Typecasting pointers

```
class Dummy {
 double m_A{ 8 }, m_B{ 12 };
};

class Addition {
public:
  int Result() { return m_X + m_Y ; }
private:
  int m_X{1}, m_Y{2};
};

int main() {
  Dummy *pDummy = new Dummy();
  Addition * pAddition = (Addition*)pDummy;
  std::cout << pAddition->Result();

  std::cin.get();
  return 0;
}
```

- Two unrelated classes
- Build: 0 errors, 0 warnings (!)
- output: 1075838976
- Expected: there is no result method in the Dummy class and still it compiles!
- Unrestricted explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or some other unexpected results.
- BAD CODE

Typecasting pointers

- Using unrestricted explicit type-casting for pointers is a bad idea.
- There are better ways to typecast pointers or references.
- They are the templated pointer conversion functions:
  - `dynamic_cast`
  - `static_cast`
  - `reinterpret_cast`
  - `const_cast`
- Their format is to follow the new type enclosed between angle-brackets <> and immediately after, the expression to be converted between parentheses. Example:

```
dynamic_cast <new_type> (expression)
```

dynamic_cast

- Safely converts pointers and references to classes up, down, and sideways along the inheritance hierarchy.

- does RunTime Type Information checking > RTTI (!)
  - Do not use in time critical sections

- Can only be used with pointers and references to classes (or with void*). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.

dynamic_cast

- Resulting returned value:
  - When successful: a value of the new type.
  - When not successful:
    - pointer type: a nullptr is returned -> always check the value of resulting pointer!!!
    - reference type: exception of bad_cast is thrown

- Upcast: Converting from pointer-to-derived to pointer-to-base.

- Downcast: Convert from pointer-to-base to pointer-to-derived of polymorphic classes (those with virtual members) if -and only if- the pointed object is a valid complete object of the target type.

dynamic_cast: upcast example

```
class Base
{
public:
  virtual void Print() const { std::cout << m_A << '\n'; }
protected:
  int m_A{2};
};
class Derived : public Base
{
public:
  virtual void Print() const override{ std::cout << m_A << m_B << '\n'; }
  void DerivedPrint() const { std::cout << "DerivedPrint\n"; }
private:
  int m_B{ 4 };
};

int main()
{
  std::cout << "Upcast example:\n";
  Derived * pDerived { new Derived{} };
  pDerived->Print(); // prints 24
  Base* pBase { dynamic_cast<Base*>(pDerived) };
  if(pBase) pBase->Print(); // prints 24 (virtual base class function)

  // compile error 'DerivedPrint': is not a member of 'Base'
  if(pBase) pBase->DerivedPrint();
}
```

- Output:

```
Upcast example:
24
24
```

Derived::DerivedPrint is not declared as virtual in the Base class, thus not visible from the Base class pointer.

dynamic_cast: downcast example

```cpp
class Base {
public:
  virtual void Print() const { std::cout << m_A << '\n'; }
protected:
  int m_A{2};
};

class Derived : public Base {
public:
  virtual void Print() const override{ std::cout << m_A << m_B << '\n'; }
  void DerivedPrint() const { std::cout << "DerivedPrint\n"; }
private:
  int m_B{ 4 };
};

int main() {
  std::cout << "Downcast example:\n";
  Base * pBase { new Derived{} };
  pBase->Print(); // prints 24 (virtual base class function)
  Derived* pDerived { dynamic_cast<Derived*>(pBase) };
  if (pDerived) { //check for failed conversion: ok
    pDerived->Print();
    pDerived->DerivedPrint();
  }
  else { std::cout << "dynamic_cast failed\n"; }
}
```

- Output:

```
Downcast example:
24
24
DerivedPrint
```

Successful dynamic cast because the targeted object type is Derived and the requested pointer type conversion is also to the type Derived.

dynamic_cast: downcast example

```cpp
class Base {
public:
  virtual void Print() const { std::cout << m_A << '\n'; }
protected:
  int m_A{2};
};

class Derived : public Base {
public:
  virtual void Print() const override{ std::cout << m_A << m_B << '\n'; }
  void DerivedPrint() const { std::cout << "DerivedPrint\n"; }
private:
  int m_B{ 4 };
};

class OtherDerived : public Base {
public:
  virtual void Print() const override{ std::cout << m_A << m_B << '\n'; }
  void OtherDerivedPrint() const { std::cout << "OtherDerivedPrint\n"; }
private:
  int m_B{ 4 };
};

int main() {
  std::cout << "Downcast example:\n";
  Base * pBase { new Derived{} };
  OtherDerived* pOtherDerived { dynamic_cast<OtherDerived*>(pBase)};
  if (pOtherDerived) { //check for failed conversion: not ok
    pOtherDerived->Print();
    pOtherDerived->OtherDerivedPrint();
  }
  else { std::cout << "dynamic_cast failed\n"; }
}
```

- Output:

```
Downcast example:
dynamic_cast failed
```

Not a successful dynamic cast because the targeted object type is Derived and the requested pointer type conversion is to the type OtherDerived.

static_cast

- Same functionality as dynamic cast with this difference:
- On downcasts (from pointer-to-base to pointer-to-derived).
  - No checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destination type.
  - Therefore, it is up to the programmer to ensure that the conversion is safe.
  - On the other side, it does not incur the overhead of the type-safety checks of dynamic_cast -> faster.
- At compile time > NO RTTI
- Can lead to unexpected results or runtime errors.
- Can convert any type to void, evaluating and discarding the value.
  - interesting: is used to pass pointers through callback operations.
- Converts enum class values into integers or floating-point values.

static_cast: downcast example

```cpp
class Base {
public:
  virtual void Print() const { std::cout << m_A << '\n'; }
protected:
  int m_A{2};
};

class Derived : public Base {
public:
  virtual void Print() const { std::cout << m_A << m_B << '\n'; }
  void DerivedPrint() const { std::cout << "DerivedPrint\n"; }
private:
  int m_B{ 4 };
};

class OtherDerived : public Base {
public:
  virtual void Print() const override{ std::cout << m_A << m_B << '\n'; }
  void OtherDerivedPrint() const { std::cout << "OtherDerivedPrint\n"; }
private:
  int m_B{ 4 };
};

int main() {
  std::cout << "Downcast example:\n";
  Base * pBase { new Derived{} };
  std::cout << "Static Downcast example:\n";
  OtherDerived* pOtherDerived { static_cast<OtherDerived*>(pBase) };
  pOtherDerived->Print();
  pOtherDerived->OtherDerivedPrint();
}
```

- Output:

```
Static Downcast example:
24
OtherDerivedPrint
```

Static cast: no checking. In this example, we seem to be lucky; the call to a function that is not a member is successful.

UNDEFINED BEHAVIOUR

reinterpret_cast

- Converts any pointer type to any other pointer type, even of unrelated classes.
  - The operation result is a simple binary copy of the value from one pointer to the other.
  - All pointer conversions are allowed: neither the content pointed to nor the pointer type itself is checked.
- Low-level operations
- Assumes the programmer knows very well what he is doing.

const_cast

- cast away the constness of variables
- change non-const class members inside a const member function

- Bad…
- Bad…
- Bad…

typecasting: conclusion

- dynamic_cast: RTTI -> RUNTIME!!
  - for related classes
  - most secure type of pointer casting
  - ensures complete pointer objects when returned pointer is not nullptr
  - always check the resulting pointer to nullptr
- static_cast -> COMPILETIME
  - for related classes
  - less secure
  - returns valid pointer even if it points to a not complete object
  - can convert any type to void*
- reinterpret_cast -> COMPILETIME
  - not secure
  - can cast unrelated pointers (from any pointer to any other pointer)
  - can convert void* to any type

typeid > RTTI

- Is an operator that allows to check the type of an expression:
  - `typeid(expression)`
- It returns a reference to a constant object of type `type_info` that is defined in the standard header `<typeinfo>`.
- Returned values can be compared using the operators == and !=
- Returned value can be used to obtain a character sequence representing the data type or class by using its name() member(!).
- Happens at runtime (!) > bad for performance

typeid: example

```
// typeid
#include <iostream>
#include <typeinfo>
using namespace std;

int main() {
  int * a, b;
  a = 0; b = 0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

Output:

a and b are of different types:
a is: int *
b is: int

typeid

```cpp
// fill a container with base class pointers
std::vector<Base*> basePointers;
for (int i{}; i < 10; ++i)
{
  if(rand()%2) basePointers.push_back(new Derived());
  else basePointers.push_back(new OtherDerived());
}

for (Base* p : basePointers)
{
  if (typeid(*p) == typeid(Derived))
  {
    Derived* pDerived = static_cast<Derived*>(pBase);
    pDerived->DerivedPrint();
  }
  if (typeid(*p) == typeid(OtherDerived))
  {
    OtherDerived* pOtherDerived = static_cast<OtherDerived*>(pBase);
    pOtherDerived->OtherDerivedPrint();
 }
}
```

- typeid can be used to determine to what type of object a pointer refers to.
- static_cast is safe to use in this situation.
- Performance hit, avoid in a game loop
- Alternative: Define an enum class in the base class. Derived constructors pass enum to base class constructor.