Advanced C

Variables and Functions in C++

Th.S Nguyen Minh Anh

Phenikaa University

Last Update: 30th January 2023

Outline

Types

Type of a variable determines the meaning of the data and what operations we can do on it.

Example: String input

```
int main()
{
  cout << " Please enter your first and second names: " ;
  string first ;
  string second ;
  // read two strings
  cin >> first >> second ;
  // concatenate strings, separated by a space
  //
  string name = first + ' ' + second ;
  cout << " Hello , " << name << "\n";
  return 0;
}
```

Example: Integer input

```
int main()
{
  cout << " Please enter your two integer numbers a and b: " ;
  int a;
  int b;
  // read two strings
  cin >> a >> b;
  // calculate sum of two numbers
  int sum = a + b;
  cout << " Sum = " << sum << "\n";
  return 0;
}
```

Integers and Strings

Strings

- `cin` >> reads a word
- `cout` << writes
- + concatenates
- += s adds the string s at end
- ++ is an error
- - is an error
- ...

Integers and floating-point numbers

- `cin` >> reads a number
- `cout` << writes
- + adds
- `+=n` increments by the int `n`
- ++ increments by 1
- - subtracts
- ...

The type of a variable determines which operations are valid and what their meanings are for that type (that's called overloading or operator overloading)

Primitive Built-in Types

C++ defines a set of primitive types that include the **arithmetic types** and a special type named **void**.

- The arithmetic types represent characters, integers, boolean values, and floating-point numbers.
- The `void` type has no associated values and can be used in only a few circumstances, most commonly as the return type for functions that do not return a value.

Type conversion

C++ allows us to convert data of one type to that of another. This is known as type conversion.
There are two types of type conversion in C++.

- Implicit Conversion
- Explicit Conversion (also known as Type Casting)

Implicit Type Conversion

The type conversion that is done automatically done by the compiler is known as implicit type conversion. This type of conversion is also known as automatic conversion.

Implicit Type Conversion

Example 1: Conversion from int to double

```
#include <iostream>
using namespace std;
int main() {
   int num_int = 9; // assigning an int value to num_int
   double num_double; // declaring a double type variable
   // implicit conversion
   // assigning int value to a double variable
   num_double = num_int;

   cout << "num_int = " << num_int << endl;
   cout << "num_double = " << num_double << endl;
   return 0;
}
```

Output:

```
num_int = 9
num_double = 9
```

Implicit Type Conversion

Example 2: Conversion from `double` to `int`

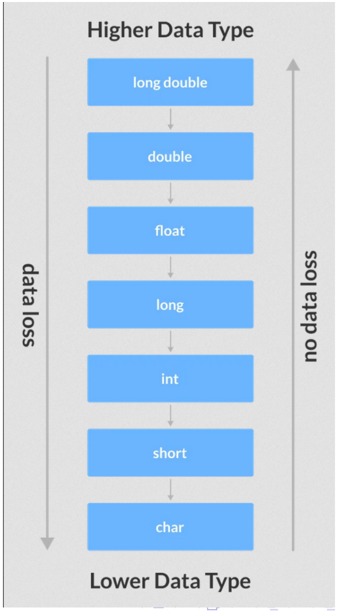
```
#include <iostream>
using namespace std;
int main() {
  int num_int;
  double num_double = 9.99;
  // implicit conversion
  // assigning a double value to an int variable
  num_int = num_double;
  cout << "num_int = " << num_int << endl;
  cout << "num_double = " << num_double << endl;

  return 0;
}
```

Output?

Implicit Type Conversion

Data Loss During Conversion (Narrowing Conversion) As we have seen from the above example, conversion from one data type to another is prone to data loss. This happens when data of a larger type is converted to data of a smaller type.



Explicit Type Conversion

When the user manually changes data from one type to another, this is known as explicit conversion. This type of conversion is also known as type casting.
There are three major ways in which we can use explicit conversion in C++. They are:

- C-style type casting (also known as **cast notation**)
- Function notation (also known as **old C++ style type casting**)
- Type conversion operators, by `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`

Explicit Type Conversion: Example

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

int main() {
   double num_double = 3.56;
   cout << "num_double = " << num_double << endl;
   // C-style conversion from double to int
   int num_int1 = (int)num_double;
   cout << "num_int1 = " << num_int1 << endl;
   // function-style conversion from double to int
   int num_int2 = int(num_double);
   cout << "num_int2 = " << num_int2 << endl;
   int num_int3 = static_cast<int>(num_double);
   cout << "num_int3 = " << num_int3 << endl;

   return 0;
}
```

Outline

Variable Definitions

A variable is a container (storage area) to hold data.

- Each variable should be given a unique name (**identifier**)
- The **type** determines the size and layout of the variable's memory, the range of values that can be stored within that memory, and the set of operations that can be applied to the variable
- A variable that is **initialized** gets the specified value at the moment it is created.

```
int sum = 0, value, // sum, value, and units_sold have type int
units_sold = 0; // sum and units_sold have initial value 0
Sales_item item; // item has type Sales_item
// string is a library type, representing a variable-length sequence of
    characters
std::string book("0-201-78345-X"); // book initialized from string
    literal
```

Identifiers

Rules for naming a variable

- A variable name can only have alphabets, numbers, and the underscore _
- A variable name cannot begin with a number
- It is a preferred practice to begin variable names with a **lowercase** character. For example, `name` is preferable to `Name`
- A variable name cannot be a **keyword**. For example, `int`, `float`, `char`, `return`, etc
- An identifier should give some indication of its meaning.

Advice: Define variables where you first use them!

It is usually a good idea to define an object near the point at which the object is first used.

- improves readability by making it easy to find the definition of the variable
- it is often easier to give the variable a useful initial value when the variable is defined close to where it is first used

Warning

It is almost always a bad idea to define a local variable with the same name as a global variable that the function uses or might use.

Compound types

A `compound type` is a type that is defined in terms of another type. C++ has several compound types, two of which we'll cover:

- references
- pointers

Compound types: References

A **reference** defines an alternative name for an object

- When we define a reference, instead of copying the initializer's value, we **bind** the reference to its initializer
- A reference is not an object. Instead, a reference is *just another name for an already existing object*
- After a reference has been defined, all operations on that reference are actually operations on the object to which the reference is bound
- Because references are not objects, we may not define a reference to a reference

```
int ival = 1024;
int &refVal = ival; //refVal refers to (is another name for) ival
int &refVal2; // error: a reference must be initialized
refVal = 2; // assigns 2 to the object to which refVal refers, i.e., to
    ival
int ii = refVal; // same as ii = ival
```

Compound types: Pointers

A **pointer** is a compound type that ”points to” another type. Like references, pointers are used for indirect access to other objects. Unlike a reference, **a pointer is an object in its own right**.

- Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime.
- Unlike a reference, a pointer need not be initialized at the time it is defined.
- Like other built-in types, pointers defined at block scope have undefined value if they are not initialized.

```
double dval;
double *pd = &dval; // ok: initializer is the address of a double
double *pd2 = pd; // ok: initializer is a pointer to double
int *pi = pd; // error: types of pi and pd differ
pi = &dval; // error: assigning the address of a double to a pointer
    to int
```

const

Sometimes we want to define a variable whose value we know cannot be changed. We can make a variable unchangeable by defining the variable's type as `const`:

```
    const int bufSize = 512; // input buffer size
    bufSize = 512; // error: attempt to write to const object
```

Because we can't change the value of a const object after we create it, it **must be initialized**.

```
    const int i = get_size(); //ok: initialized at run time
    const int j = 42; // ok: initialized at compile time
    const int k; // error: k is uninitialized const
```

C++14 Hint

You can use the type of an initializer as the type of a variable

```
// 'auto' means 'the type of the initializer'
auto x = 1; // 1 is an int , so x is an int
auto y = 'c'; // 'c' is a char , so y is a char
auto d = 1.2; // 1.2 is a double , so d is a double
auto s = "Howdy"; // "Howdy" is a string literal of type const char []
                // so don't do that until you know what it means !
auto sq = sqrt (2); // sq is the right type for the result of sqrt (2)
auto duh; // and you don't have to remember what that is
         // error : no initializer for auto
```

Outline

Recap: Why Functions?

- Chop a program into manageable pieces
  ”divide and conquer”
- Match our understanding of the problem domain
  - Name logical operations
  - A function should do one thing well
- Functions make the program easier to read
- A function can be useful in many places in a program
- Ease testing, distribution of labor, and maintenance
- Keep functions small
  Easier to understand, specify, and debug

Function basics

- General form: a return type, a name, a list of zero or more parameters, and a body.

```
return_type name (parameters); // a declaration
return_type name (parameters){ // a definition
  body
}
```

  For example:

```
double f (int a , double d) { return a*d; }
```

- If you don't want to return a value give void as the return type

```
void sayHello(string name){ cout << "Hello " << name; }
```

Function basics: Example

Writing a function

```
int fact(int val)
{
  int ret = 1;
  while (val > 1)
    ret *= val--;
  return ret;
}
```

Calling a function

```
int main()
{
  int j = fact(5);
  cout << "5! is " << j << endl;
  return 0;
}
```

Calling `"int j = fact(5)"` is equivalent to the following:

```
int val = 5;   // initialize val from the literal 5
int ret = 1;   // code from the body of fact
while (val > 1)
  ret *= val--;
int j = ret;  // initialize j as a copy of ret
```

Forward Declaration

Forward Declaration refers to the beforehand declaration of the syntax or signature of an identifier, variable, function, class, etc. prior to its usage (done later in the program).
Use case example:

```
int main(){
  std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
  return 0;
}

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

How do we resolve the error? Add the following line in front (before the usage)

```
int add(int x, int y); // function declaration includes return type, name,
    parameters, and semicolon. No function body!
```

Parameters and Arguments

- The type of each argument must match (or converted to) the corresponding parameter.
- We must pass exactly the same number of arguments as the function has parameters.
- Parameter list
    - may be empty, but not omitted
    - must specify each param's type

```
fact("hello"); // error: wrong argument type
fact(); // error: too few arguments
fact(42, 10, 0); // error: too many arguments
fact(3.14); // ok: argument is converted to int
```

Argument Passing

The type of a parameter determines the interaction between the parameter and its argument. If the parameter is a reference, then the parameter is bound to its argument. Otherwise, the argument's value is copied.

- **passed by reference**: a reference parameter is an alias for the object to which it is bound
- **passed by value**: the argument value is copied, the parameter and argument are independent objects

Argument Passing: Pass-by-Value

Passing an argument by value works exactly the same way we initialize a nonreference type variable

```
    int n = 0; // ordinary variable of type int
    int i = n; // i is a copy of the value in n
    i = 42; // value in i is changed; n is unchanged
```

Nothing the function does to the parameter can affect the argument. For example, inside `fact` function (previous slide):

```
    ret *= val--; // decrements the value of val
```

Although fact changes the value of `val`, that change has no effect on the argument passed to `fact`. Calling `fact(i)` does not change the value of `i`.

Argument Passing: Pass-by-Reference

- To allow a function to change the value of one or more of its arguments. Example:

```
// function taking a reference to an int and sets the given object to
    zero
void reset(int &i) // i is just another name for the object passed to
    reset
{
  i = 0; // changes the value of the object to which i refers
}
```

- When we call this version of reset, we pass an object directly; there is no need to pass its address:

```
  int j = 42;
  reset(j); // j is passed by reference; the value in j is changed
  cout << "j = " << j << endl; // prints j = 0
```

In this call, the parameter `i` is just another name for `j`. Any use of `i` inside `reset` is a use of `j`.

Using References to Avoid Copies

- Inefficient to copy objects of large class types or large containers.
- Moreover, some class types (including the IO types) cannot be copied.
- **Best practice**: Reference parameters that are not changed inside a function should be references to `const`.

```
// compare the length of two strings
bool isShorter(const string &s1, const string &s2)
{
  return s1.size() < s2.size();
}
// Because strings can be long, we'd like to avoid copying them,
    so we'll make our parameters references. Because comparing
    two strings does not involve changing the strings, we'll make
    the parameters references to const
```

Using Reference Parameters to Return Additional Information

Sometimes a function has more than one value to return.
Question: How can we define a function that returns a position and an occurrence count?

- define a new type that contains the position and the count

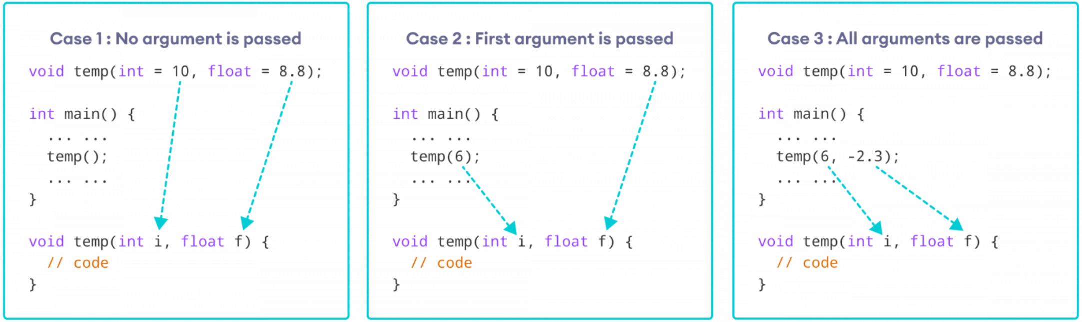
Using Reference Parameters to Return Additional Information

Sometimes a function has more than one value to return.
Question: How can we define a function that returns a position and an occurrence count?

- define a new type that contains the position and the count
- easier solution: to pass an additional reference argument to hold the occurrence count

Default Arguments

- In C++ programming, we can provide default values for function parameters.
- If a function with default arguments is called without passing arguments, then the default parameters are used.
- If arguments are passed while calling the function, the default arguments are ignored.



Case 1 : No argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp();
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 2 : First argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(6);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 3 : All arguments are passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(6, -2.3);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Default Arguments

Remember: Once we provide a default value for a parameter, all subsequent parameters **must also have default values**. For example:

```
// Invalid
void add(int a, int b = 3, int c, int d);

// Invalid
void add(int a, int b = 3, int c, int d = 4);

// Valid
void add(int a, int c, int b = 3, int d = 4);
```

Overloaded Functions

- Functions that have the same name but different parameter lists and that appear in the same scope are **overloaded**.

```
    void print(const char* cp);
    void print(const int x, const int y);
    void print(const char* cp, double b);
```

- When we call these functions, the compiler can deduce which function we want **based on the argument type we pass**:

```
    print("Hello World"); // calls print(const char*)
    print(2, 3); // calls print(const int, const int)
    print("Hello World", 2.5); // calls print(const char*, double)
```

Function overloading eliminates the need to invent—and remember—names that exist only to help the compiler figure out which function to call.

main: Handling Command-Line Options

We sometimes need to pass arguments to main. The most common use of arguments to main is to let the user specify a set of options to guide the operation of the program

```
    prog -d -o ofile data0
```

Such command-line options are passed to main in two (optional) parameters:

```
    int main(int argc, char *argv[]) { ... }
```

Given the previous command line, argc would be 5, and argv would hold the following C-style character strings:

```
    argv[0] = "prog"; // or argv[0] might point to an empty string
    argv[1] = "-d";
    argv[2] = "-o";
    argv[3] = "ofile";
    argv[4] = "data0";
    argv[5] = 0;
    // remember that the optional arguments begin in argv[1]; argv[0]
        contains the program's name, not user input.
```

Handling Command-Line Options: Exercise