Runtime Type Information (RTTI)

Chapter 16

Objectives

- **Describe the C++ runtime type information (RTTI) mechanism.**
- **Use RTTI for special purposes in programs where the standard virtual function mechanism is not adequate.**
- **Use *dynamic_cast* to achieve type safety in working with pointer conversions.**
- **Describe the new C++ cast notation and discuss its benefits.**
- **Gain experience through code walk-throughs and lab exercises.**
  - The example programs are in the chapter directory.
  - Labs located in Labs/Lab16

Runtime Type and Polymorphism

- **The normal virtual function mechanism of C++ supports "intelligent" objects that can respond polymorphically to requests based on their object type.**
  - The code that invokes the virtual function just calls the function through a pointer (or reference) and does *not* know the exact data type.
  - Not knowing the data type is usually an advantage for code that invokes the virtual function, because this code is general and does not have to change as new data types are added.
- **Sometimes code *does* need to know the data type.**
- **The ANSI C++ standard provides a *runtime type information* mechanism that supports inquiry at runtime about the data type of an object.**

Runtime Type Example

- **Examine the program Demopoly.**
- **The program illustrating polymorphism is modified to place an asterisk next to each sales employee using RTTI.**

type_info Class

- ***type_info* is a built-in C++ class that describes type information about an object that is generated by the compiler.**
  - Objects of this class effectively store a pointer to a name for the type (called a "type id").

- **Important members of *type_info* include:**
  - Operators **==** and **!=** for comparing type id's.
  - Function **name** that returns a **const char *** pointer to a name for the type.

typeid Operator

- **The *typeid* operator allows the type of an object to be determined at run time.**
  - The result of a **typeid** expression is a **const type_info&**.
  - The value is a reference to a **type_info** object that represents the type of the expression.
  - const type_info& id = typeid(*pAccount);
- **The type id value returned then can be compared to a given type, using the comparison operators of the *type_info* class.**
  - if (id != typeid(SavingsAccount)) ...
- **You must use a special include file.**
  - #include <typeinfo>

Safe Pointer Conversions

- **When base class pointers are used to refer to objects and later it is desired to access members of a derived class, the pointer must be cast.**

```
Employee* pEmp = GetEmployee(empId);
HourlyEmployee *pHourly = (HourlyEmployee*) pEmp;
auto hoursWorked = ((HourlyEmployee *)pEmp)->GetHoursWorked();
```

- **Such code is *error-prone at runtime* because the particular object may *not* be of the expected type.**

Safe Pointer Conversions (continued)

- **We could use the typeid mechanism here to do a type check.**
- **RTTI uses an exact type, so if using a type id for this purpose we must check for specific types.**

```
Employee *pEmp = GetEmployee(empId);
const type_info& id = typeid(*pEmp);
if (id == typeid(HourlyEmployee)){
  auto hoursWorked = ((HourlyEmployee *)pEmp)->GetHoursWorked();

  . . .

}
```

Dynamic Cast

- **A more convenient solution is to use the C++ *dynamic_cast* template function.**
  - This "attempt to cast" function returns the desired pointer if successful and a **null pointer** if unsuccessful.
  - To use **dynamic_cast** you should have enabled RTTI in the compiler.

```
Employee *pEmp = GetEmployee(empId);
HourlyEmployee *pHourly = dynamic_cast<HourlyEmployee *>(pEmp);

    if (pHourly != nullptr){
        auto hoursWorked = pHourly->GetHoursWorked();
        . . .
    }
```

C++ Style Casts

- **The traditional C style cast is now discouraged.**

```
long x;
short n = (short) x;
```

- **Casts are a frequent source of program errors, and old style syntax *(type)* does not permit a search of all casts in a program.**
  - Each data type, included class types, has a unique cast operator in the old style.
- **The C++ cast syntax uses four keywords.**
  - static_cast
  - dynamic_cast
  - const_cast
  - reinterpret_cast

Static Cast

- **The *static_cast* function does "well behaved" casts, like widening and narrowing of data types and static navigation of class hierarchies**

```
long a;
short b;
b = static_cast<short>(a);
```

  - There is no runtime type checking in downcasting in a class hierarchy as in **dynamic_cast**.
  - You cannot use **static_cast** to convert a pointer type to a non-pointer type (use **reinterpret_cast**).
  - You cannot cast away "constness" (use **const_cast**).

Reinterpret Cast

- **The *reinterpret_cast* allows converting to another type with a completely different meaning.**
  - You can convert a pointer type to a non-pointer type.
- **Typical usage is to convert to a numerical type to allow "bit twiddling."**

```
long a;
long* pa;
a = reinterpret_cast<long>(pa);
```

Const Cast

- Use *const_cast* to convert a constant type to a non-constant type.

```
const int i = 0;
// int *pi = &i;                    // illegal
*pi = 5;                            // value changed!

int *qi = const_cast<int*>(&i);
*qi = 10;                           // value changed again
```

- See the program CastDemo for examples of various cast operations.

Summary

- The new C++ standard supports a runtime type information (RTTI) mechanism.
- *type_info* is a built-in C++ class that describes type information about an object that is generated by the compiler.
- The *typeid* operator allows the type of an object to be determined at run time.
- You can use RTTI for special purposes in programs where the standard virtual function mechanism is not adequate.
- dynamic_cast can be used to achieve type safety in working with pointer conversions.
- The new C++ cast notation uses keywords such as *dynamic_cast* and *static_cast*, making it easy to search for all occurrences of casts in your programs.