Computer Programming

Variable Scope

Variables in Functions

```
int main( )
{
  int mass=15
  double velo
  double ener
    …
}
```

```
double kenergy(int m, double v)
{
  double ke;
  ke = 0.5*m*v*v;
  return ke;
}
```

- Using variables
  - Variable name, type, size, and value
    - A variable by default is *created* when the program enters its block and *destroyed* when the program leaves its defining block
    - A variable by default can be accessed in the block and *from the point* where it is declared

  ① Scope (visibility)
    - Where a variable can be accessed in the program

  ② Storage class (persistence)
    - How long a variable exists in memory

  ③ Linkage
    - Whether a variable is known only in the source file it is declared or across multiple files that are linked together

① Scope

- Scope
  - The region in your code in which a declaration is active
  - Function scope (local variable)
    - Declared inside a function – Not valid outside the function
  - Program (file) scope (global variable)
    - Declared outside *any* function (including `main()` function)
  - Block scope
    - Declared inside a block (of a function) enclosed by `{ }`
    - Not valid outside the block

☞ Shadowing

- Identical variable names can exist in *different scopes*
- The outer variable is shadowed (not directly accessible)

Scope Example

For nested local scopes, the scope resolution operator provides access to only the global identifiers (not next outermost scopes)

```
#include <iostream>
using namespace std;
int days_in_month=31;

void april()
{
    int birthday=2;
    days_in_month=30;
}

int main( )
{
    int birthday=0;
    cout<<"Before: days_in_month="<< days_in_month <<endl
        <<"birthday="<< birthday <<endl;

    april();
    cout<<"After: days_in_month="<< days_in_month <<endl
        <<"birthday="<< birthday <<endl;
}
```

This variable cannot be seen inside function `main()`

Access the global variable (declared in Line 3) stored in the DATA segment

This variable cannot be seen inside function `april()`

Use the scope resolution operator `::days_in_month` to access global variable in case of shadowing

② Storage Class

To declare a register variable:

```
register int x;
```

- Persistent variables
  - Space used by variables declared inside a function is freed when the function returns
  - How to keep the value/space of variables when the function returns?
  - ☞ static variables
- Static storage class
  - The variable is *allocated statically* and exist for the duration of the program
- ☞ Other storage classes: auto and register
  - By default all local variables are of auto storage class
  - Register is a small, high-speed storage place in the CPU

Static Variables

Global variables are allocated statically

- Static variable
  - A static variable exists for the entire program
    - It keeps its value between function calls
  - A static variable is initialized only once in a program
    - C++ initializes static variables to zero by default
    - When explicitly initialized in the declaration, a static variable is assigned the value the first time its function is called, but not initialized again on subsequent calls
  - ☞ Scope vs. lifetime
    - Existence does not mean accessibility
    - Static variables declared in a function are known only in its own function
    - ☞ Global vs. local static variables

Using Static Variable (1/2)

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

double max_value(double);

int main( )
{
    double num, max=-1;

    while (1)
    {
        cout << "Enter any positive number; negative value to stop: ";
        cin >> num;
        cout<< "old max=" << max << "\tinput number=" << num;
        if (num<=0) break;
        else        max = max_value(num);
        cout<< "\tnew max=" << max <<endl;
    }
}
```

Using Static Variable (2/2)

```
double max_value(double num)
{
    static double max=-1;

    max = (num>max) ? num : max;

    return max;
}
```

What if "`static`" is not used?

- Another use of static variables (*return by reference*)
  - The caller by default only gets the *value* of the returned variable (which is destroyed after the callee returns)
  - Return by reference allows the caller to access the returned variable itself
    - A static local variable can be used for this purpose

```
double& max_value(double num) {…; return max;}
```

`return` Revisited

The function `max()` returns the **value** of variable `m` to function `main()` by copying the value to, say, a register or a memory location prepared by the caller

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

int max(int n1, int n2)
{
    int m;
    m = n1>n2 ? n1 : n2;
    return m;
}
int main( )
{
    int num1, num2, m;

    cout << "Enter two integer numbers:";
    cin >> num1 >> num2;

    m = max(num1, num2);
    cout << "The max of " << num1 << " and " << num2 << " is "
         << m << endl;
}
```

This variable `m` is visible only in the function `max()`

This variable `m` is created (or destroyed) when the control enters (or leaves) the function `max()`

This variable `m` is visible only in the function `main()`

Return by Reference

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

int& max(int& n1, int& n2);

int main()
{
    int x=3, y=2;

    max(x,y) = 5;

    cout << "x=" << x << ", y=" << y << endl;
}

int& max(int& n1, int& n2)
{
    if (n1>n2) return n1;
    else       return n2;
}
```

The function max() returns a reference to a variable, instead of returning a value of a variable

Note the syntax here

Why should the function max() use "call by reference" here rather than "call by value"?

Example on Variables (1/3)

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

void useLocal();
void useStaticLocal();
void useGlobal();

int x = 1;
int main()
{
    int x = 5;
    cout << "local x in main's outer scope is " << x << endl;

    { // start new scope
        int x = 7;
        cout << "local x in main's inner scope is " << x << endl;
    } // end new scope
```

Example on Variables (2/3)

```
    cout << "local x in main's outer scope is " << x << endl;
    useLocal();
    useStaticLocal();
    useGlobal();
    useLocal();
    useStaticLocal();
    useGlobal();

    cout << "\nlocal x in main is " << x << endl;
}

void useLocal( void )
{
    int x = 25;
    cout << endl << "local x is " << x
         << " on entering useLocal" << endl;
    ++x;
    cout << "local x is " << x
         << " on exiting useLocal" << endl;
}
```

Example on Variables (3/3)

```
void useStaticLocal( void )
{
    static int x = 50;
    cout << endl << "local static x is " << x
         << " on entering useStaticLocal" << endl;
    ++x;
    cout << "local static x is " << x
         << " on exiting useStaticLocal" << endl;
}

void useGlobal( void )
{
    cout << endl << "global x is " << x
         << " on entering useGlobal" << endl;
    x *= 10;
    cout << "global x is " << x
         << " on exiting useGlobal" << endl;
}
```

③ Linkage

Global variables by default have program scope

- Using variables across multiple files
  - Program and file scopes differ for a *multi-file* program
  - Use "extern" to access program-scope variables defined in a different file

file1.cpp

```
int x;
void func1() {
   cout << "x=" << x << endl;
}
```

file2.cpp

```
extern int x;
void func2() {
   cout << "x=" << x << endl;
}
```

☞ Use "static" to confine global variables in file scope

```
static int x;
void func1() {
   cout << "x=" << x << endl;
}
```

```
static int x;
void func2() {
   cout << "x=" << x << endl;
}
```

Review

- Writing functions
  - Function declaration, definition, and call
  - Passing arguments
  - Returning values
  - Overloading function names
- Function call stack
  - Inline and recursive functions
- Variable scope and storage class