C++ Typecasting

CSci 588: Data Structures, Algorithms and Software Design

http://www.cplusplus.com/doc/tutorial/typecasting/

Typecasting

Typecasting is a way of converting some value into another type.

You've already had to do some versions of this in previous labs.

Implicit Typecasting

Classes can also be converted implicitly if they implement a constructor which can perform the conversion:

```
class A {};
class B { public: B (A a) {} };

A a;
B b1 = a;
B b2(a);  // because the previous line means the same as this

B b3;
A a1 = b3; // this will not work as this conversion is not
           // two-way. A does not have a constructor for B
```

Explicit Typecasting

The typecast can also be explicitly specified (which will get rid of the warning messages for some cases):

```
double d = 1000;

int a;
a = (int)d; // c-style cast
a = int(d); // functional cast
```

Both style casts do the same thing. C++ allows both for backwards compatibility (and to make life more confusing).

Explicit Typecasting

The legacy explicit typecasting from c comes with a fair set of problems however:

```
class Point {
  public:
   int x, y;
   …
};

double *d = (double*)malloc(sizeof(double) * 100);

Point *my_point = (Point*)d;

cout << my_point.x << endl;
```

Explicit typecasting allows you to explicitly say that a variable is a certain type — even if you are wrong about it. This can lead to very unexpected results and difficult to track-down crashes.

Typecasting in C++

C++ provides 4 types of casting statements (hooray complexity) to help alleviate some of these issues:

```
dynamic_cast <new_type> (expression)

reinterpret_cast <new_type> (expression)

static_cast <new_type> (expression)

const_cast <new_type> (expression)
```

These are very similar (if not identical to) template functions which do the conversion and check for errors.

dynamic_cast

dynamic_cast is the safest (and therefore slowest) way to cast pointers and references to objects.

```
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 allows you to convert pointers to non-pointers, and derived (child) classes to their base (parent) class. In the case of non-polymorphic (ie., classes without virtual members) it will produce a compilation error when converting a base class to one of its derived classes.

dynamic_cast

```
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

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();
  }
  return 0;
}
```

With polymorphic classes, dynamic_cast does a check at runtime to see if the conversion is allowable.

If it is not allowable, dynamic cast returns null if the object being converted is not a complete representation of what it's being converted to (ie., all members are defined in the class its being converted to) — if not it will return null.

If it tries to convert the class to a different class (not a derived class) then it will throw a bad_cast exception.

dynamic_cast

dynamic_cast will also convert null values between any different pointer types, even between unrelated classes (as any pointer can be null).

It can also cast any pointers to void* (which means a pointer of unknown type).

dynamic_cast

Compatibility note: `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.

static_cast

static_cast is more efficient than dynamic_cast, but not as safe. It will convert between base and derived classes, but does not perform a runtime check to make sure that the class being converted is a full implementation of what it's being converted to.

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

The above is valid code, however if b is dereferenced it could lead to runtime errors as the class b is pointing to is not a full implementation of the class CDerived.

static_cast

static_cast can also be used to do any of the implicit/ explicit conversions (either between primitive types, or between classes where there is a conversion constructor):

```
double d = 3.14159265;
int i = static_cast<int>(d);

class A {};
class B { public: B (A a) {} };

A a;
B b1 = static_cast<B>(a);
```

reinterpret_cast

reinterpret_cast converts any pointer type to any other pointer type (and thus is quite efficient).

All it does is a simple binary copy of the pointer value from one pointer to another. (Why you'd want to do this I'm not sure).

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

This code is valid, however if you used b, you'd end up with a runtime error. Generally, reinterpret_cast is used for system-specific low level conversions.

reinterpret_cast

reinterpret_cast can also cast a pointer to or from an integer:

```
class A {};
A * a = new A;
int pointer_value = reinterpret_cast<int>(a);
```

This will work as long as the system pointer representation is large enough to fit into an int (otherwise it return a compile time error?) — basically it will ensure that given a 32 bit or 64 bit system you're converting your pointer into something large enough to represent it.

const_cast

const_cast is dangerous. it allows you to add or remove the ‘const’ or ‘volatile’ modifiers from a variable (which would lead to code having unexpected results):

```
// const_cast
#include <iostream>
using namespace std;

void print (char * str) {
  cout << str << endl;
}

int main () {
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  return 0;
}
```

I would avoid using this at all costs. I’m surprised it even exists.

type_id

type_id allows you to check the type of an expression. It's a primitive version of Java's reflection. It also allows you to determine if two variables are of the same type or not:

```
// 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;
}
```

type_id

type_id also uses RTTI (so make sure your compiler has it turned on — almost all do nowadays). When applied to a polymorphic class (note: not a pointer) it gives you, dynamically calculated at runtime, the most complete derived class the object can be:

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class CBase { virtual void f(){} };
class CDerived : public CBase {};

int main () {
  try {
    CBase* a = new CBase;
    CBase* b = new CDerived;
    cout << "a is: " << typeid(a).name() << '\n'; //prints CBase*
    cout << "b is: " << typeid(b).name() << '\n'; //prints CBase*
    cout << "*a is: " << typeid(*a).name() << '\n'; //prints CBase
    cout << "*b is: " << typeid(*b).name() << '\n'; //prints CDerived
  } catch (exception& e) { cout << "Exception: " << e.what() << endl; }
  return 0;
}
```

type_id

The results of type_id(variable).name is compiler specific, so using it is generally not portable (as on different systems the name could be different for the same class/ type).

The only requirement of type_id::name is that it be human readable, but not all compilers even follow that. It could just return any string.

If a null value is passed to type_id, it will throw a bad_typeid exception.