【C++】Day18(2)

Class	C++
Date	@December 8, 2021
Material	
Series Number	
Summary	

【Ch6】Functions

6.5 Features for Specialized Uses

6.5.1 Default Arguments

Some functions have parameters that are given a particular value in most, but not all, calls. In such cases, we can declare that common value as a default argument for the function. Functions with default arguments can be called with or without that argument.

For example, we might use a string to represent the contents of a window. By default, we might want the window to have a particular height, width, and background character.

However, we might also want to allow users to pass values other than the defaults. To accommodate both default and specified vaules we would declare our function to define the window as follows:

```
typedef string::size_type sz;
string screen(sz ht = 24, sz wid = 80, char backgrnd = ' ');
```

Here we've provided a default for each parameter. A default argument is specified as an initializer for a parameter in the parameter list.

We may define defaults for one or more parameters. However, if a parameter has a default argument, all the parameters that follow it must also have default arguments.

Calling Functions with Default Arguments

If we want to use the default argument, we omit that argument when we call the function.

Because screen provides defaults for all of its parameters, we can call screen with zero, one, two, or three arguments:

```
string window;
window = screen(); //equivalent to screen(24, 80, ' ')
window = screen(60); //equivalent to screen(60, 80, ' ')
window = screen(66, 256); //screen(66, 256, ' ');
windw = screen(66, 256, '#'); //screen(66, 256, '#')
```

Arguments in the call are resolved by position. The default arguments are used for the trailing(right=most) arguments of a call.

For example, to override the default for background, we must also suppy arguments for height and width:

```
windw = screen(, , '?'); //error: can omit only trailing arguments
window = screen('?'); //calls equivalent to screen('?', 80, ' ')
```

Part of the work of designing a function with default arguments is ordering the parameters so that those least likely to use a default value appear first and those most likely to use a default appear last.

Default Argument Declarations

Although it is normal practie to declare a function once inside a header, it is legal to redeclare a function multiple times.

However, each parameter can have its default specified only once in a given scope. Thus, any subsequent declaration can add a default only for a parameter that has not previously had a default specified.

For example:

```
//no default for the height or width parameters
string screen(sz, sz, char = ' ');
```

We cannot change an already declared default value:

```
string screen(sz, sz, char = '*'); //error: redeclaration
```

But we can add a default argument as follows:

```
string screen(sz = 24, sz = 80, char); //ok: adds default arguments
```

Best Practice: Default arguments ordinarily should be specified with the function declaration in an appropriate header.

Default Argument Initializers

Local variables may not be used as a default argument. Excepting that restriction, a default argument can be any epxression that has a type that is convertible to the type of the parameter:

```
//the declarations of wd, def, and ht must appear outside a function
sz wd = 80;

char def = ' ';
sz ht();
string screen(sz = ht(), sz = wd, char = def);
strign windwo = screen(); //caslls screen(ht(), 80, ' ')
```

Names used as default arguments are resolved in the scope of the function declaration. The value that those names represent is evaluated at the time of the call:

```
void f2() {
  def = '*'; //changes the value of a default argument
  sz wd = 100; //hodes the outer definition of wd but does not change the default
  window = screen(); //calls screen(ht(), 80, '*')
}
```

Inside f2, we changed the value of def. The call to screen passes this updated value. Our function also declared a local variable that hides the outer wd. However, the local named wd is unrelated to the default argument passed to screen.

6.5.2 Inline and constexpr Functions

inline Functions Avoid Function Call Overhead

A function specified as `inline` (usually) is expanded "in line" at each call. If shorterString were defined as inline, then this call

```
cout << shorterString(s1, s2) << endl;
```

would be expanded during compilation into something like:

```
cout << (s1.size() < s2.size() ? s1 : s2) << endl;
```

The run-time overhead of making shorterString a function is thus removed.

We can define shorterString as an inline function by putting the keyword `inline` before the function's return type:

```
//inline version: find the shorter of two strings
inline const string &shorterString(const string &s1, const string &s2) {
  return s1.size() < s2.size() ? s1 : s2;
}
```

Note: The inline specification is only a request to the compiler. The compiler may choose to ignore this request.

In general, the inline mechanism is meant to optimize small, straight-line functions that are called frequently. Many compilers will not inline a recursive function. A 75-line function will almost surely not be expanded inline.

constexpr Functions

A constexpr function is a function that can be used in a constant expression. A constexpr function is defined like any other function but must meet certain restrictions: The return type and the type of each parameter in it must be a literal type, and the function body must contain exactly one return statement:

```
constexpr int new_sz() {
  return 42;
}
constexpr int foo = new_sz(); //ok: foo is a constant expression
```

Here we defined `new_sz` as a constexpr that takes no arguments. The compiler can verify-at compile time-that a call to new_sz returns a contant expression, so we can use `new_sz` to initialize our constexpr variable, `foo`.

When if can do so, the compiler will replace a call to a constexpr function with its resulting value. In order to be able to expand the function immediately, constexpr functions are implicitly inline.

A constexpr function body may contain other statements so long as those statements generate no actions at run time. *For example, a constexpr function may contain null statements, type aliases, and using declarations.*

A constexpr function is permitted to return a value that is not a constant:

```
//scale(arg) is a constant expression if arg is a contant expression
constexpr size_t scale(size_t cnt) {
  return new_sz() * cnt;
}
```

The scale function will return a constant expression if its argument is a constant expression but not otherwise:

```
int arr[scale(2)]; //ok: scale(2) is a constant expression
int i = 2; //i is not a constant expression
int a2[scale(i)]; //error: scale(i) is not a constant expression
```

Put inline and constexpr Functions in Header Files

Unlike otehr functions, inline and constexpr functions may be defined multiple times in the program. After all, the compiler needs the definition, not just the declaration, in order to expand the code. However, all of the definitions of a given incline or constexpr must match exactly. As a result, inline and constexpr functions normally are defined in headers.