Chapter 2: Variables (Objects) and Basic Types

2.3.1 Reference (l-value reference)

Examples

```
int a = 10;
int& r = a; (int &r = a;)
//(don't confuse with address of operator; see 2.3.2 below)
r = 20; // assign 20 to the object r refers, i.e., to a
Sales_item w;
Sales_item& x = w;
```

Quick Check: What does the following code print?

RefEx.cpp

```
#include <iostream>

using namespace std;

int main()
{
    int i;
    int& ri = i;
    i = 5;
    ri = 10;
    cout << "i = " << i << endl;
    cout << "ri = " << ri << endl;
    return 0;
}
```

A:

Remark: The primary usage of reference is parameter-passing for functions. We will have more to say on the topic later.

(see ppt)

2.3.2 Pointer

Some symbols, such as `&` and `*`, are used as both an operator in an expression and as part of a declaration. The context in which a symbol is used determines what the symbol means:

```
int i = 42;
int &r = i; // & follows a type and is part of a declaration; r is a reference
int *p; // * follows a type and is part of a declaration; p is a pointer
```

```
p = &i; // & is used in an expression as the address-of operator
*p = i; // * is used in an expression as the dereference operator
int &r2 = *p; // & is part of the declaration; * is the dereference operator
```

Quick Check: What does the following code print?

PointerEx.cpp

```
#include <iostream>

using namespace std;

int main()
{
    int i = 4;
    int* pi = &i;
    *pi = *pi * *pi;
    cout << "i = " << i << endl;
    cout << "*pi = " << *pi << endl;
    return 0;
}
```

A:

Brief Summary: Compound Type

In C++, a type that is defined in terms of another type is called compound type (複合型別). We have introduced two compound types so far: (1) reference: to define an alias for another object and (2) pointer: to define an object that can hold the address of an object.

(see ppt)

22 `auto` Type Specifier

Appreciating `auto`

One of the goals of C++11 is to make C++ easier to use, letting the programmer concentrate more on design and less on details. The automatic type deduction feature `auto` reflects a philosophical shift in the role of the compiler.

(C++98)

```
std::map<int, std::string> m;
std::map<int, std::string>::iterator i1 = m.begin();
```

(C++11)

```
std::map<int, std::string> m;
auto i1 = m.begin(); // i1: std::map<int, std::string>::iterator
```

In C++98, the compiler uses its knowledge to tell you when you are wrong. In C++11, at least with this feature, it uses its knowledge to help you get the right declaration.

Reference, `const` and `auto`

The type that the compiler infers for `auto` **is not always exactly the same as the initializer's type.** Instead, the compiler adjusts the type to conform to normal initialization rules.

Reference: when we use a **reference**, we are really using the object to which the reference refers. In particular, when we use a reference as an initializer, the initializer is the corresponding object. The compiler uses that object's type for `auto`'s type deduction:

```
int i = 0, &r = i;
auto a = r; // a is an int (r is an alias for i, which has type int)
```

If we really want the deduced type to have a reference, we must say so explicitly:

```
int i = 0, &r = i;
auto& a = r; // a is now an int&
```

Remark: Deduce a type to have a reference is often used with the new standard range-for loop:

```
string s = "Hello World";
for (auto& c: s) c = tolower(c);
cout << s << endl;
```

We will have more to say on the topic later.

Top-level `const`: similarly, `auto` ordinarily ignores top-level `const`. If we really want the deduced type to have a top-level `const`, we must say so explicitly:

```
const int ci = 40;
const auto fi = ci;
```

Quick Check: Determine the types of `j`, `k`, `p`, `j2`, `k2` deduced in each of the following definitions.

```
const int i = 42;

auto j = i;
const auto &k = i;
auto *p = &i;
const auto j2 = i, &k2 = i;
```

A:

Brief Summary: In C++11, `auto` is used to deduce the type of a variable from its initializer. This is a very useful and convenient feature, especially when that type is either hard to know exactly or hard to write.

(see ppt)

2.3 `decltype` Type Specifier

```
int i;
const int ci = 0, &cj = ci;
decltype(ci) x = 0; // x has type const int
decltype (i) a; // a is an uninitialized int
decltype(cj) y = x; // y has type const int& and is bound to x
```

```
double f() {return 3.01;}
decltype(f()) sum = x;
// sum has whatever type f returns, double in this case
```

Quick Check: Assignment is an example of an expression that yields a reference type. The type is a reference to the type of the left-hand operand. That is, if `i` is an `int`, then the type of the expression `i = x` is `int&`. Using this knowledge, determine the type of `d` deduced from `decltype` statement

```
int a = 3, b = 4;
decltype(a = b) d = a;
```

A:

DeclEx.cpp

Q: What are the outputs?

```
#include <iostream>

using namespace std;

int main()
{
    int a = 3, b = 4;
    decltype(a) c = a;
    decltype(a = b) d = a;
    c++;
    d += 2;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    cout << "d = " << d << endl;
    return 0;
}
```

A:

Remark: the `decltype` is very useful for template programming as we will realize later in the class.

(see ppt)

2.6 Define Our Own Data Structures (or Define Our Own Types)

C++ allows the programmer to define a new **type**. One way to do this is through the `struct` which facilitates data abstraction. The other way to do this is through the `class` which we will cover later.

(From A to A+: Language RULE) If the class is defined with the `struct` keyword, then members are `public` if no further access label is imposed.

Data abstraction (資料抽象化) is a powerful mechanism whereby a set of related objects (often of different types) can be considered as a single object/type. For example, we can define a data abstraction `Student` that contains `name (string)`, `id (int)` and `age(int)`.

To do so, we use the keyword `struct` to define the `Student` data type:

```
struct Student
{
    std::string name;
    int id;
    int age;
};
```

The definition begins with a keyword `struct`, followed the name of your choice. Then one or more **data members** are declared within curly braces. Finally a semicolon terminates the `struct` definition.

(Reflection) Data abstraction allows us to handle data in a meaningful manner (or 人比較容易瞭解的方式). For example, we now can **think of** (我思故我在) Student as a new type that can represent「Student」in the real world. We can then pass the object of `Student` in/out of functions as we always do. We can also put the objects of `Student` in an array such as `vector`.

In-class member initialization: when we create objects, the in-class initializers will be used to initialize the data members. Members without an initializer are default initialized. Thus under the new standard, we can do:

```
struct Student
{
    std::string name;
    int id = 0;
    int age = 0;
};
```

Q: what does it mean when we define a Student object now?

```
Student john;
```

A:

(see ppt)

2.6.3 Writing Our Own Header Files

In order to ensure that the class definition is the same in each file, classes are usually defined in header files. Typically, classes are stored in headers whose name derives from the name of

the class. For example, the `string` library type is defined in the `string` header. Similarly, as we've already seen, we will define our `Sales_item` class in a header file named `Sales_item.h`.

The most common technique for making it safe to include a header multiple times relies on the preprocessor. The preprocessor—which C++ inherits from C—is a program that runs before the compiler and changes the source text of our programs. Our programs already rely on one preprocessor facility, `#include`. When the preprocessor sees a `#include`, it replaces the `#include` with the contents of the specified header.

C++ programs also use the preprocessor to define header guards. Header guards rely on preprocessor variables. **Preprocessor variables have one of two possible states: defined or not defined.** The `#define` directive takes a name and defines that name as a preprocessor variable. There are two other directives that **test** whether a given preprocessor variable has or has not been defined: `#ifdef` is true if the variable has been defined, and `#ifndef` is true if the variable has not been defined. If the test is true, then everything following the `#ifdef` or `#ifndef` is processed up to the matching `#endif`. We can use these facilities to guard against multiple inclusion as follows:

In `Sales_data.h`

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

Q: what happens when `Sales_data.h` is included at the first time?

In `main.cpp`

```
#include "Sales_data.h" // first time
...
#include "a.h" // second time
```

In `a.h`

```
#ifndef A_H
#define A_H
#include "Sales_data.h"
```

```
...

#endif
```

A:

Q: what happens when `Sales_data.h` is included at the second time?

A:

In-class Exercise: write a header file `Coord.h` that defines a type `Coord`with three doubles x, y, z with a proper header guard. Write a client code that uses the header file, read two `Coord`objects and output their sum. A sample run looks like:

```
Read the first Coord object with three doubles: 1.1 2.2 3.3
Read the second Coord object with three doubles: 0.2 0.4 0.6
The sum of these two Coord objects are: 1.3 2.6 3.9
```

A: