The C in C++

김동원

2003.02.05

Overview

Introduction to data types

- ***Data types***
  - The way you use storage (memory) in the programs you write
  - By specifying a data type, you tell the compiler how to create a particular piece of storage, and also how to manipulate that storage
  - The types of built-in data are almost identical in C and C++
  - In contrast, a user-defined data type is one that you or another programmer create as a class

Built in data type

- **Define variables anywhere in a scope, and you can define and initialize them at the same time**

```
int main()
{
    // Definition without initialization:
    char protein;
    int carbohydrates;
    float fiber;
    double fat;
    // Simultaneous definition & initialization:
    char pizza = 'A', pop = 'Z';
    int dongdings = 100, twinkles = 150,
    heehos = 200;
    float chocolate = 3.14159;
    // Exponential notation:
    double fudge_ripple = 6e-4;
}
```

bool, true, & false

- The Standard C++ bool type can have two states
  - true
    - Converts to an integral one
  - false
    - Converts to an integral zero

Element	Usage with bool
&& || !	Take bool arguments and produce **bool** results.
< > <= >= == !=	Produce **bool** results.
if, **for**, **while**, **do**	Conditional expressions convert to **bool** values.
? :	First operand converts to **bool** value.

Specifiers

- Modify the meanings of the basic built-in types
- Expand them to a much larger set
- There are four specifiers
  - long, short, signed, and unsigned
- **long** and **short**
  - Modify the maximum and minimum values that a data type will hold
  - short int, int, long int
  - float, double, and long double
  - long float is not a legal type
  - There are no short floating-point numbers

Specifiers

- **signed** and **unsigned**
  - Tell the compiler how to use the sign bit with integral types and characters
  - An unsigned number does not keep track of the sign and thus has an extra bit available
  - signed number can store positive numbers twice as large as the positive numbers
  - char may or may not default to signed

Specifiers

```
#include <iostream.h>
int main()
{
    char c;
    unsigned char cu;
    int i;
    unsigned int iu;
    short int is;
    short iis; // Same as short int
    unsigned short int isu;
    unsigned short iisu;
    long int il;
    long iil; // Same as long int
    unsigned long int ilu;
    unsigned long iilu;
    float f;
    double d;
    long double ld;
```

Specifiers (example)

```
cout
    << "\n char= " << sizeof(c)
    << "\n unsigned char = " << sizeof(cu)
    << "\n int = " << sizeof(i)
    << "\n unsigned int = " << sizeof(iu)
    << "\n short = " << sizeof(is)
    << "\n unsigned short = " << sizeof(isu)
    << "\n long = " << sizeof(il)
    << "\n unsigned long = " << sizeof(ilu)
    << "\n float = " << sizeof(f)
    << "\n double = " << sizeof(d)
    << "\n long double = " << sizeof(ld)
    << endl;
}
```

Introduction to pointers

- **Whenever you run a program**
  - It is first loaded into the computer's memory
  - All elements of your program are located somewhere in memory
  - The size of each space depends on the architecture of the particular machine and is usually called that machine's *word size*
- ***pointer***
  - Special type of variable that holds an address

Introduction to pointers (example)

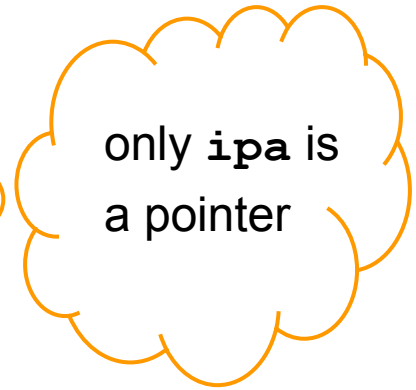
```
#include <iostream.h>
int dog, cat, bird, fish;
void f(int pet)
{
  cout << "pet id number: " << pet << endl;
}

int main()
{
    int i, j, k;
    cout << "f(): " << (long)&f << endl;
    cout << "dog: " << (long)&dog << endl;
    cout << "cat: " << (long)&cat << endl;
    cout << "bird: " << (long)&bird << endl;
    cout << "fish: " << (long)&fish << endl;
    cout << "i: " << (long)&i << endl;
    cout << "j: " << (long)&j << endl;
    cout << "k: " << (long)&k << endl;
}
```

Introduction to pointers

- **Pointer define**

```
int* ip; // ip points to an int
         // variable
int a, b, c;
int* ipa, ipb, ipc;
int* ipa;
int* ipb;
int* ipc;
```




```
int a = 47;
int* ipa = &a;
*ipa = 100;
```

Modifying the outside object

- ***pass-by-value*** (***call-by-value***)
  - When you pass an argument to a function, a copy of that argument is made inside the function
- ***pass-by-address*** (***call-by-address***)
  - When you pass an argument to a function, a alias of that argument is made inside the function

Pass-by-value (example)

```
#include <iostream.h>

void f(int a)
{
    cout << "a = " << a << endl;
    a = 5;
    cout << "a = " << a << endl;
}
int main()
{
    int x = 47;
    cout << "x = " << x << endl;
    f(x);
    cout << "x = " << x << endl;
}
```

```
x = 47
a = 47
a = 5
x = 47
```

Pass-by-address (example)

```
#include <iostream.h>

void f(int* p)
{
    cout << "p = " << p << endl;
    cout << "*p = " << *p << endl;
    *p = 5;
    cout << "p = " << p << endl;
}
int main()
{
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f(&x);
    cout << "x = " << x << endl;
}
```

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

Introduction to C++ references

- ***pass-by-reference*** (***call-by-reference***)
  - The basic idea is the same as the demonstration of pointer use above
    - You can pass the address of an argument using a reference
  - The difference between references and pointers
    - *Calling* a function that takes references
    - Syntactic difference that makes references essential in certain situations
  - Allows a function to modify the outside object, just like passing a pointer

Pass-by-refrence (example)

```
#include <iostream.h>
void f(int& r)
{
    cout << "r = " << r << endl;
    cout << "&r = " << &r << endl;
    r = 5;
    cout << "r = " << r << endl;
}
int main()
{
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f(x); // Looks like pass-by-value,
    cout << "x = " << x << endl;
}
```

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

void type pointers

- Any type of address at all can be assigned to that pointer
- Once you assign to a **void*** you lose any information about what type it is
- Before you can use the pointer, you must cast it to the correct type

void type pointers (example)

```
int main()
{
    void* vp;
    char c;
    int i;
    float f;
    double d;
    // The address of ANY type can be
    // assigned to a void pointer:
    vp = &c;
    vp = &i;
    vp = &f;
    vp = &d;
}
```

void type pointers (example)

```
int main()
{
   int i = 99;
   void* vp = &i;
   // Can't dereference a void pointer:
   // *vp = 3; // Compile-time error
   // Must cast back to int before dereferencing:
   *((int*)vp) = 3;
}
```

Scoping

- **Scoping rules**
  - Where a variable is valid
  - Where it is created
  - Where it gets destroyed

Scoping (example)

```
int main()
{
    int scp1;
    // scp1 visible here
    {
        // scp1 still visible here
        //.....
        int scp2;
        // scp2 visible here
        //.....
        {
            // scp1 & scp2 still visible here
            //..
            int scp3;
            // scp1, scp2 & scp3 visible here
            // ...
        }  // <-- scp3 destroyed here
```

Scoping (example)

```
        // scp3 not available here
        // scp1 & scp2 still visible here
        // ...
    } // <-- scp2 destroyed here
    // scp3 & scp2 not available here
    // scp1 still visible here
    //..
} // <-- scp1 destroyed here
```

Defining variables on the fly

- **The C Language**
  - Forces you to define all the variables at the beginning of a scope
  - Most people don't know all the variables they are going to use before they write the code
    - Jumping back to the beginning of the block to insert new variables
- **The C++ Language**
  - Allows you to define variables anywhere in a scope
  - Define a variable right before using it
  - Initialize the variable at the point that define it

Defining variables on the fly

- Define variables inside the control expressions
  - for loops and while loops
  - Inside the conditional of an if statement
  - Inside the selector statement of a switch

Defining variables on the fly (example)

```
#include <iostream.h>
int main()
{
    //..
    { // Begin a new scope
       int q = 0; // C requires definitions here
       //..
       // Define at point of use:
       for(int i = 0; i < 100; i++) {
       q++; // q comes from a larger scope
       // Definition at the end of the scope:
       int p = 12;
       }
       int p = 1; // A different p
    } // End scope containing q & outer p
    cout << "Type characters:" << endl;
```

Defining variables on the fly (example)

```
    while(char c = cin.get() != 'q') {
       cout << c << " wasn't it" << endl;
       if(char x = c == 'a' || c == 'b')
         cout << "You typed a or b" << endl;
       else
         cout << "You typed " << x << endl;
    }
    cout << "Type A, B, or C" << endl;
    switch(int i = cin.get()) {
    case 'A': cout << "Snap" << endl; break;
    case 'B': cout << "Crackle" << endl; break;
    case 'C': cout << "Pop" << endl; break;
    default: cout << "Not A, B or C!" << endl;
    }
}
```

Specifying storage allocation

- Global variables
- Local Variables
- Static
- Extern
- Constant
- Volatile

Global variables

- Defined outside all function bodies and are available to all parts of the program
- Unaffected by scopes and are always available
- If the existence of a global variable in one file is declared using the extern keyword in another file, the data is available for use by the second file

Local variables

- Local variables occur within a scope
- Called *automatic* variables
- Automatically come into being when the scope is entered and automatically go away when the scope closes
- The keyword *auto* makes this explicit, but local variables default to *auto*

Local variables

- **Register variables**
  - Tells the compiler “Make accesses to this variable as fast as possible”
  - Placing the variable in a register
  - There is no guarantee
    - The variable will be placed in a register
    - The access speed will increase
  - Hint to the compiler
  - A *register* variable can be declared only within a block
  - Cannot have global or *static register* variables
  - use a register variable as a formal argument in a function

static

- **A value to be extant throughout the life of a program**
- **Can define a function's local variable to be *static* and give it an initial value**
  - The initialization is performed only the first time the function is called
  - The data retains its value between function calls
- **You may wonder why a global variable isn't used instead**
  - The beauty of a static variable is that it is unavailable outside the scope of the function
    - it can't be inadvertently changed

static (example)

```
#include <iostream>
using namespace std;
void func()
{
   static int i = 0;
   cout << "i = " << ++i << endl;
}
int main()
{
   for(int x = 0; x < 10; x++)
   func();
}
```

extern

- Tells the compiler that a variable or a function exists
- This variable or function may be defined in another file or further down in the current file

extern (example)

```
#include <iostream>
using namespace std;
extern int i;
extern void func();
int main()
{
   i = 0;
   func();
}
int i;
void func()
{
   i++;
   cout << i;
}
```

Linkage

- **Storage as it is seen by the linker**
- **There are two types of linkage**
  - *internal linkage*
    - Storage is created to represent the identifier only for the file being compiled
    - Specified by the keyword static in C and C++
  - *external linkage*
    - A single piece of storage is created to represent the identifier for all files being compiled
    - The storage is created once, and the linker must resolve all other references to that storage
    - Global variables and function names have external linkage
    - Variables defined outside all functions (with the exception of const in C++)
    - Function definitions default to external linkage

Linkage

- **The linker doesn't know about automatic variables, and so these have *no linkage***
  - Automatic (local) variables exist only temporarily, on the stack, while a function is being called

Constants

- **In old C**
  - if you wanted to make a constant, you had to use the preprocessor
    - `#define PI 3.14159`
  - Everywhere you used PI, the value 3.14159 was substituted by the preprocessor
  - Don't change me
- **In C++**
  - You must specify the type of a **const**, like this
    - `const int x = 10;`
  - must always have an initialization value
- Have a scope, just like a regular variable

volatile

- **Tell the compiler**
  - You never know when this will change
  - Prevents the compiler from performing any optimizations based on the stability of that variable
- **Always read whenever its value is required, even if it was just read the line before**

Operators and their use

- Assignment
- Mathematical operators
- Relational operators
- Logical operators
- Bitwise operators
- Shift operators
- Unary operators
- The ternary operator
- The comma operator
- Common pitfalls when using operators
- Casting operators
- C++ explicit casts
- sizeof – an operator by itself
- The asm keyword
- Explicit operators

Assignment

- **Performed with the operator =**
- **Take the right-hand side (often called the *rvalue*) and copy it into the left-hand side (often called the *lvalue*)**
- ***rvalue***
  - Any constant, variable, or expression
- ***lvalue***
  - Must be a distinct, named variable
- **A = 4; // (o)**
- **4 = A; // (x)**

Mathematical operators

- **Addition (+), subtraction (-), division (/), multiplication (*), and modulus (%)**
- **Shorthand notation**
  - An operator followed by an equal sign
  - x += 4;

Mathematical operators

```
#include <iostream>
using namespace std;
#define PRINT(STR, VAR) \
  cout << STR " = " << VAR << endl
int main()
{
    int i, j, k;
    float u, v, w;
    cout << "enter an integer: ";
    cin >> j;
    cout << "enter another integer:
      ";
    cin >> k;
    PRINT("j",j); PRINT("k",k);
    i = j + k; PRINT("j + k",i);
    i = j - k; PRINT("j - k",i);
    i = k / j; PRINT("k / j",i);
    i = k * j; PRINT("k * j",i);
    i = k % j; PRINT("k % j",i);
    j %= k; PRINT("j %= k", j);
    cout << "Enter a floating-point
      number: ";
    cin >> v;
    cout << "Enter another floating-
      point number:";
    cin >> w;
    PRINT("v",v); PRINT("w",w);
    u = v + w; PRINT("v + w", u);
    u = v - w; PRINT("v - w", u);
    u = v * w; PRINT("v * w", u);
    u = v / w; PRINT("v / w", u);
    PRINT("u", u); PRINT("v", v);
    u += v; PRINT("u += v", u);
    u -= v; PRINT("u -= v", u);
    u *= v; PRINT("u *= v", u);
    u /= v; PRINT("u /= v", u);
}
```

Introduction to preprocessor macros

- Use of the macro `PRINT()` to save typing
- Preprocessor macros are traditionally named with all uppercase letters
- The preprocessor removes the name `PRINT` and substitutes the code wherever the macro is called

Relational operators

- **Establish a relationship between the values of the operands**
- **Produce a Boolean (true or false)**
- **Operators**
  - less than (**<**)
  - greater than (**>**)
  - less than or equal to (**<=**)
  - greater than or equal to (**>=**)
  - equivalent (**==**)and not equivalent (**!=**)

Logical operators

- *and* (&&) and *or* (||) produce a `true` or `false` based on the logical relationship of its arguments

Logical Operator

```
#include <iostream>
using namespace std;
int main()
{
    int i,j;
    cout << "Enter an integer: ";
    cin >> i;
    cout << "Enter another integer: ";
    cin >> j;
    cout << "i > j is " << (i > j) << endl;
    cout << "i < j is " << (i < j) << endl;
    cout << "i >= j is " << (i >= j) << endl;
    cout << "i <= j is " << (i <= j) << endl;
    cout << "i == j is " << (i == j) << endl;
    cout << "i != j is " << (i != j) << endl;
    cout << "i && j is " << (i && j) << endl;
    cout << "i || j is " << (i || j) << endl;
    cout << " (i < 10) && (j < 10) is " << ((i < 10) && (j < 10)) <<
      endl;
}
```

Bitwise operators

- **Allow you to manipulate individual bits in a number**
- **Perform Boolean algebra on the corresponding bits in the arguments to produce the result**
- **Operators**
  - The bitwise *and* operator (&)
  - The bitwise or operator (|)
  - The bitwise *exclusive or*, or *xor* (^)
  - The bitwise *not* (~, also called the *ones complement* operator)
  - Bitwise operators can be combined with the = sign to unite the operation and assignment: &=, |=, and ^=

Shift operators

- **Manipulate bits**
- **The left-shift operator (<<)**
  - Produces the operand to the left of the operator shifted to the left by the number of bits specified after the operator
- **The rightshift operator (>>)**
  - Produces the operand to the left of the operator shifted to the right by the number of bits specified after the operator
- **Shifts can be combined with the equal sign (<<= and >>=)**

Shift operators

- If the value after the shift operator is greater than the number of bits in the left-hand operand, the result is undefined
- If the left-hand operand is unsigned, the right shift is a logical shift so the upper bits will be filled with zeros
- If the left-hand operand is signed, the right shift may or may not be a logical shift (that is, the behavior is undefined)

Shift operators (example)

```
//: C03:printBinary.h
void printBinary(const unsigned char val);
#include <iostream>
void printBinary(const unsigned char val)
{
    for(int i = 7; i >= 0; i--)
    if(val & (1 << i))
      std::cout << "1";
    else
      std::cout << "0";
}
```

Shift operators (example)

```
#include "printBinary.h"
#include <iostream>
using namespace std;
// A macro to save typing:
#define PR(STR, EXPR) \
cout << STR; printBinary(EXPR);
  cout << endl;
int main()
{
    unsigned int getval;
    unsigned char a, b;
    cout << "Enter a number
      between 0 and 255: ";
    cin >> getval; a = getval;
    PR("a in binary: ", a);
    cout << "Enter a number
      between 0 and 255: ";
    cin >> getval; b = getval;
    PR("b in binary: ", b);
    PR("a | b = ", a | b);
    PR("a & b = ", a & b);
    PR("a ^ b = ", a ^ b);
    // An interesting bit
      pattern:
    unsigned char c = 0x5A;
    PR("c in binary: ", c);
    a |= c;
    PR("a |= c; a = ", a);
    b &= c;
    PR("b &= c; b = ", b);
    b ^= a;
    PR("b ^= a; b = ", b);
}
```

Shift operator (example)

```
unsigned char rol(unsigned
  char val)
{
    int highbit;
    if(val & 0x80) // 0x80 is
      the high bit only
      highbit = 1;
    else
      highbit = 0;
    // Left shift (bottom bit
      becomes 0):
    val <<= 1;
    // Rotate the high bit onto
      the bottom:
    val |= highbit;
    return val;
}
```

```
unsigned char ror(unsigned char
  val)
{
    int lowbit;
    if(val & 1) // Check the low
      bit
      lowbit = 1;
    else
      lowbit = 0;
    val >>= 1; // Right shift by
      one position
    // Rotate the low bit onto the
      top:
    val |= (lowbit << 7);
    return val;
}
```