Variable

COMP3220 – Principle of Programming Languages

Zhitao Gong

2016 Spring

Outline

Variable Name

A name is a string of characters to identify an entity. The following design choices vary in different languages.

- What characters may be used in names?
- Case sensitive or not?
- Special words in the language is reserved words or keywords?

Valid Characters

- Most PL allow only ASCII characters, specifically `[a-zA-Z0-9_]`.
- Some allow certain Unicode characters, e.g., `C#`. `Java`.

```
class 数据库连接对象
{
    连接(string 服务器名 , string 用户名, string 密码, string 数据库名)
}
```

Figure: `C#` class with Chinese Characters LOL

- Character set isn't the problem. The non-descriptive, one-letter identifiers are.

C++ Convention

```
int main() {
  int sum;                          // good
  int v1, vv;                       // OK, but meaningless
  int ____v2, v3_____, _;           // OK, but NEVER DO THIS
  int 2v;                           // WRONG
}
```

never use "underhanded names," ones that begin with an underscore or that contain a double underscore (p2 , C++ Coding Standards, Herb Sutter and Andrei Alexandrescu)

- Names beginning with an underscore or a double underscore are RESERVED for the C++ implementers.
- Names with an underscore are reserved for the library to work.

C++ Naming Style

CamelCase
: In Compound words or phrases, each word or abbreviation begins with a capital letter, e.g., `UpperCamelCase` or `lowerCamelCase`.

snake_case
: Words are separated with one underscore character (_) and no spaces, with each element's initial letter usually lowercased.

Special Words

Special words in programming languages are used to make programs more readable by naming actions to be performed.

Keyword
: A word of a programming language that is special only in certain contexts, a part of the syntax, e.g., FORTRAN.

```
Integer Apple
Integer = 4
Integer Real
Real Integer
```

Reserved Word
: A special word of a programming language that *cannot* be used as a name. It is always a better choice unless too many words are reserved, e.g., `COBOL`.

Special Words Cont'd

In practice *most* keywords are reserved words and vice versa.

- Keyword may not be a reserved word, e.g. a keyword only has meaning in a special context, and may be used as an identifier. E.g., in Fortran, we may have

```
IF (if) THEN
   then
ELSE
   else
END IF
```

- Reserved word may not be keyword, e.g. reserved for future use. E.g., `goto` in `Java`.

Outline

Address of Variable

Address of a variable: The machine memory address with which it is associated.

- The same variable name may be associated with different addresses at different times in the program
- Multiple variables name may refer to the same memory location.

The address of a variable is sometimes called its L-value, since it appears on the left-hand side of an assignment.

One Variable Bound to Multiple Addresses

```
void foo() {
  int a = 3;
}

int main() {
  foo();
  foo();
}
```

Variable a is created on the *runtime stack*. Different calls of `foo()` may result in the local variable a bound to different addresses.

Multiple Variables Bound to Same Address

These variables are *aliases*.

- Aliasing is a hindrance to readability.
- It is error-prone.
- However, alias is sometimes preferred.

```
void foo(int& v) {
  v += 2;
}
int main() {
  int a = 3;
  foo(a);  // value of a after this call?
  int& b = a;
  b += 3;  // value of a after this statement?
}
```

C++ Alias Problem

The following code causes problems because of aliases.

```
int main() {
  int* a = new int{3};
  int* b = a;

  *b = 100;  // value of a after this statement?
  delete a;  // what about b?
  delete b;  // ERROR
}
```

We will revisit the *dangling pointer* problem.

C++ Alias Advantage

If we have a big user-defined types, e.g., `class C`. Pass it by value to a function call may be too space and time consuming.

```
class C {
  // a big class
};
void foo(const C& c) {
  // awesome function
}
int main() {
  C c;
  foo(c);
}
```

In this case, an alias is preferred.

Value

Value of a variable The contents of the memory cell or cells associated with the variable. A variable's value is sometimes called its R-value.

```
int main() {
  int a = 3;
  int b = a + 3;
  b + 4 = 4  // ERROR!
}
```

Outline

Concept of Binding

Binding An association between an attribute, e.g., name, type, value, address and etc., and an entity, e.g., an identifier or a symbol.

Some binding examples

- The meaning of reserved words `if`.
- The operation associated with a symbol `*`.
- The entity (variable, keyword, etc.) represented by an identifier.
- The range of data type, e.g., `int`.
- etc.

Binding Time

The time when a binding occurs is called *binding time*. Common binding times include (in chronological order):

- Language definition, e.g, general meaning of +
- Language implementation, e.g., internal representation of 23
- *Program translation (compile time)*, e.g., data type
- Link edit, e.g., a function call to a library
- Load, e.g., variable storage binding
- *Program execution (run time)*, e.g., value.

Name Binding

In most PL's, programmers may bind a name, *identifiers*, to program entities, e.g., variables, constants, functions, data types, and etc.

Name Binding Find the corresponding binding occurrence (definition/declaration) for an applied occurrence (usage) of an identifier.

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

int main() {
  int b = 3;
  {
    int b = 4;
    cout << b << endl;
  }
  cout << b << endl;
}
```

Static Type Binding

In both cases, the type of a variable cannot change.

Explicit Declaration: Lists variable names and specifies that they are a particular type.

```
int main() {
  int a = 3;
}
```

Implicit Declaration: Associating variables with types through default conventions. E.g., in `FORTRAN`, variables prefixed with `I`, `J`, `K`, `L`, `M`, `N` or their lowercase versions are *implicitly* `Integer`, otherwise `Real`.

```
int main() {
  auto a = 3.3;  // type deduction in C++11
}
```

C++ Type Deduction

```
#include <iostream>
#include <cmath>
#include <typeinfo>

// same as: double (*get_fun())(double)
auto get_fun() -> double (*)(double)
{
  return std::sin;
}
int main() {
  auto fun = get_fun();
  std::cout << "type of my_fun: " << typeid(fun).name() << '\n';
  std::cout << "fun: " << fun(3) << '\n';
}
```

You will see the type "PFddE" which is GCC mangled name.
`c++filt -t PFddE` returns `double (*)(double)`.

Dynamic Type Binding

Variable type is determined by "assignment". Usually found in script languages, `Javascript`, `Python`, `Lua`, etc.

```
var a = 3;
a = [1, 2];
a = "A String";
a = {"hello": 1, "world": 3};
```

Disadvantages include

- Code is less reliable.
- Dynamic binding cost is high.
- Type checking must be done at run time.
- Space for Run-time type descriptor.

Outline

Scope

Scope of a Variable: The range of statements in which the variable is visible. i.e., it can be referenced in that statement.

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

int main() {
  int a = 3;
  for (int i = 0; i < 10; ++i) {
    a += i;
  }
  cout << a << endl;
  cout << i << endl;  // ERROR
}
```

Reference Environment

Referencing Environment (RE) The collection of all variables that are visible in the statement.

```
int g = 3;

int main() {
  // RE: global g

  int a{3}, b{4};
  for (int i = 0; i < 10; ++i) {
    // RE: global g and local a, b, i
  }

  // RE: global g and local a, b

  // Why global g is accessible?
}
```

Local Scope

Visible within function or statement block from point of declaration until the end of the block.

```
void foo(int n) {
  int a = 3;
}

int main() {
  int a = 3;
  for (int i = 0; i < 3; ++i) {
    int b;
  }
}
```

Class Scope

Visible in class members. a_ is declared in the private section of class C, and is accessible within the definition of the class.

```
class C {
 public:
  C() : a_(4) {  }

  int get_a() const { return a_; }
  void set_a(int a) { a_ = a; }
 private:
  int a_;
};  // REMEMBER Semicolon!!!

int main() {
  C c1;
  C c2;
}
```

Namespace Scope

Visible within same namespace block.

```
namespace foo {
int a;
}

void f1() {
  a = 3;  // ERROR
}

namespace foo {
void f2() {
  a = 4;  // OK
}}

int main() {
  a = 3;        // ERROR
  foo::a = 3; // OK
}
```

File Scope

Visible within current file.

Foo

```cpp
// foo.cpp
#include "foo.h"
int pub_var;
static int pri_var;

void foo()
{
  pub_var = 1;
  pri_var = 2;
}
```

```cpp
// foo.h
void foo();
```

Main

```cpp
#include <iostream>
#include "foo.h"
using namespace std;

int main() {
  extern int pub_var;
  extern int pri_var;
  foo();
  cout << pub_var << endl   // OK
       << pri_var << endl; // ERROR
}
```

Compile with `g++ main.cpp foo.cpp`.

Global Scope

Visible everywhere unless *shadowed*.

```
int g = 3;

int main() {
  g = 3;
  int g = 4;
  ::g = 33;
}
```

Different from file scope: `static` prefix.

Scope Rule

Scope rules of a PL determine how a particular occurrence of a name is associated with a variable. Implicitly, variable are visible inside there declaration block, i.e., *local scope*.

```
int a = 3;

int main() {
  a = 4;
  int a = 100;
}
```

In particular, scope rules determine how references to variables declared *outside* the currently executing subprogram or block are associated with their declarations and thus their attributes.

Static/Lexical Scoping

Bind a name to a non-local variable at *compile time*. Find the smallest block syntactically enclosing the reference and containing a declaration of the variable.

```cpp
#include <iostream>
using namespace std;
int b = 0;
int foo() {
  int a = b + 1;  // b is NOT declared inside foo
  return a;
}
int bar() {
  int b = 1;
  return foo();
}
int main() {
  cout << foo() << endl;
  cout << bar() << endl;
}
```

Nested Static Scoping

Two categories of static scoping: subprograms may be nested (nested static scoping) and those may not be nested.

```
function fun1() {
  var v1 = 3;                   // v1 in fun1

  function fun2() {             // fun2 defined in fun1
    var v1 = 3;                 // v1 in fun2
    fun3();
  }

  function fun3() {             // fun3 defined in fun1
    console.log(v1);            // which v1 to use?
  }

  fun2();
}
```

Dynamic Scoping

Bind a name to a non-local variable at *run time*. Find the most recent, currently active run-time stack frame containing a declaration of the variable.

```
#include <iostream>
using namespace std;
int b = 0;
int foo() {
  int a = b + 1;  // b is NOT declared inside foo
  return a;
}
int bar() {
  int b = 1;
  return foo();
}
int main() {
  cout << foo() << endl;
  cout << bar() << endl;
}
```

Nested Dynamic Scoping

Suppose the following program uses dynamic scoping.

```
function fun1() {
  var v1 = 1;                     // v1 in fun1

  function fun2() {               // fun2 defined in fun1
    var v1 = -1;                  // v1 in fun2
    fun3();
  }

  function fun3() {               // fun3 defined in fun1
    console.log(v1);              // which v1 to use?
  }

  fun2();
}
```

Problem – Static Scoping

Code refining may break the program. Solution? Global variable or modularization/encapsulation.

```
function fun1() {
  var v1 = 1;

  function fun2() {
    var v1 = -1;
    fun3();
  }

  function fun3() {
    console.log(v1);
  }

  fun2();
}
```

```
function fun1() {
  var v1 = 1;

  function fun2() {
    var v1 = -1;

    function fun3() {
      console.log(v1);
    }
    fun3();
  }

  fun2();
}
```

Problem – Dynamic Scoping

- Programs hard to read and understand.
  - The correct attributes (e.g., values) of non-local variables cannot be determined statically (e.g., by reading).
  - Reference to the name of a variable is not always to the same variable.
- Type checking for non-local variables.
- Access to non-local variable is generally slower.

Lifetime

Lifetime of a variable The time period in which the object has valid memory, i.e., the period during execution of a program in which a variable or function exists. The *storage duration* of the identifier determines its lifetime.

Generall speaking

$$\text{Lifetime} \geq \text{Scope}$$

Static Variable Lifetime

A *static variable* is stored in the *data segment* of the "object file" of a program. Its lifetime is the entire duration of the program's execution.

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

int foo() {
  static int a = 0;
  a++;
  return a;
}

int main() {
  cout << foo() << endl;  // output?
  cout << foo() << endl;  // output?
}
```

Automatic Variable Lifetime

- Begins when program execution enters the function or statement block or compound and
- ends when execution leaves the block.

Automatic variables are stored on a "function call stack".

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

int foo() {
  int a = 0;
  ++a;
  return a;
}

int main() {
  cout << foo() << endl;  // output?
  cout << foo() << endl;  // output?
}
```

Dynamic Variable Lifetime

- Begins when memory is allocated for the object and
- ends when memory is deallocated.

Dynamic objects are stored on "the heap".

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

int foo() {
  int* a = new int{3};
  int* b = new int{3};
  ++(*a);
  delete b;
  return *a;
}
int main() {
  cout << foo() << endl;  // output?
  cout << foo() << endl;  // output?
}
```

Declaration vs Definition

Declaration Tells the compiler the type of a variable, object or function.

Definition Allocates memory for a variable or object or implement the details of a function.

Multiple declarations are allowed, but only one definition.

```
void foo();  // declaration of foo
int main() {
  int a;          // declaration of a;
  a = 3;          // definition of a
}

void foo() { }  // definition of foo
```

Outline

Summary

- Variable names, naming styles
- Variable scope and lifetime.