms of the built in types.

(Refer Slide Time: 12:48)

**Type Casting Rules: Built-in Type: Numerical Types**

- Casting is **safe** for **promotion** (All the data types of the variables are upgraded to the data type of the variable with larger data type)

bool → char → short int → int → unsigned int → long → unsigned → long long → float → double → long double

- Casting in built-in types **does not invoke** any conversion function. It only **re-interprets** the binary representation
- Casting is **unsafe** for **demotion** – may lead to loss of data

Now built in types if you consider then there are primarily you will see there are primarily two kinds of types. One is numerical types which are all of these some of which are integral like this, this, this these are integral whereas float, double, long double these are of floating point type. Now casting is safe if you do promotion. That is if you take a smaller sized data type to a larger size data type, so this arrow chain shows what is the chain of promotion that is safe.

So, this promotion implicitly or explicitly among numerical types will usually be safe. The casting of built in type does not invoke any conversion function. You are converting 2.5 to say 2 or you are converting 3 to 3.0. So, naturally there are bit patterns which has to change but it does not happen through invocation of a function which we will subsequently see. It just reinterprets kind of things whatever is written here as a double 2.5, it just thinks that it is integer 2 and creates the bit pattern for that. Casting will be unsafe if you do demotion. For example, if you take a double into an int, smaller size data type it will be naturally unsafe, we have already seen examples therein.



(Refer Slide Time: 14:20)

**Type Casting Rules: Built-in Type: Pointer Types**

- Implicit casting between different pointer types is not allowed ✓
- Any pointer can be implicitly cast to void\* (with loss of type); but void\* cannot be implicitly cast to any pointer type
- Conversion between array and corresponding pointer is not type casting – these are two different syntactic forms for accessing the same data

```
int i = 1, *p = &i, a[10]; double d = 1.1, *q = &d; void *r;

q = p; // error: cannot convert 'int*' to 'double*' ✓
p = q; // error: cannot convert 'double*' to 'int*' ✓
q = (double*)p; // Okay ✓
p = (int*)q; // Okay ✓

r = p; // Okay to convert from 'int*' to 'void*' ✓
q = r; // error: invalid conversion from 'void*' to 'int*' ✓
p = (int*)r; // Okay ✓

p = &i; // Okay by array pointer duality. p[i], a[i], *(p+i), *(a+i) are equivalent
a = p; // error: incompatible types in assignment of 'int*' to 'int[10]'
```

Now, the other type which is dominant as a built in type is a pointer type, pointed to different types. Let us say we restrict to only pointers to numerical types. So, implicit casting is not allowed. You cannot have implicit casting so if you have a integer pointer here and a double pointer here, then you are not allowed to cast either into the other. Mind you, the integer or double can directly be cast but their pointers are not implicitly castable.

You will have to cast with an explicit intention of the cast operator. Any pointer can be implicitly cast to void\*, here I have a pointer void\* which means that it is a pointer but not known to which type. You can convert any pointer, for example p, pointer to integer to void\* implicitly, compiler will not complain because when you do that you are not making any additional assumption. You knew that this is pointing to an integer, you are just choosing to forget that. But if you try to do the reverse that is void\* to int\*, the compiler will give an error because obviously by saying that r is void\*, you are saying that it points to something I do not know.

So, you cannot implicitly say convert it to integer saying that it is now pointing to an integer, if you have to do that then you will have to explicitly say that, so, this is the second main point. The third which might arise to your mind is, we have often used array and pointer I mean one in the place of the other. You must have seen this in C quite often. This is called array pointer duality particularly important for multi-dimensional arrays.

So, this conversion is not actually a type casting. Please keep that in mind, it is just a syntactic form of writing the same intended expression in multiple syntax. For example, if I have an array a, 10 of integers and I have an integer pointer p, then I can assign a to p

because it is a address, the base address of the array. I can assign that to p and then I can use pi, ai, \*p plus i, \*a plus i, all these are equivalent of accessing the ith element. So, this is not a case of conversion.

If you try to do the reverse, if you try to assign a pointer to the array base address, you will get some kind of an error like this, the basic reason that you get the error is the base address of an array is a constant. It is a constant pointer. So, we will come to the role of const-ness in this but simply you can remember that you cannot change the base address of an array. So, you cannot assign a pointer of the same type to the base address and you will get an error.

(Refer Slide Time: 17:39)

**Type Casting Rules: Built-in Type: Pointer Types**

- Implicit casting between pointer type and numerical type is *not allowed*
- However, explicit casting between pointer and integral type (`int` or `long` etc.) is a common practice to support various tasks like *serialization* (save a file) and *de-serialization* (open a file)
- Care should be taken with these explicit cast to ensure that the integral type is of the *same size* as of the pointer. That is: `sizeof(void*) = sizeof(< integraltype >)`

```
int i, *p = 0; long j;

// sizeof(i) = sizeof(int) = 4
// sizeof(j) = sizeof(long) = 8
// sizeof(p) = sizeof(int*) = sizeof(void*) = 8

i = p; // error: invalid conversion from 'int*' to 'int'
p = i; // error: invalid conversion from 'int' to 'int*'

i = (int)p; // error: cast from 'int*' to 'int' loses precision
p = (int*)i; // warning: cast to pointer from integer of different size

j = (long)p; // Okay
p = (int*)j; // Okay
```

• Here, the conversion should be done between `int*` and `long` and not between `int*` and `int`

Now, this is about the conversion between pointer types. Now, if you want to do a conversion between pointer type and numerical type the basic rule it is not allowed. But even there are situations where explicit casting between pointer and an integral type, I mean not a floating point type but an integral type like `int`, `short int`, `long`, `long long` and those kind of are the common practice and actual is the only way to do certain tasks like serialization and deserialization.

You may not be familiar with these terms, what does it mean is when you store a file, save a file like you are editing a document in word and you say, you do control s and save the file you are actually serializing from the in memory data structure of the document to a linear sequential file in the system. Similarly, deserialization is a reverse process when you load or open a file.

So, in this context, I mean maybe later we will try to explain why you need this kind of conversion but certainly the basic intuition is when you have, when you are editing your document you have a data structure in the system which has pointers.

So, when you write it to the system to save it as a file, you need to write those pointer addresses but those pointer addresses are not going to be available next time you open the file. So, you need to convert them in some form of binary representation which is best done in terms of an integral type.

Now when you do this, you must be careful that your integral type is large enough to fit the pointer address. Typically it is advised that you always choose an integral type which is the same, which has the same size as of the pointer type. Just as an example, here I show, I mean this code I tried in online GDB. So, we can see that I have an integer value, a pointer to an integer and a long value `j` and if I look at the size, you will see these are the size that is the integer is of 4 bytes whether the pointer is of 8 bytes because it is a 64 bit system.

So, the long as well as the pointer are of 8 bytes. So, they are compatible but this is not compatible with `int`, the size of `int`. So, the first thing is if you just try to do an assignment between the integer pointer and the integer and otherwise you will obviously get errors because they are not allowed.

If you try to, now convert the point at `p` to `i` with an explicit cast of `int`, you will still get an error. Why would you get an error? Because `p` is of size 8 bytes and `i` is of size 4 bytes. So, it cannot and losing this information for a pointer is absolutely critical, it is not like reducing precision only from 2.5 to 2 but it basically you have lost the address. So, this is not allowed.

Even the reverse, if you want to do, convert an integer value `i` to an integer pointer by explicit conversion well the compiler will allow it to do but it will give you warning because the value that you have in `i` is a 4 bytes and the value that you expect in `p` is of 8 bytes. So, certainly when you do this conversion, there are 4 bytes of you do not know what kind of values. So, to do this properly what you can do is, you can use the long type in this case. For example, now if you explicitly cast, the pointer to integer to long then the compiler will not say anything because it is ok, both have size of 8 bytes.

Similarly, the reverse that is taking a long and casting it to a pointer of integer type is also ok. So, I mean I took a specific case where the integer and the pointer types are of different size in your machine, they could be of the same size in your compiler, they could be of the same

size. It does not matter what that reality is, all that you have to ensure is you do this kind of a conversion between an integral type and a pointer only based on the fact that they have equal size. So, this is some of the nuances of conversion in the pointer type and we close the built in type rules here.

(Refer Slide Time: 22:29)

**Type Casting Rules: Unrelated Classes**

- **(Implicit)** Casting between *unrelated classes* is not permitted ✓

```
class A { int i; };
class B { double d; };

A a;
B b;

A ap = &a;
B bq = &b;

a = b; // error: binary '=' : no operator which takes a right-hand operand of type 'B'
a = (A)b; // error: 'type cast' : cannot convert from 'B' to 'A'
b = a; // error: binary '=' : no operator which takes a right-hand operand of type 'A'
b = (B)a; // error: 'type cast' : cannot convert from 'A' to 'B'

p = &a; // error: 'A*' : cannot convert from 'B*' to 'A*'
q = &b; // error: 'B*' : cannot convert from 'A*' to 'B*'

p = (A*)&b; // explicit on pointer; type cast is okay for the compiler
q = (B*)&a; // explicit on pointer; type cast is okay for the compiler
```

Now, let us look at typecasting between unrelated types particularly unrelated classes. This is not permitted, if I have just two classes A and B, I have two objects, I have two pointers to these objects and then I have tried to do all kinds of mixed stuff, that is assigning the one object to another not allowed. Trying to explicitly cast it is not allowed because there is no conversion available.

Similarly, the other way these are not allowed, I try to cast the pointers they are not allowed. The only thing that is allowed is, I can cast the pointer if I explicitly cast it. So, here you can see that I am explicitly casting the address of b which is of type B\* to the type A\* but it could be very very risky. So, this is the basic ground rules for unrelated classes.

(Refer Slide Time: 23:49)

The slide content is as follows:

```
• Forced Casting between unrelated classes is dangerous
```

```
class A { public: int i };
class B { public: double d };

A a;
B b;

a.i = 5;
b.d = 7.2;

A *p = &a;
B *q = &b;

cout << p->i << endl; // prints 5
cout << q->d << endl; // prints 7.2

p = (A*)b; // Forced casting on pointer: Dangerous
q = (B*)a; // Forced casting on pointer: Dangerous

cout << p->i << endl; // prints -88888888 GARBAGE
cout << q->d << endl; // prints -9.25896e+061 GARBAGE
```

So, what you can do is certainly as we have done by the explicit casting. We can forced this cast to take place but this could be really dangerous. So, the same example, we have put some values it has an integer i, it has a double d. So, we have put respective values put two pointers and now I print this with p pointer i, I get the correct value, I print q point and d, I get the correct value everything is ok.

Now, I make a change? What I do is, I take the address of b and cast it to A\* and take that value in p. So, p is a, A type pointer which is now pointing to a B type object and similarly that is what I have done for q. So, what will p pointer i print? p knows p is declared as a type. So, p knows that p pointer i is an integer but what you actually have that? You have taken a B object so what you actually have there is a floating point value. So, the formats of these two numbers are very different. So, what it prints, it does compile but it prints garbage. Similarly, it happens the other way because you are thinking that you are printing a double but actually there is an int.

So, when you do forced casting between unrelated types be very careful because it can lead you to really really severe errors. We will show what are the proper ways of doing such things.

(Refer Slide Time: 25:24)

The slide is titled "Type Casting Rules: Inheritance Hierarchy". It features a blue header with a logo on the left and a small video feed of a speaker on the right. The main content area has a white background with a blue border. At the top, a bullet point states: "Casting on a **hierarchy** is permitted in a limited sense". Below this, C++ code is shown: 

```
class A { };
class B : public A { };
A *pa = 0;
B *pb = 0;
void *pv = 0;
pa = pb; // UPCAST: okay
pb = pa; // DOWNCAST: error: '*' : cannot convert from 'A*' to 'B*' ✓
pv = pa; // okay, but lose the type for A* to void*
pv = pb; // okay, but lose the type for B* to void*
pa = pv; // error: '*' : cannot convert from 'void*' to 'A*'
pb = pv; // error: '*' : cannot convert from 'void*' to 'B*'
```

 To the right of the code is a hand-drawn diagram showing a vertical arrow pointing up from 'B' to 'A', and a diagonal arrow pointing down from 'A' to 'B', illustrating the relationship between the base class and the derived class.

The third set of rules relate to the classes on a hierarchy. If the classes are on a hierarchy then the casting is permitted in a limited sense. So, I have a class A which is the base class, I have a class B which is the derived class. I have pointers to A and B and I have a void pointer as well. So, this is called up-cast, why up cast, because I have A and B is A. So, A is above. So, I am taking from B to A. So, actually pb is pointing to a B object which has a base which is a and then whatever additional.

So, if I treat this as an A object then I do not lose any information because the base part is already there only restriction would be that I will deal with the base part. But if I do the reverse that is, I take A object pointer and think that it is a B object, then I will have serious problem because the B pointer pointed to B type will expect those additional data of the B class above the base class part which are not there. So, this will not be permitted this is called down cast. Now, I can assign any one of them, convert any one of them implicitly to void pointer which we have seen earlier as well but the reverse naturally is not allowed.

(Refer Slide Time: 27:13)

The slide displays the following C++ code:

```
class A { public: int dataA_; };
class B : public A { public: int dataB_; };

A a;
B b;

a.dataA_ = 2;
b.dataA_ = 3;
b.dataB_ = 5;

A *pa = &a;
B *pb = &b;

cout << pa->dataA_ << endl; // prints 2
cout << pb->dataA_ << " " << pb->dataB_ << endl; // prints 3 5

pa = &b;

cout << pa->dataA_ << endl; // prints 5
cout << pa->dataB_ << endl; // error: 'dataB_' is not a member of 'A'
```

So, we conclude that up casting is safe, so if I now take explicit classes A and B with two data members and create, so A has one member, B has an additional member and of course it will have the member of A inherited. So, in the A object I set the dataA\_, in the B object I set dataA\_ as well as dataB\_ and I create two respective pointers with the addresses, I print through the pointer to A, I print through the pointer through B, I get correct values.

If I do this, then on the right hand side, I have an address of a B object and I am taking that to be an A object. So, what I will be able to do is, I will be able to print the A part of it, that is this three. But if I try to print the B part, I will get a compilation error because pa is of type a class.

So, it does not know of a data member which is of type of the dataB\_ which is dataB\_. So, it is actually there because the address is of a B type object but I will not be able to access that in this way through the because the compiler will not know this type.

(Refer Slide Time: 28:42)

```
class A { public: int dataA; };  
class B : public A { public: int dataB; };  
  
A a;  
B b;  
  
a.dataA = 2;  
b.dataA = 3;  
b.dataB = 5;  
  
B *pb = (B*)a; // Forced downcast  
  
cout << pb->dataA << endl; // prints 3  
cout << pb->dataB << endl; // Compilation okay. Prints garbage for 'dataB' -- as 'dataB' is 'A' object
```

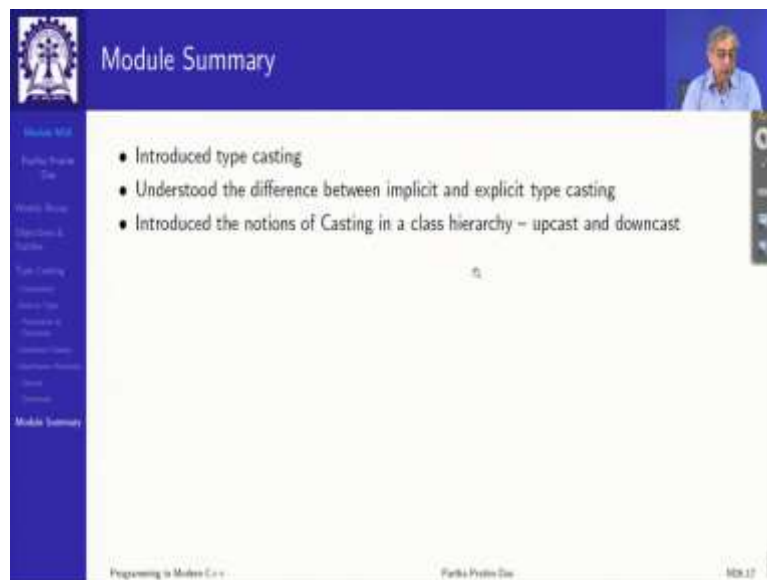
Now if I do down casting forcibly in the same example, the same initialization all that I am doing, I am doing a down casting here that is I am taking the address of an A object and putting it, as if it is a B object. So, I am leading to you know information which does not exist.

Now compiler cannot do anything in this case because you have told the compiler it is an explicit cast, explicit down cast, but with this, with pb, I can access dataA\_ and I think, I can access dataB\_ also because it has both of this. It is a, pb is a b type pointer. Now naturally, if I access pa, pb, I am sorry, if I access dataA\_ with pb then I will get this value 2 which is correct.

But when I do the other that actually does not exist, though the pointer thinks that it exist. So, it will be compilation ok, but it will print a garbage. So, you can see that why down casting is risky and why it needs to be severely restricted and done only in the proper context.



(Refer Slide Time: 29:58)



The image shows a presentation slide titled "Module Summary" with a blue header. In the top right corner, there is a small video feed of a man in a light blue shirt. The slide content is as follows:

- Introduced type casting
- Understood the difference between implicit and explicit type casting
- Introduced the notions of Casting in a class hierarchy – upcast and downcast

At the bottom of the slide, it says "Programming in Modern C++" and "Patrick Proft, DLR".

So, to conclude we have talked about, the introduce a basic notion of type casting with particular reference to implicit and explicit type casting and different type rules for built in types, numerical types, pointers, unrelated classes and introduce the notion of up-cast and downcast on a class hierarchy. Thank you very much for your attention and we will meet in the next module.