C++ Program Design
-- Variable Scope and More Types

Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

Blocks (compound statements) and Local variables, scope, and duration

Blocks (compound statements)

- A **block** of statements, also called a **compound statement**,

```
int main()
{ // start outer block

    int x(5);

    { // start nested block
        int y(7);
        // we can see both x and y from here
        std::cout << x << " + " << y << " = " << x + y;
    } // y destroyed here

    // y can not be used here because it was already destroyed!

    return 0;
} // x is destroyed here
```

Local variables, scope, and duration

- A variable’s **scope** determines where a variable is accessible.
- A variable’s **duration** determines where it is created and destroyed.

- Variables defined inside a block are called **local variables**
  - Local variables have block scope
  - Local variables have automatic duration

```
int main()
{
    int i(5); // i created and initialized here
    double d(4.0); // d created and initialized here
    return 0;
} // i and d go out of scope and are destroyed here
```

scope & nested block

```
int main()
{ // start outer block

    int x(5);

    { // start nested block
        int y(7);
        // we can see both x and y from here
        std::cout << x << " + " << y << " = " << x + y;
    } // y destroyed here

    // y can not be used here because it was already destroyed!

    return 0;
} // x is destroyed here
```

functions can have variables or parameters with the same names as other functions.

```
// add’s x can only be seen/used within function add()
int add(int x, int y){ // add’s x is created here
    return x + y;
} // add’s x is destroyed here

// main’s x can only be seen/used within function main()
int main(){
    int x = 5; // main’s x is created here
    int y = 6;
    std::cout << add(x, y) << std::endl; // the value from main’
s x is copied into add’s x
    return 0;
} // main’s x is destroyed here
```

name hiding or shadowing

```
int main(){ // outer block
    int apples(5);

    if (apples >= 5){ // nested block
        int apples(10); // hides previous variable named apples
        std::cout << apples;
    } // nested block apples destroyed

    // apples now refers to the outer block apples
    std::cout << apples; return 0;
} // outer block apples destroyed
```

- variables inside nested blocks with same name as variable inside outer blocks.
- Then the nested variable “hides” the outer variable. **Should be avoided!**

limiting the scope of a variable

```
int main(){
    // do not define y here
    {
        // y is only used inside this block, so define it here
        int y(5);
        cout << y;
    }
    // otherwise y could still be used here
    return 0;
}
```

- reduce complexity since number of active variables is reduced.
- easier to see where variables are used.
- make the program easier to understand.

- *Rule: Define variables in the smallest scope possible*.
- *Rule: Avoid using nested variables with the same names as variables in an outer block*.

Function parameters

- Although function parameters are not defined inside the block belonging to the function, in most cases, they can be considered to have block scope.

```
int max(int x, int y) // x and y defined here
{
    // assign the greater of x or y to max
    int max = (x > y) ? x : y; // max defined here
    return max;
} // x, y, and max all die here
```

Quiz

- Write a program that asks the user to enter two integers, the second larger than the first. If the user entered a smaller integer for the second integer, use a block and a temporary variable to swap the smaller and larger values. Then print the value of the smaller and larger variables. Add comments to your code indicating where each variable dies.

- The program output should match the following:

```
Enter an integer: 4
Enter a larger integer: 2
Swapping the values
The smaller value is 2
The larger value is 4
```

Global variables and linkage

Global variables

```
int g_x; // global variable g_x
const int g_y(2); // global variable g_y

int doSomething(){
    g_x = 3; // global variables can be seen everywhere in program
    std::cout << g_y << ″\n″;
}

int main(){
    g_x = 5; // global variables can be seen everywhere in program
    std::cout << g_y << ″\n″;
     return 0;}
```

- **static duration**: created when the program starts & destroyed when it ends.
- **global scope** (also called “global namespace scope” or “file scope”): visible until the end of the file

Global variables

- **name hiding**
  - global scope operator (::)
  - int g_x;

```
int value(5); // global variable
int main(){
    int value = 7; // hides the global variable value
    value++; // increments local value, not global value
    ::value--; // decrements global value, not local value

    std::cout << "global value: " << ::value << "\n";
    std::cout << "local value: " << value << "\n";
    return 0;
} // local value is destroyed
```

Internal and external linkage via the static and extern keywords

- variables' properties: Scope, duration, linkage
- Local variables: no linkage
- Global const variables: internal linkage
  - `const int g_x; // g_x is const, and can only be used within this file`
  - `static int g_y;// g_y is static, and can only be used within this file`
- Global non-const variables: external linkage
  - `double g_x(9.8); // g_x is external, and can be used by other files`
  - `extern double g_y(9.8); // g_y is external, and can be used by other files`

- Keyword static will be explained later

Variable forward declarations via the extern keyword

- use an external global variable that has been declared in another file, you have to use a variable **forward declaration**

```
// define two global variables in glob
al.cpp
int g_x;
int g_y(2);
```

```
extern int g_x; // forward declaration for g_x
int main(){
    extern int g_y; // forward declaration for g_y
    g_x = 5;
    std::cout << g_y; // should print 2
    return 0;
}
```

- If a variable is declared as static, trying to use a forward declaration to access it will not work: will not find it because it has internal linkage.

Side note on function linkages

- Functions always default to external linkage, but can be set to internal linkage via the static keyword:

```
// This function is declared as static, and can now be used on
ly within this file
// Attempts to access it via a function prototype will fail
static int add(int x, int y)
{
    return x + y;
}
```

Global symbolic constants

- constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H
// define your own namespace to hold constants
namespace Constants
{
    const double pi(3.14159);
    const double avogadro(6.0221413e23);
    const double my_gravity(9.2); // m/s^2 -- gravity is light
on this planet
    // ... other related constants
}
#endif
```

Global symbolic constants

- constants.h:

```
namespace Constants{
    const double pi(3.14159);
    const double avogadro(6.0221413e23);}
```

- if constants.h gets included into 20 different code files, each of these variables is duplicated 20 times.
- Header guards won’t stop this from happening
- If they are lots of memory-intensive variables, …

- constants.cpp:

```
namespace Constants{// actual global variables
    extern const double pi(3.14159);
    …
}
```

- constants.h:

```
namespace Constants{// forward declarations only
    extern const double pi;
    …}
```

- main.cpp

```
#include ″constants.h″
double circumference = 2 * radius * Constants::pi;
```

A word of caution about (non-const) global variables

- New programmers are often tempted to use lots of global variables, because they are easy to work with, especially when many functions are involved.

- However, use of non-const global variables should generally be avoided altogether!

Summary

- **Scope** determines where a variable is accessible. **Duration** determines where a variable is created and destroyed. **Linkage** determines whether the variable can be exported to another file or not.

- Global variables can have either **internal or external linkage**, via the **static and extern keywords** respectively.

Why (non-const) global variables are evil

```
int g_mode;
void doSomething(){
    g_mode = 2;
}

int main(){
    g_mode = 1;
    doSomething();
    // we expects g_mode to be 1. But doSomething changed it to 2!

    if (g_mode == 1)
        cout << "No threat detected." << endl;
    else
        cout << "Launching nuclear missiles..." << endl;

    return 0;}
```

A joke

What’s the best naming prefix for a global variable?

Answer: //

Static duration variables

- The static keyword is one of the most confusing keywords in the C++ language.
- it has different meanings depending on where it is used.
  - Static global variable => internal linkage
  - Static local variable => static variable, static duration
- Static variables: created (& initialized) once, then persisted throughout the life of the program. (example in next page)

- “g_”: global variables, “s_”: static variables
  - internal linkage global variables (also declared using the static keyword) get a “g_”, not a “s_”.

Static variables

```
void incrementAndPrint(){
    static int s_value = 1; // static duration via static keywor
d.   This line is only executed once.
    ++s_value;
    std::cout << s_value << endl;
} // s_value is not destroyed here, but becomes inaccessible

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

most common uses for static variables

- Unique identifier generators

```
int generateID()
{
    static int s_itemID = 0;
    return s_itemID++;
// makes copy of s_itemID, increments the real s_itemID,
// then returns the value in the copy
}
```

Quiz

- What effect does using keyword “static” have on a global variable? What effect does it have on a local variable?

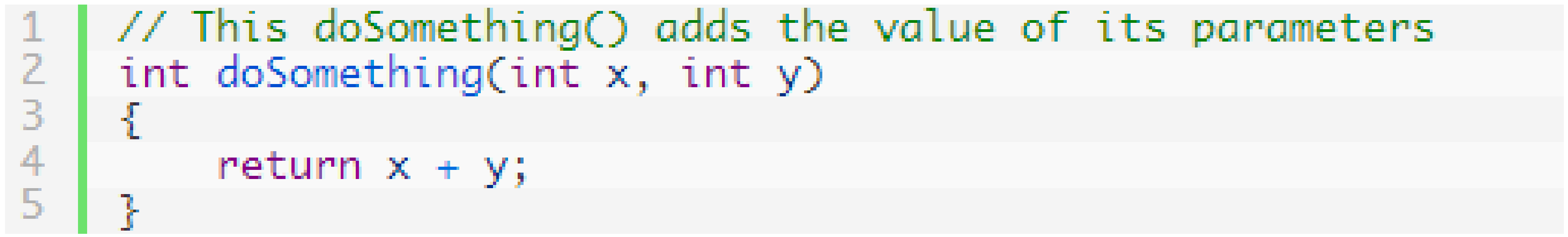
- Ans:
  - When applied to a global variable, the static keyword defines the global variable as having internal linkage, meaning the variable cannot be exported to other files.
  - When applied to a local variable, the static keyword defines the local variable as having static duration, meaning the variable will only be created once, and will not be destroyed until the end of the program.

Namespaces

Namespaces & naming collision

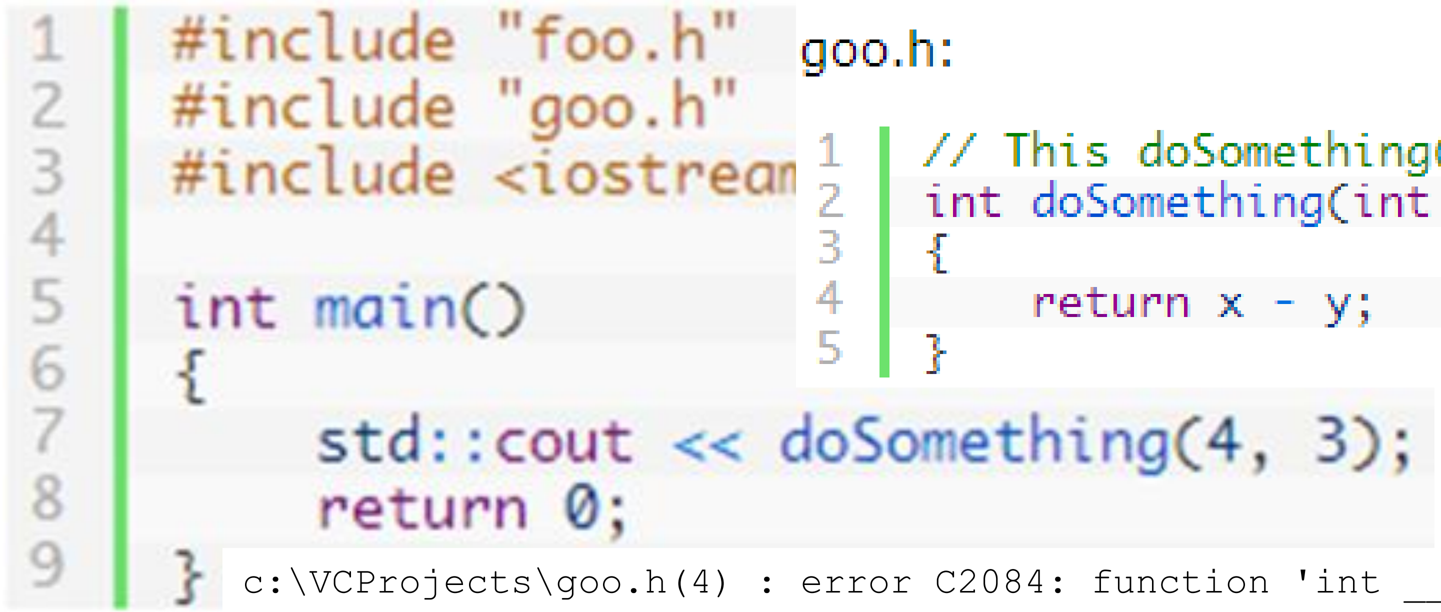
- As programs get larger and larger, the number of identifiers increases linearly, which in turn causes the probability of naming collisions to increase exponentially.

foo.h:



```cpp
// This doSomething() adds the value of its parameters
int doSomething(int x, int y)
{
    return x + y;
}
```

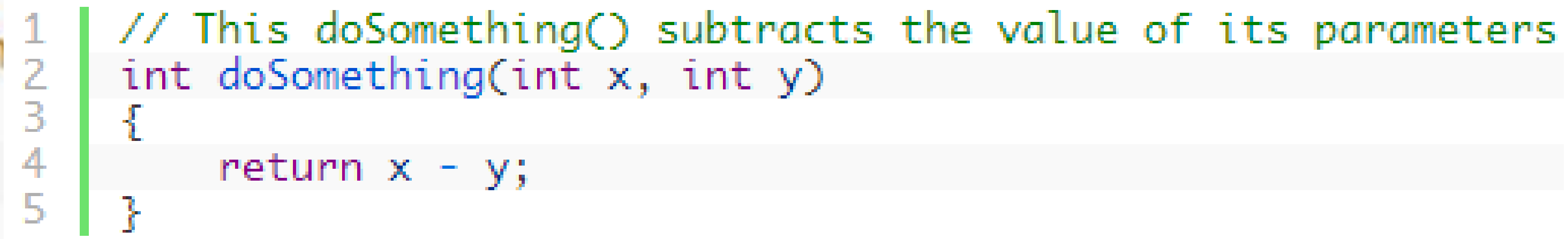
main.cpp:



```cpp
#include "foo.h"
#include "goo.h"
#include <iostream

int main()
{
    std::cout << doSomething(4, 3);
    return 0;
}
```

goo.h:



```cpp
// This doSomething() subtracts the value of its parameters
int doSomething(int x, int y)
{
    return x - y;
}
```

```
c:\VCProjects\goo.h(4) : error C2084: function 'int __cdecl doSomething(int,int)' already has a body
```

global namespace

- global variables and normal functions are defined in the **global namespace**

```
int g_x = 5;

int foo(int x)
{
    return -x;
}
```

- Both g_x and foo() are defined in the global namespace.

declare our own namespaces

```
namespace Foo{
    int doSomething(int x, int y)
     {
          return x + y;
     }
}
```

```
namespace Goo{
    int doSomething(int x, int y)
     {
          return x - y;
     }
}
```

```
    int main(void)
    {
         std::cout << Foo::doSomething(4, 3) << std::endl;
         std::cout << Goo::doSomething(4, 3) << std::endl;
         return 0;
    }
```

The using keyword 1

```
int main(void)
{
    using namespace Foo; // look in namespace Foo
    std::cout << doSomething(4, 3) << std::endl; // calls Foo::d
oSomething()
    return 0;
}
```

The using keyword 2

```
int main(void)
{
    using Foo::doSomething; // Tell compiler that doSomething() means Foo::doSomething()

    std::cout << doSomething(4, 3) << std::endl; // will resolve to Foo::doSomething()

    return 0;
}
```

Rule: Don’t use the “using” keyword in the global scope.

Cancelling or replacing a using statement

```
int main()
{
    using namespace Foo;

    // there's no way to cancel the "using namespace Foo" here!
    // there's also no way to replace "using namespace Foo" with
"using namespace Goo"!

    return 0;
} // using namespace Foo ends here
```

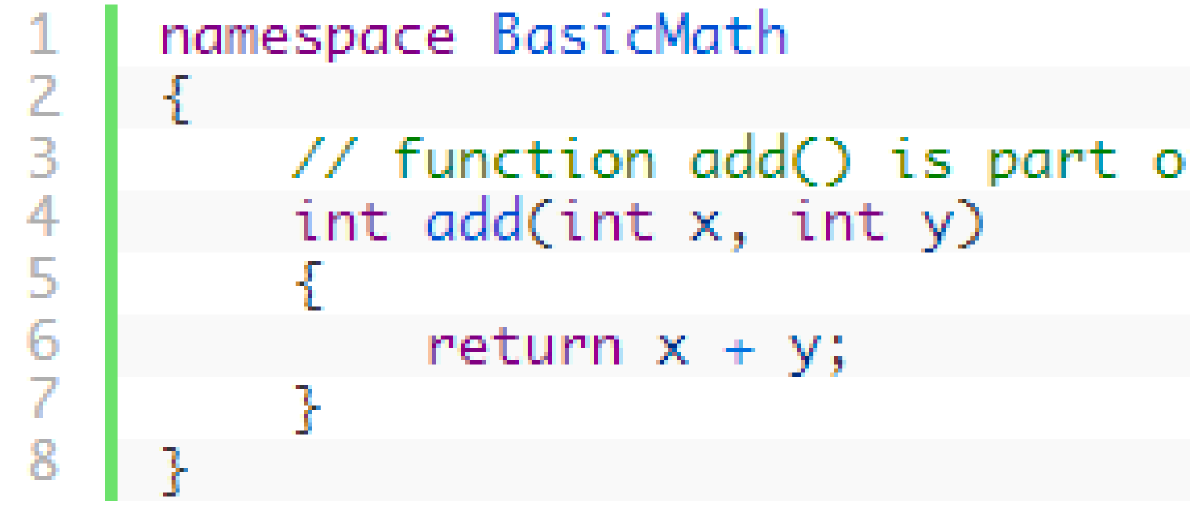
Cancelling or replacing a using statement

```
int main()
{
    {
        using namespace Foo;
        // calls to Foo:: stuff here
    } // using namespace Foo expires
    {
        using namespace Goo;
        // calls to Goo:: stuff here
    } // using namespace Goo expires

    return 0;
}
```

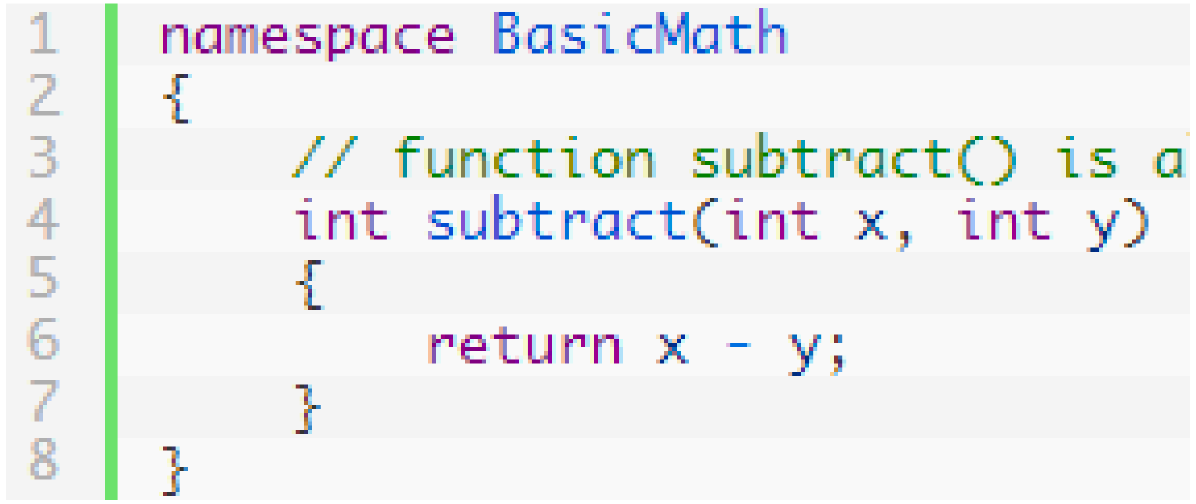
Multiple namespace blocks with the same name allowed

add.h:



```cpp
namespace BasicMath
{
    // function add() is part o
    int add(int x, int y)
    {
        return x + y;
    }
}
```

subtract.h:



```cpp
namespace BasicMath
{
    // function subtract() is a
    int subtract(int x, int y)
    {
        return x - y;
    }
}
```

Nested namespaces and namespace aliases

```cpp
namespace Foo{
    namespace Goo{
        const int g_x = 5;
    }
}
namespace Boo = Foo::Goo; // Boo now refers to Foo::Goo

int main(){
    std::cout << Boo::g_x; // This is really Foo::Goo::g_x
    return 0;
}
```

In general, you should avoid nesting namespaces if possible, and there's few good reasons to nest them more than 2 levels deep.

type conversion: implicit & explicit

Implicit type conversion

- what happens when we do something like this?
- `float f = 3; // initialize floating point variable with integer 3`
  - **type conversion:** convert the integer 3 to a floating point number,
  - then be assigned to variable f
- Type conversions:

```
void doSomething(long l){}
doSomething(3); // pass integer value 3 to a function expectin
g a long parameter

float doSomething(){
    return 3.0; // Pass double value 3.0 to a function that r
eturns a float
}
```

Implicit & explicit type conversion

- **Implicit:** compiler automatically transforms one fundamental data type into another
- **Explicit:** developer uses a casting operator to direct the conversion

Numeric promotion

- Whenever a value from one type is converted into a value of a larger similar data type, this is called a **numeric promotion** (or **widening**)

```
long l(64); // widen the integer 64 into a long
double d(0.12f); // promote the float 0.12 into a double
```

- always safe, and no data loss will result.

Numeric conversions

- When we convert a value from a larger type to a similar smaller type, or between different types, this is called a **numeric conversion**
- may result in a loss of data

```
double d = 3; // convert integer 3 to a double, safe
char c = 3000; // convert integer 3000 to a char, dangerous
```

Evaluating arithmetic expressions

- compiler breaks each expression down into individual subexpressions
- arithmetic operators require their operands to be of the same type
- compiler will implicitly convert one operand to agree with the other using a process called **usual arithmetic conversion**

```
#include <typeinfo> // for typeid()
int main(){
    double d(4.0);
    short s(2);
    std::cout << typeid(d + s).name() << ” ” << d + s << std::en
dl; // show us the type of d + s

    return 0;
}
```

double 6.0

Explicit type conversion (casting)

float f = 10 / 4;

```
int i1 = 10;
int i2 = 4;
float f = i1 / i2;
```

1. because 10 and 4 are both integers, no promotion takes place.
2. Integer division is performed on 10 / 4,
3. resulting in the value of 2,
4. which is then implicitly converted to 2.0 and assigned to f!

- C-style cast with a more function-call like syntax

```
int i1 = 10;
int i2 = 4;
float f = (float)i1 / i2;
```

static_cast

- *Rule: Avoid C-style casts*
  - C-style casts are not checked by the compiler at compile time,
  - C-style casts can be inherently misused,

```
int i1 = 10;
int i2 = 4;
float f = static_cast<float>(i1) / i2;

int i = 48;
char ch = i; // implicit conversion
```

compiler will complain

```
int i = 48;
char ch = static_cast<char>(i);
```

No complain

An introduction to std::string

- **What is a string?**

```
#include <iostream>
int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

- a collection of sequential characters called a **string**

std::string

```
#include <string>
std::string myName("Alex"); // initialize myName with string literal "Alex"
myName = "John"; // assign variable myName the string literal "John “
std::string myID("45"); // "45" is not the same as integer 45!
```

```
int main()
{
    std::cout << "Enter your full name: ";
    std::string name;
    std::cin >> name; // this won't work as expected since std::
cin breaks on whitespace

    std::cout << "Enter your age: ";
    std::string age;
    std::cin >> age;

    std::cout << "Your name is " << name << " and your age is "
<< age;
}
```

```
Enter your full name: John Doe
Enter your age: Your name is John and your age is Doe
```

Use std::getline() to input text

```
int main()
{
    std::cout << "Enter your full name: ";
    std::string name;
    std::getline(std::cin, name); // read a full line of text into name

    std::cout << "Enter your age: ";
    std::string age;
    std::getline(std::cin, age); // read a full line of text into age

    std::cout << "Your name is " << name << " and your age is " 
<< age;
}
```

Mixing std::cin and std::getline()

```
int main(){
std::cout << "Pick 1 or 2: ";
int choice;
std::cin >> choice;

std::cout << "Now enter your name: ";
std::string name;
std::getline(std::cin, name);

std::cout << "Hello, " << name << ", you picked " << choice << '\n';

return 0;
}
```

- Hello, , you picked 2

Mixing std::cin and std::getline()

```
int main(){
std::cout << "Pick 1 or 2: ";
int choice;
std::cin >> choice;

std::cin.ignore(32767, '\n'); // ignore up to 32767 characters until a \n is removed

std::cout << "Now enter your name: ";
std::string name;
std::getline(std::cin, name);

std::cout << "Hello, " << name << ", you picked " << choice << '\n';
return 0;}
```

- *Rule: If reading numeric values with std::cin, it's a good idea to remove the extraneous newline using std::cin.ignore().*

Appending strings

```
int main()
{
    std::string a("45");
    std::string b("11");

    std::cout << a + b << "\n"; // a and b will be appended, not added
    a += " volts";
    std::cout << a;

    return 0;
}
```

```
4511
45 volts
```

String length

```
#include <string>
#include <iostream>
int main()

{
    std::string myName("Alex");
    std::cout << myName << " has " << myName.length() << " chara
cters\n";
    return 0;
}
```

- Alex has 4 characters

Conclusion

- std::string is complex (we haven’t covered yet).
- Fortunately, you don’t need to understand these complexities to use std::string for simple tasks

- We encourage you to start experimenting with strings now, and we’ll cover additional string capabilities later.

enumeration

Enumerated types

- An **enumerated type** (also called an **enumeration**) is a data type where every possible value is defined as a symbolic constant (called an **enumerator**)

```
enum Color{    // Here are the enumerators
    COLOR_BLACK, // Each enumerator is separated by a comma, not a semicolon
    COLOR_RED,
    COLOR_BLUE// the last enumerator should not have a comma
}; // however the enum itself must end with a semicolon

// Define a few variables of enumerated type Color
Color paint = COLOR_WHITE;
Color house(COLOR_BLUE);
```

Naming enums

```
enum Color
{
RED,
BLUE, // BLUE is put into the global namespace
GREEN
};

enum Feeling
{
HAPPY,
TIRED,
BLUE // error, BLUE was already used in enum Color in the global namespace
};
```

Enumerator values

- Each enumerator is automatically assigned an integer value based on its position in the enumeration list

```
enum Color
{
    COLOR_BLACK, // assigned 0
    COLOR_RED, // assigned 1
    COLOR_BLUE, // assigned 2
    COLOR_GREEN, // assigned 3
    COLOR_WHITE // assigned 4
};

Color paint(COLOR_WHITE);
std::cout << paint;
```

Enumerator values

- Each enumerator is automatically assigned an integer value based on its position in the enumeration list

```
enum Animal
{
    ANIMAL_CAT = -3,
    ANIMAL_DOG, // assigned -2
    ANIMAL_PIG, // assigned -1
    ANIMAL_HORSE = 5,
    ANIMAL_GIRAFFE = 5, // shares same value as ANIMAL_HORSE
    ANIMAL_CHICKEN // assigned 6
}
```

Enum type evaluation and input/output

- Because enumerated values evaluate to integers, they can be assigned to integer variables.
- `int mypet = ANIMAL_PIG;`
- `std::cout << ANIMAL_HORSE;// evaluates to integer before being passed to std::cout`

- The compiler will *not* implicitly convert an integer to an enumerated valu e.
- `Animal animal = 5; // will cause compiler error`

Enum type evaluation and input/output

```
enum Color
{
    COLOR_BLACK, // assigned 0
    COLOR_RED, // assigned 1
    COLOR_BLUE
};

Color color;
std::cin >> color; // will cause compiler error

 int inputColor;
 std::cin >> inputColor;

 Color color = static_cast<Color>(inputColor);
```

What are enumerators useful for?

```
int readFileContents()
{
    if (!openFile())
        return -1;
    if (!readFile())
        return -2;
    if (!parseFile())
        return -3;

    return 0; // success
}
```

```
enum ParseResult{
    SUCCESS = 0,    ERROR_OPENING_FILE = -1,
    ERROR_READING_FILE = -2,    ERROR_PARSING_FILE = -3
};

ParseResult readFileContents()
{
    if (!openFile())
        return ERROR_OPENING_FILE;
    if (!readFile())
        return ERROR_READING_FILE;
    if (!parsefile())
        return ERROR_PARSING_FILE;

    return SUCCESS;
}
```

- using magic numbers like this isn’t very descriptive.

Enumerated types are best used when defining a set of related identifiers

```
enum ItemType
{
    ITEMTYPE_SWORD,
    ITEMTYPE_TORCH,
    ITEMTYPE_POTION
};
```

```
std::string getItemName(ItemType itemType){
    if (itemType == ITEMTYPE_SWORD)
        return std::string("Sword");
    if (itemType == ITEMTYPE_TORCH)
        return std::string("Torch");
    if (itemType == ITEMTYPE_POTION)
        return std::string( "Potion");
}
int main(){
    ItemType itemType(ITEMTYPE_TORCH);

    std::cout << "You are carrying a " << getItemName(itemType) << "\n";

    return 0;
}
```

Quiz

- 1) Define an enumerated type to choose between the following monster types: orcs, goblins, trolls, ogres, and skeletons.
- 2) Declare a variable of the enumerated type you defined in question 1 and assign it the troll type.

Quiz: True or false. Enumerators can be:

3a) assigned integer values
3b) not assigned a value
3c) explicitly assigned floating point values
3d) negative
3e) non-unique

3a) True.
3b) True.
3c) False.
3d) True.
3e) True

Enum classes

- enumerated types are not type safe, allow you to do things that don’t make sense.

```
enum Color                    enum Fruit
    {                         {
        RED,                      BANANA,
        BLUE                      APPLE
    };                        };

int main() {
    Color color = RED;
    Fruit fruit = BANANA;

    if (color == fruit) // The compiler will compare a and b as integers
        std::cout << "color and fruit are equal\n"; // and find they are equal!
    else
        std::cout << "color and fruit are not equal\n";
  return 0;
  }
```

Enum classes

- C++11 defines a new concept, the enum class (also called a scoped enumeration)

```
enum class Color                        enum class Fruit
{                                       {
    RED,                                    BANANA,
    BLUE                                    APPLE
};                                      };

int main(){
    Color color = Color::RED;
    Fruit fruit = Fruit::BANANA;

    if (color == fruit) // compile error here, as the compiler doesn't know how to compare different ty
pes Color and Fruit
        std::cout << "color and fruit are equal\n";
    else
        std::cout << "color and fruit are not equal\n";
   return 0;
   }
```

- can still compare enumerators from within the same enum classcan still compare enumerators from within the same enum class

```
Color color = Color::RED;
if (color == Color::RED) // this is okay
   std::cout << "The color is red!\n";
else if (color == Color::BLUE)
   std::cout << "The color is blue!\n";
```

- With enum classes, the compiler will no longer implicitly convert enumerator values to integers.

```
Color color = Color::BLUE;
std::cout << color; // won't work, because there's no implicit conversion to int
std::cout << static_cast<int>(color); // will print 1
```

- If you're using a C++11 compiler, there's really no reason to use normal enumerated types instead of enum classes.

Typedefs

- create an alias for a data type

```
typedef double distance_t; // define distance_t as an alias for type double
// The following two statements are equivalent:
double howFar;
distance_t howFar;
```

Using typedefs for legibility

- One use for typedefs is to help with documentation and legibility.

```
int GradeTest();
```

- the return value is an integer, but what does the integer mean?

```
typedef int testScore_t;
testScore_t GradeTest();
```

Using typedefs for easier code maintenance

- change the underlying type of an object without having to change lots of code

- Originally: short studentID_t
- Now, you wish: long studentID_t

- Many search and replacement operations v.s.
- typedef short studentID_t => typedef long studentID_t

Platform independent coding

```
#ifdef INT_2_BYTES
typedef char int8_t;
typedef int int16_t;
typedef long int32_t;
#else
typedef char int8_t;
typedef short int16_t;
typedef int int32_t;
#endif
```

Using typedefs to make complex types simple

```
typedef std::vector<std::pair<std::string, int> > pairlist_t;
pairlist_t pairlist;
boolean hasDuplicates(pairlist_t pairlist){
    // some code here
}
```

In c++11, they are equal

```
typedef double distance_t; // define distance_t as an alias for type double
using distance_t = double; // define distance_t as an alias for type double
```

cleaner for than typedefing

Rule: Use the using version of typedef instead of the typedef keyword if you compiler is C++11 compatible.

Structs

Declaring and defining structs

```
struct Employee
{
    short id;
    int age;
    double wage;
};
```

Accessing struct members

- **member selection operator** (which is a period)

```
Employee joe; // create an Employee struct for Joe
joe.id = 14; // assign a value to member id within struct joe
joe.age = 32; // assign a value to member age within struct joe
joe.wage = 24.15; // assign a value to member wage within struct joe

Employee frank; // create an Employee struct for Frank
frank.id = 15; // assign a value to member id within struct frank
frank.age = 28; // assign a value to member age within struct frank
frank.wage = 18.27; // assign a value to member wage within struct frank
```

- Struct member variables act just like normal variables, so it is possible to do normal operations on them

Initializing structs

```
Employee joe = { 1, 32, 60000.0 }; // joe.id = 1, joe.age = 32, joe.wage = 60000.0
Employee frank = { 2, 28 }; // frank.id = 2, frank.age = 28, frank.wage = 0.0 (default initialization)
```

- C++11

```
Employee joe { 1, 32, 60000.0 }; // joe.id = 1, joe.age = 32, joe.wage = 60000.0
Employee frank { 2, 28 }; // frank.id = 2, frank.age = 28, frank.wage = 0.0 (default initialization)
```

C++11/14: Non-static member initialization

```
struct Triangle{
    double length = 1.0;
    double width = 1.0;
};

int main(){
    Triangle x; // length = 1.0, width = 1.0

    x.length = 2.0; // you can assign other values like normal

    return 0;
}
```

Structs and functions

```cpp
void printInformation(Employee employee){
    std::cout << "ID:    " << employee.id << "\n";}

int main(){
    Employee joe = { 14, 32, 24.15 };
    Employee frank = { 15, 28, 18.27 };
    // Print Joe's information
    printInformation(joe);
    std::cout << "\n";
    // Print Frank's information
    printInformation(frank);
    return 0;
}
```

A function can also return a struct, which is one of the few ways to have a function return multiple variables.

```
struct Point3d
{
    double x;
    double y;
    double z;
};

Point3d getZeroPoint()
{
    Point3d temp = { 0.0, 0.0, 0.0 };
    return temp;
}
```

Nested structs

```
struct Employee
{
    short id;
    int age;
    float wage;
};

struct Company
{
    Employee CEO; // Employee is a struct within the Company struct
    int numberOfEmployees;
};

Company myCompany = {{ 1, 42, 60000.0f }, 5 };
```

Struct size and data structure alignment

- On many platforms, a short is 2 bytes, an int is 4 bytes, and a double is 8 bytes, so we'd expect Employee to be 2 + 4 + 8 = 14 bytes.

```
struct Employee{
    short id;
    int age;
    double wage;
};
int main(){
    std::cout << "The size of Employee is " << sizeof(Employee) << "\n";
    return 0;}
```

```
The size of Employee is 16
```

- The size of a struct will be *at least* as large as the size of all the variables it contains.
- But it could be larger! For performance reasons, the compiler will sometimes add gaps into structures (this is called **padding**).

can read about **data structure alignment** on Wikipedia

QUIZ

- 1) You are running a website, and you are trying to keep track of how much money you make per day from advertising. Declare an advertising struct that keeps track of how many ads you’ve shown to readers, what percentage of users clicked on ads, and how much you earned on average from each ad that was clicked. Read in values for each of these fields from the user. Pass the advertising struct to a function that prints each of the values, and then calculates how much you made for that day (multiply all 3 fields together).

QUIZ

- 2) Create a struct to hold a fraction. The struct should have an integer numerator and an integer denominator member. Declare 2 fraction variables and read them in from the user. Write a function called multiply that takes both fractions, multiplies them together, and prints the result out as a decimal number. You do not need to reduce the fraction to its lowest terms.

The auto keyword

Auto prior to C++11

```
int main()
{
    auto int foo(5); // explicitly specify that foo should have automatic duration

    return 0;
}
```

- explicitly specify that a variable should have automatic duration
- Since local variables have automatic duration, auto was entirely obsolet e.

Automatic type deduction in C++11

```
auto d = 5.0; // 5.0 is a double literal, so d will be type double
auto i = 1 + 2; // 1 + 2 evaluates to an integer, so i will be type int

int add(int x, int y)
{
    return x + y;
}

int main()
{
    auto sum = add(5, 6); // add() returns an int, so sum will be type int
    return 0;
}
```

Automatic type deduction for functions in C++14

- In C++14, the auto keyword was extended to be able to auto-deduce a function’s return type.

```
auto add(int x,  int y)
{
    return x + y;
}
```

- While this may seem neat, we recommend that this syntax be avoided for functions that return a fixed type.
- The return type of a function is of great use in helping to document a function.
- When a specific type isn’t specified, the user may not know what is expected.

Overview

- Block {}
- Local variables v.s. global variables
    - Scope, duration, linkage
    - Non-const global variables are evil.
- static keyword
    - give a global variable internal linkage
    - give a local variable static duration

Overview

- Block {}
- Local variables v.s. global variables
    - Scope, duration, linkage
    - Non-const global variables are evil.
- static keyword
    - give a global variable internal linkage
    - give a local variable static duration
- Namespace
- Implicit v.s. explicit type conversion
- String
- Enumerated types v.s. Enum classes
- Typedefs
- Struct

QUIZ

- 1) In designing a game, we decide we want to have monsters, because everyone likes fighting monsters. Declare a struct that represents your monster. The monster should have a type that can be one of the following: an ogre, a dragon, an orc, a giant spider, or a slime. If you’re using C++11, use an enum class for this. If you’re using an older compiler, use an enumeration for this.

- Each individual monster should also have a name, as well as an amount of health that represents how much damage they can take before they die. Write a function named printMonster() that prints out all of the struct’s members. Instantiate an ogre and a slime, initialize them using an initializer list, and pass them to printMonster().

- Your program should produce the following output:

- This Ogre is named Torg and has 145 health.
- This Slime is named Blurp and has 23 health.