3. The C in C++

Hu Sikang

skhu@163.com

School of Computer
Beijing Institute of Technology

Content

- **Operators**
- **Build_in type**
- **Variables**
- **Scoping**
- **Array**
- **Const**
- **Cast**
- **Function and function pointer**

3.1 Creating functions

3.2 Controlling execution

- if-else
- while
- do-while
- for -- *for*
- break
- continue
- switch-case
- *goto*

3.2 Controlling execution

for (data_type var : collection_to_loop) // the 11th version of C++
{ // Here are codes… }

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

int main() {
    int array[] = {10, 23, 45, 12, 56};

    for (int a : array)   // a is read-only
        cout << a << endl;
    return 0;
}
```

3.2 Controlling execution

```
for  (data_type  var  :  collection_to_loop)
{   // Here are codes…    }
```

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

int main() {
    vector<int> v;

    for (int i = 0; i < 5; i++)
        v.push_back(i * 3);

    for (int x : v)
        cout << x << endl;
    return 0;
}
```

3.3 Operators

➢ Unary Operators

new, delete, new[ ], delete[ ],

++, --, (), [ ], +, -, *, &, !, ~,

➢ Binary operators

+, -, *, /, %, =, +=, -=, *=, /=, %=,

&, |, ^, ^=, &=, |=, ==, !=, >, <, >=,

<=, ||, &&, <<, >>, >>=, <<=, ->, ->*

➢ Other operators

.   member selection

.* member selection by a pointer

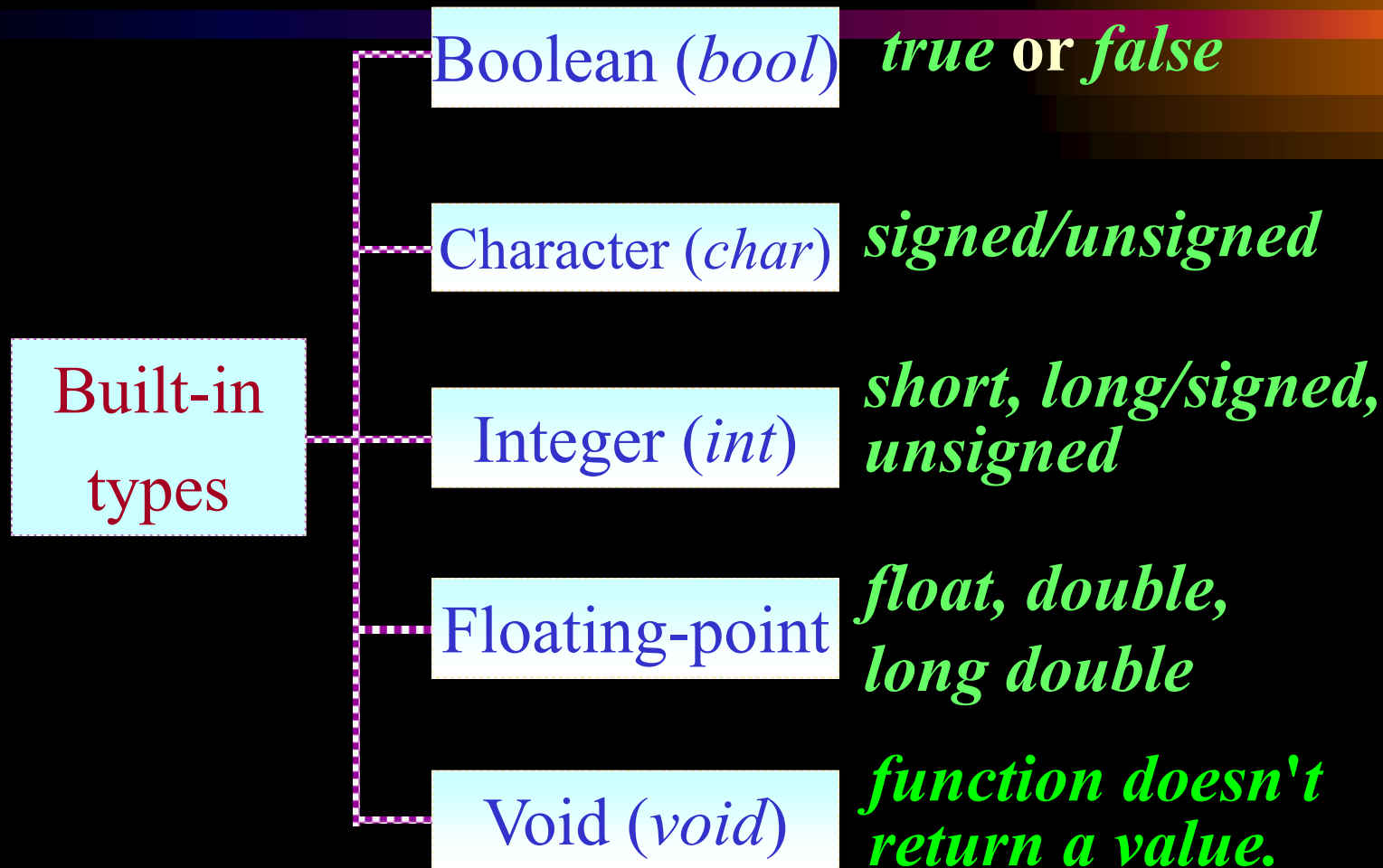::   scope resolution

?: ternary conditional expression

sizeof

3.4 Introduction to data types

3.4.1 Built-in types

3.4.2 Pointers

3.4.3 references

3.4.1 Built-in types

Built-in types

- Boolean (*bool*) — *true* **or** *false*
- Character (*char*) — *signed/unsigned*
- Integer (*int*) — *short, long/signed, unsigned*
- Floating-point — *float, double, long double*
- Void (*void*) — *function doesn't return a value.*

3.4.2 Introduction to pointers

- Every element of your program occupies storage and has an address.

- C++ have a special type of variable that holds an address. This variable is called a *pointer*.

3.4.3 Argument Passing

- Pass by value

- Pass by address (pointer)

- Pass by reference

(1) Pass by value

```cpp
#include <iostream>
using namespace std;
void f(int a)
{
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}
```

```cpp
int main()
{
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
  return 0;
}
```

(2) Pass by address

```cpp
#include <iostream>
using namespace std;
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;
  return 0;
}
```
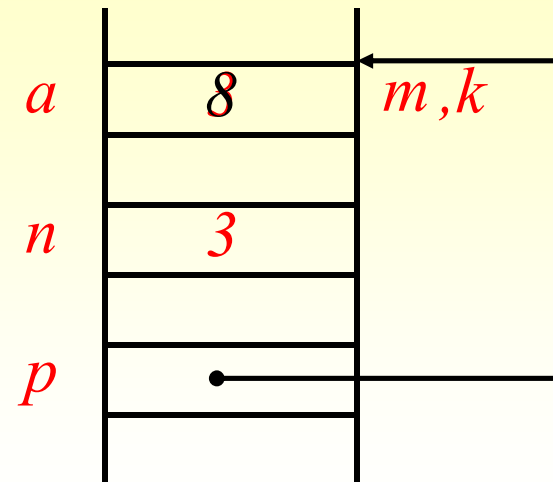
(3) references

- ***Pass-by-reference*** *is* an additional way to pass an **address** into a function.

- Pass-by-reference allows a function to modify the outside object, just like passing a pointer does.

- *Calling* a function that takes references is cleaner, syntactically, than calling a function that takes pointers.

- We must *initialize* the reference except as a parameters of the function.

```
//: C01:PassByValue.cpp
#include <iostream>
using namespace std;
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;
  return 0;
} ///:~
```

```
//: C02:PassReference.cpp
#include <iostream>
using namespace std;
int 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;
  return 0;
} ///:~
```

```cpp
//: C01:PassByValue.cpp
#include <iostream>
using namespace std;
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;
  return 0;
} ///:~
```

```cpp
//: C02:PassReference.cpp
#include <iostream>
using namespace std;
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;
  return 0;
} ///:~
```

16

3.5 Scoping

- The scope of a variable is defined by its "nearest" set of brace.
- A variable can be used only when inside its scope.

```cpp
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
        // scp3 not available here
        // scp1 & scp2 still visible here
    } // <-- scp2 destroyed here
    // scp3 & scp2 not available here
    // scp1 still visible here
    return 0;
} // <-- scp1 destroyed here
```

3.6 Specifying storage allocation

3.6.1 Global variables

3.6.2 Local variables

3.6.3 **Static** variables

3.6.4 Extern

3.6.5 Constants

3.6.5 Constants

- #define PI 3.14159 *// replacement.* **Its scope is:**
  from #define to #undef

- const **double** PI=3.14159 ; **// a variable**

- A **const** is just like a variable, except that its value cannot be changed.

3.6.5 Constants

A const must always have an initialization value except as a parameters of the function.

```
int f(int& x)
{
        return ++x;   //OK
}
int g(const int& x)
{
        return ++x;   //ERROR
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int a = 9;
    f(a);
    g(a);
    return 0;
}
```

3.7 Operators and their use

3.7.1 Assignment

$$A = 4$$

- **A:** an lvalue, a distinct, named **variable**
- **4:** an rvalue, a constant, variable, or expression that can produce a **value**

3.7.2 Mathematical operators

- addition (**+**)

- subtraction (**-**)

- division (**/**)

- multiplication (*****)

- modulus (**%**)**;** this produces the remainder from **integer** division.

- **x = x + 4;   x += 4;**

3.7.3 Relational operators

- less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), equivalent (==), and not equivalent (**!=**).

- They produce a Boolean **true** if the relationship is true, and **false** if the relationship is false.

- If you print a **bool**, you'll typically see a '**1**' for **true** and '**0**' for **false**.

3.7.4 Logical operators

- *and* (**&&**) ; *or* (**||**) ; *not* (**!**)
- The result is **true** if it has a non-zero value, and **false** if it has a value of zero.

```
int i = 10, j = 5;
cout << ((i == 10) && (j > 10));
cout << ((i < 10) || (j < 10));
```

3.7.5 Bitwise operators

- bitwise *and* (**&**)
- bitwise or (|)
- bitwise *not / complement* (~)
- bitwise *exclusive or / xor* (^)
- **&**=; |=; ^=

3.7.6 Shift operators

- left-shift (<<)
- right-shift (>>)
- <<= ; >>=

1, 16, 1, 8

```cpp
#include <iostream>
using namespace std;
int main()
{
    unsigned int a=1;
    cout<<a<<',';
    unsigned int b=a<<4;
    cout<<b<<','<<a<<',' ;
    a=b>>1;
    cout<<a<<endl;
    return 0;
} ///:~
```

3.7.7 Unary operators

- Bitwise *not* (**~**)
- *logical not* (**!**)
- unary minus (**-**) ; unary plus (**+**)
- increment (**++**); decrement (**--**)
- address-of (**&**)
- dereference (***** and **->**)
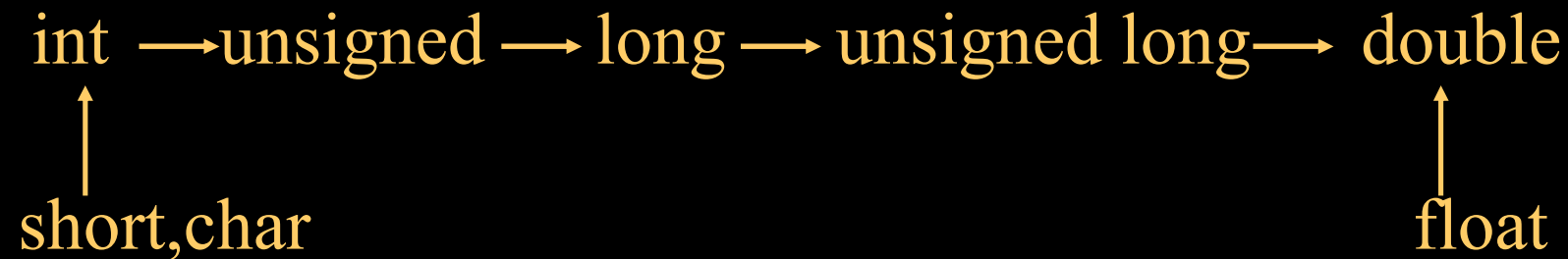- **cast**
- **new** ; **delete**

3.7.8 The ternary operator

- $c = a > b \ ? \ a : b$
- if $(a > b)$ $c = a;$

  else $c = b;$

3.7.9 The comma operator

3.7.11 Casting operators

- The compiler will automatically change one type of data into another if it makes sense.

int ⟶ unsigned ⟶ long ⟶ unsigned long ⟶ double

short,char ↑int

float ↑double

3.7.12 sizeof– an operator

```cpp
int main()
{
    cout<<"bool:"<<sizeof(bool)<<endl;
    cout<<"char:"<<sizeof(char)<<endl;
    cout<< "int:" <<sizeof(int)<<endl;
    cout<<"float:"<<sizeof(float)<<endl;
    cout<<"double:"<<sizeof(double)<<endl;
    cout<<"long double:"<<sizeof(long double)<<endl;
    return 0;
}
```

bool:1

char:1

int:4

float:4

double:8

long double:8

The results may be *vary* with different machines/operating systems/compilers.

3.8 Composite type creation

- C++ provide tools that allow you to compose more sophisticated data types from the fundamental data types.

- The most important of these is **struct**, which is the foundation for **class** in C++.

- the simplest way to create more sophisticated types is simply to alias a name to another name via **typedef**.

3.8.1 Clarifying programs with enum

➢ An *enumeration* is a type that can hold a set of values specified by the user.

  enum keyword *{ASM, AUTO, BREAK}*;

➢ By *default*, the values of enumerators are initialized increasing from *0*;

  ASM=0, AUTO=1, BREAK=2;

  BREAK==6

➢ An enumerator can be initialized by a *constant-expression* of integer type.

  enum keyword *{ASM=2, AUTO=5, BREAK}*;

33

3.9 Pointers and arrays

- The name of an array can be used as a *pointer to its initial elements*.

- Access array can be achieved either through a pointer to an array plus an index or through a pointer to an element.

- Notion: Most C++ implementations offer *no range checking* for arrays.

3.10 Function addresses

- Once a function is compiled and loaded into the computer to be executed, it occupies a chunk of memory, and has an address.

- You can use function addresses with pointers just as you can use variable addresses.

3.10.1 Defining a function pointer

$$void\ (*funcPtr)(\ );$$

- **funcPtr** is a pointer to a function that has no arguments and no return value.

- **void* funcPtr ( );**

- **funcPtr** is a function that returns a void* .

```cpp
// Defining and using a pointer to a function
#include <iostream>
using namespace std;

void func()
{
  cout << "func() called..." << endl;
}
int main()
{
    void (*fp)( );                 // Declare a function pointer
    fp = func;                     // Initialize it
    (*fp)( );                      // Call the function
    void  (*fp2)( ) = func;        // Define and initialize
    (*fp2)( );                     // Call the function
   return 0;
}
```

3.10.2 Call Function with Function Pointer

```cpp
#include <iostream>
#include <tchar.h>
#include <windows.h>
using namespace std;
typedef  int (*CallFunction) (int a, int b);
int main(void) {
    HINSTANCE hDLL;
    CallFunction JIA;
    hDLL = LoadLibrary(_T("MyDll.dll"));  // _T: ANSI or Unicode
    if (hDLL == nullptr)
    {
        cout << "NULL" << endl;
        return 0;
    }
    // loading Dynamatic Link Libaray
    JIA = (CallFunction)GetProcAddress(hDLL, "Add");
    cout << (*JIA)(10, 20) << endl;
    FreeLibrary(hDLL);                          // unload DLL file
    return 0;
}
```

Summary

- **Operators**
- **Build_in type**
- **Variables**
- **Scoping**
- **Array**
- **Const**
- **cast**
- **Function and function pointer**