Type Casting and Function Pointers

Implicit type conversion

- Implicit type conversions do not require any operator. They are automatically performed when a value is copied to a compatible type.

```
short a=2000;
int b;
b=a;
// value of "a" promoted from short to int –
// known as standard conversion
```

- Standard conversions affect fundamental data types (short to int, int to float, double to int...)
- Compiler often signals a warning if a standard conversion implies a lack of precision

Implicit type conversion

- Implicit type conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions.

```
class A {};
class B { public: B (A a) {} };
A a;
B b=a;
// an implicit conversion happened between objects of class A
//  and class B.  Because B has a constructor that takes an object
// of class A as parameter, implicit conversions from A to B is okay
```

Explicit type conversion

```
short a=2000;
int b;
b = (int) a;    // c++-like cast notation
b = int (a);    // functional notation
```

- The functionality of these explicit conversion operators is enough for most needs with fundamental data types.
- However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors (an example in the following slide).

Bad explicit type conversion

```
class CDummy {
    float i,j;
};
class CAddition {
            int x,y;
  public:
            CAddition (int a, int b) { x=a; y=b; }
            int result() { return x+y;}
};
int main () {
  CDummy d;
  CAddition * padd;
  padd = (CAddition*) &d; // terrible!
  cout << padd->result();
  }
```

Specific type casting operators

- dynamic_cast
- reinterpret_cast
- static_cast
- const_cast

dynamic_cast

- can be used only with pointers and references to objects
- ensures that the result of the type conversion is a valid complete object of the requested class
- dynamic_cast is always successful when we cast a class to one of its base classes

```
class CBase { };
class CDerived: public CBase { };
CBase b; CBase* pb;
CDerived d; CDerived* pd;
pb = dynamic_cast<CBase*>(&d);     // ok: derived-to-base
pd = dynamic_cast<CDerived*>(&b);  // wrong: base-to-derived
```

dynamic_cast

- A class with a virtual function is called “polymorphic”
- For polymorphic classes, dynamic_cast performs runtime checking to ensure that the expression yields a valid complete object of the requested class

```
class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };
int main () {
  try {
    CBase * pba = new Cderived;
    CBase * pbb = new CBase;
    CDerived * pd;
    pd = dynamic_cast<CDerived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast" << endl;
    pd = dynamic_cast<CDerived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast" << endl;
  } catch (exception& e) {cout << "Exception: " << e.what();}
}
```

RTTI

- dynamic_cast requires the Run-Time Type Information (RTTI) to keep track of dynamic types.
- Some compilers support this feature as an option which is disabled by default.
- This must be enabled for runtime type checking using dynamic_cast to work properly.
- Most modern compilers support it by default.

static_cast

- static_cast can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived.
- This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type.
- Con: it is up to the programmer to ensure that the conversion is safe.
- Pro: the overhead of the type-safety checks of dynamic_cast is avoided.

Caveat of static_cast

```
class CBase {};
class CDerived: public CBase {};
CBase* a = new CBase;
CDerived* b = static_cast<CDerived*>(a);
```

- Above would be valid, although “b” would point to an incomplete object of the class and would lead to runtime errors if dereferenced.

static_cast

- can also be used to perform any other non-pointer conversion that could also be performed implicitly

```
double d=3.14159265;
int i = static_cast<int>(d);
// yet another way to specify cast explicitly for
// fundamental types
```

reinterpret_cast

- Used for casts that are not safe:
  - Between integers and pointers
  - Between pointers and pointers
  - Between function-pointers and function-pointers
  - (see funptr.cpp posted along with today's lecture notes to understand function pointers)

reinterpret_cast is dangerous!

- The only guaranty that you get is that if you cast an object back to the original data-type (before the first cast) then the original value is also restored (of course only if the data-type was big enough to hold the value).
- However, a reinterpret_cast cannot be used to cast a const object to non-const object.

```
char *const MY = 0;
int *ptr_my = reinterpret_cast<int *>( MY);
// above is invalid
```

const_cast

- Manipulates the “constness” of an object, either to be set or to be removed
- Cannot convert the type

```
const char *my = "Hello";
int *a;
a  = const_cast<int *>(my); // error
```

- reinterpret_cast cannot cast the const away

```
a = reinterpret_cast<const char*>(my); // error
```

- Dirty, but the following would work

```
a = reinterpret_cast<int *>(const_cast<char *>(my) );
```

typeid

- Allows to check the type of an expression

```
int main()
  {
     int a;
     char *c;
     cout << "The type of a is: " << typeid(a).name() << endl;
     cout << "The type of c is: " << typeid(c).name() << endl;
}
```