Chapter 2: Variables (Objects) and Basic Types

Primitive Built-in Types[1]:

- Integral Types
  - Integers: `short`, `int`, `long`, `long long` --- `signed`, `unsigned`
  - Characters: `char` --- `signed`, `unsigned`
  - Extended Characters: `wchar_t`, `char16_t`, `char32_t`
  - Boolean values: `bool`
  - The `unsigned int` can be abbreviated as `unsigned`.
- Floating-Point Types
  - `float`, `double`, `long double`

Examples

```
int a = 10; // 'int a' is the declaration, and initilized with 10
short i = 1;
double d = 0.5;
long double ld;
```

You can use the `sizeof` operator[2] to determine size of the variable type on your machine:

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

int main()
{
    cout << sizeof(char)<<"\n";
    cout << sizeof(int)<<"\n";
    cout << sizeof(float)<<"\n";
    cout << sizeof(double)<<"\n";
    return 0;
}
```

Output:

```
1
4
4
8
```

[1] If you are interested, the size of these variables can be found here: URL: http://www.cplusplus.com/doc/tutorial/variables/

[2] More details on `sizeof()` URL: https://www.geeksforgeeks.org/sizeof-operator-c/

Variables (Objects): A variable provides us with named storage that our programs can manipulate. C++ programmers tend to refer to variables as "variables" or as "objects" interchangeably. Each variable in C++ has a type. The type determines

- the size and layout of the variable's memory
- the range of values that can be stored within that memory
- the set of operations that can be applied to the variable.

A *compound type* (複合型別) is a type defined in terms of another type. We will cover two compound types: *reference* and *pointer*.

2.3.1 Reference (l-value reference[3])

A reference is an alternative name for an object (e.g., 孫文 and 孫中山). An object declared as a reference is merely a second name (alias) assigned to an existing object. No new object is created.

A reference is defined by preceding a variable name by the ampersand (&) symbol.

Examples

```
int a = 10;
int& r = a; // or (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., assign 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()
{
```

[3] For reference types, you can find info here: URL: https://www.learncpp.com/cpp-tutorial/15-2-rvalue-references/

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

A:

```
i = 10
ri = 10
```

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

2.3.2 Pointer

A pointer is a compound type that "points to" another type.

Conceptually, pointers are simple: a pointer holds the memory address of another object[4].

We use `*` operator symbol in a variable declaration to indicate that an identifier is a pointer.

```
string *pstring;
```

When attempting to understand pointer declarations, read them from **right to left**: `pstring` is a pointer that can point to string objects.

To retrieve the address of an existing object, use the *address-of operator* (`&`).

```
string  s("hello world");
string *sp = &s;   //sp holds the address of s
//initialize sp to point to the string named s;
```

We use the *dereference operator* (`*`) to access the object/value to which the pointer points.

```
string  s("hello world");
string *sp = &s;   //sp holds the address of s
cout << *sp;
```

[4] More details on pointers and memory can be found: URL: http://www.cplusplus.com/doc/tutorial/pointers/

Pointer with reference:

```
int ival = 1024;
int *pi = &ival; // pi points to an int
int **ppi = &pi; // ppi points to a pointer to int
cout << "The value of ival\n"
     << "direct value: " << ival << "\n"
     << "indirect value: " << *pi << "\n"
     << "doubly indirect value: " << **ppi
     << endl;
```

[Attention] 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 program print?

```
#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:

```
i = 16
*pi = 16
```

Brief Summary: Compound Type

In C++, a type that is defined in terms of another type is called the compound type. 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.

`const` Qualifier

The `const` qualifier provides a way to transform an object into a constant.

For example, if we want to define a constant such as `PI`.

When using constants in programming languages, we must initialize it when it is defined.

```
const double PI = 3.14159265358979323846264338327795;
```

There are times when the data type is obvious to the compiler given the context. The `auto` and `decltype` provide automatic type inference[5]. We introduce them in the following.

2.5.2 `auto` Type Specifier

We can let the compiler figure out the type for us by using the `auto` type specifier.

Unlike typical type specifiers, such as `double`, that name a specific type, `auto` tells the compiler to deduce (推斷) the type from the initializer (right hand side value).

```
auto i = 0;
vector<int> v;
vector<int>::iterator p1 = v.begin();
auto p2 = v.begin();
```

Appreciating `auto`

The automatic type deduction feature `auto` reflects a philosophical shift in the role of the compiler. This is included in C++11 and later.

When compiling the program, we need to place -std=c++11 to avoid warnings/errors. For example, the following are used when compiling the program, where the code is in testAuto.cpp:

```
g++ -std=c++11 testAuto.cpp
```

[5] Although this is a more advanced topic, more details on automatic type inference can be found URL: https://www.geeksforgeeks.org/type-inference-in-c-auto-and-decltype/

(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 is a std::map<int,
                        // std::string>::iterator
```

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&
```

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:

```
j // int
k // const int&
```

```
p // int*
j2 // const int
k2 // const int&
```

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/type.

2.5.3 `decltype` Type Specifier

`decltype` tells the compiler to deduce type from an expression. The compiler analyzes the expression to determine its type but does NOT evaluate the expression.

```
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
```

[Attention] 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:

```
d // int&
```

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:

```
a = 5
b = 4
c = 4
d = 5
```

Remark: the `decltype` is very useful for template programming as we will realize later in the class. [Attention] The operand of `decltype` does NOT get evaluated!

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

C++ allows the definition of a new **type**. One way to do this is through the `struct`. 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 or grouped 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** `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 or `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:

It means `john.name` is an empty string, `john.id` and `john.age` are initialized to zero.

Here is an example on how to use the `struct`:

```
#include <iostream>
#include <string>

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

int main() {
    Student s;
    s.name = "OOP";
    s.id = 6;
    s.age = 21;

    std::cout << "Student info: \n";
```

```
    std::cout << "\tname: " << s.name << "\n";
    std::cout << "\t  id: " << s.id << "\n";
    std::cout << "\t age: " << s.age << "\n";

    return 0;
}
```

Output:

```
Student info:
    name: OOP
    id: 6
    age: 21
```

2.6.3 Writing Our Own Header Files

Remember how to write headers in lecture 1?

In `Student.h`

```
#ifndef STUDENT_H
#define STUDENT_H
#include <string>
struct Student
{
    std::string name;
    int id = 0;
    int age = 0;
};
#endif
```

And in `main.cpp`

```
#include <iostream>
#include "Student.h"

int main() {
    Student s;
    s.name = "OOP";
    s.id = 6;
    s.age = 21;

    std::cout << "Student info: \n";
    std::cout << "\tname: " << s.name << "\n";
    std::cout << "\t  id: " << s.id << "\n";
    std::cout << "\t age: " << s.age << "\n";

    return 0;
}
```

Here `STUDENT_H` is the preprocessor variable.

Now if in the main we have include Student.h 2 times. Once directly in the `main.cpp`, and the other in `a.h`:

In `main.cpp`

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

In `a.h`

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

#endif
```

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

A: The first time `Student.h` is included, the `#ifndef` test will succeed. The preprocessor will process the lines following `#ifndef` up to the `#endif`. As a result, the preprocessor variable `STUDENT_H` will be defined and the contents of `Student.h` will be copied into our program.

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

A: If we include `Student.h` later on in the same file, the `#ifndef` directive will be false. The lines between it and the `#endif` directive will be ignored.