The Complete Program

ch04/sentinel.cpp

```
/**
   Turns a number between 10 and 19 into its English name.
   @param number = an integer between 10 and 19
   @return the name of the given number ("ten" ... "nineteen")
*/
string teens_name(int number)
{
   if (number == 10) return "ten";
   if (number == 11) return "eleven";
   if (number == 12) return "twelve";
   if (number == 13) return "thirteen";
   if (number == 14) return "fourteen";
   if (number == 15) return "fifteen";
   if (number == 16) return "sixteen";
   if (number == 17) return "seventeen";
   if (number == 18) return "eighteen";
   if (number == 19) return "nineteen";
   return "";
}
```


The Complete Program

ch04/sentinel.cpp

```
/**
   Gives the name of the tens part of a number between 20 and 99.
   @param number = an integer between 20 and 99
   @return the name of the tens part of the number ("twenty" ...
   "ninety")
*/
string tens_name(int number)
{
   if (number >= 90) return "ninety";
   if (number >= 80) return "eighty";
   if (number >= 70) return "seventy";
   if (number >= 60) return "sixty";
   if (number >= 50) return "fifty";
   if (number >= 40) return "forty";
   if (number >= 30) return "thirty";
   if (number >= 20) return "twenty";
   return "";
}
```


The Complete Program

ch04/sentinel.cpp

```
/**
   Turns a number into its English name.
   @param number = a positive integer < 1,000
   @return the name of the number (e.g. "two hundred seventy four")
*/
string int_name(int number)
{
   int part = number; // The part that still needs to be converted
   string name; // The return value

   if (part >= 100)
   {
      name = digit_name(part / 100) + " hundred";
      part = part % 100;
   }
   if (part >= 20)
   {
      name = name + " " + tens_name(part);
      part = part % 10;
   }
```


The Complete Program

ch04/sentinel.cpp

```
   else if (part >= 10)
   {
      name = name + " " + teens_name(part);
      part = 0;
   }

   if (part > 0)
   {
      name = name + " " + digit_name(part);
   }

   return name;
}

int main()
{
   cout << "Please enter a positive integer: ";
   int input;
   cin >> input;
   cout << int_name(input) << endl;
   return 0;
}
```


EX:

379

int_name(379)

part = 379

379 ≥ 100 → name = digit_name(3) + "hundred"

"three"

integer division

379/100

name = "three hundred"

part = remainder of 379/100 = 79

is part ≥ 20? yes

name = "three hundred" + " " + tens_name(79)

"seventy"

part = rem. of 79/10 = 9

if part > 0 → yes!

name = "three hundred seventy" + digit_name(9)

"nine"

Good Design – Keep Functions Short

- There is a certain cost for writing a function:
  - You need to design, code, and test the function.
  - The function needs to be documented.
  - You need to spend some effort to make the function *reusable* rather than tied to a specific context.

Tracing Functions

When you design a complex set of functions, it is a good idea to carry out a manual walkthrough before entrusting your program to the computer.

This process is called *tracing* your code.

You should trace each of your functions separately.

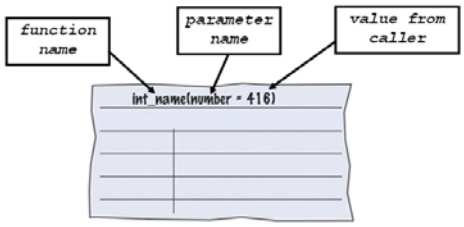
Tracing Functions

To demonstrate, we will trace the `int_name` function when 416 is passed in.

Tracing Functions

Here is the call: `... int_name(416) ...`

Take an index card (or use the back of an envelope) and write the name of the function and the names and values of the parameter variables, like this:

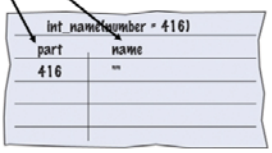


Tracing Functions

Then write the names and values of the function variables.

```
string int_name(int number)
{
   int part = number; // The part that still needs
                      // to be converted
   string name; // The return value, initially ""
```

Write them in a table, since you will update them as you walk through the code:



Tracing Functions

The test (`part >= 100`) is `true` so the code is executed.

```
if (part >= 100)
   {
      name = digit_name(part / 100) + " hundred";
      part = part % 100;
   }
```

Tracing Functions

part / 100 is 4

```
if (part >= 100)
   {
      name = digit_name(part / 100) + " hundred";
      part = part % 100;
   }
```

so digit_name(4) is easily seen to be "four".

Tracing Functions

```
if (part >= 100)
   {
      name = digit_name(part / 100) + " hundred";
      part = part % 100;
   }
```

part % 100 is 16.

Tracing Functions

name has changed to
`name + " " + digit_name(part / 100) + "hundred"`
which is the string "four hundred",

part has changed to `part % 100`, or 16.

int_name(number = 416)

part	name
416	""

Tracing Functions

name has changed to
`name + " " + digit_name(part / 100) + "hundred"`
which is the string "four hundred",

part has changed to `part % 100`, or 16.

Cross out the old values and write the new ones.

int_name(number = 416)

part	name
~~416~~	~~""~~
16	"four hundred"

Tracing Functions

If digit_name's parameter had been complicated, you would have started *another* sheet of paper to trace that function call.

Your work table will probably be covered with sheets of paper (or envelopes) by the time you are done tracing!

Tracing Functions

Let's continue...
Here is the status of the parameters and variables now:

int_name(number = 416)

part	name
~~416~~	~~""~~
16	"four hundred"

Tracing Functions

The test `(part >= 20)` is `false` but
the test `(part >= 10)` is `true` so that code is executed.

```
if (part >= 20)...
else if (part >= 10) {
   name = name + " " + teens_name(part);
   part = 0;
}
```

`teens_name(16)` is "sixteen", `part` is set to 0, so do this:

int_name(number = 416)	
part	name
~~416~~	~~""~~
~~16~~	~~"four hundred"~~
0	"four hundred sixteen"

Tracing Functions

Why is `part` set to 0?

```
if (part >= 20)...
else if (part >= 10) {
   name = name + " " + teens_name(part);
   part = 0;
}

if (part > 0)
{
   name = name + " " + digit_name(part);
}
```

After the `if-else` statement ends, `name` is complete.

The test in the following `if` statement needs to be "fixed" so that part of the code will not be executed

- nothing should be added to **`name`**.

Stubs

- When writing a larger program, it is not always feasible to implement and test all functions at once.
- You often need to test a function that calls another, but the other function hasn't yet been implemented.

Stubs

- You can temporarily replace the body of function yet to be implemented with a *stub*.
- A stub is a function that returns a simple value that is sufficient for testing another function.
- It might also have something written on the screen to help you see the order of execution.
- Or do both of these things

Stubs

Here are examples of stub functions.

```
/**
Turns a digit into its English name.
@param digit = an integer between 1 and 9
@return the name of digit ("one" ... "nine")
*/
string digit_name(int digit)
{
    return "mumble";
}
/**
Gives the name of the tens part of a number between 20 and 99.
@param number = an integer between 20 and 99
@return the tens name of the number ("twenty" ... "ninety")
*/
string tens_name(int number)
{
    return "mumblety";
}
```


Stubs

If you combine these stubs with the completely written `int_name` function and run the program testing with the value 274, this will the result:

```
Please enter a positive integer: 274
mumble hundred mumblety mumble
```

which *eveyone* knows indicates that the basic logic of the `int_name` function is working correctly.

(OK, only you know, but that is the important thing with stubs)

Now that you have tested `int_name`, you would "unstubify" another stub function, then another...

Variable Scope (5.7)

?

Variable Scope

You can only have *one* **`main`** function
but you can have as many variables and parameters
spread amongst as many functions as you need.

Can you use the same name in different functions?

Variable Scope

A variable or parameter that is defined within a function
is visible from the point at which it is defined until
the end of the block named by the function.

This area is called the *scope* of the variable.

Variable Scope

The scope of a variable is the part of the
program in which it is *visible*.

Variable Scope

The scope of a variable is the part of the
program in which it is *visible*.

Because scopes do not overlap,
a name in one scope cannot
conflict with any name in another scope.

Variable Scope

The scope of a variable is the part of the
program in which it is *visible*.

Because scopes do not overlap,
a name in one scope cannot
conflict with any name in another scope.

A name in one scope is "invisible"
in another scope

Variable Scope

```
double cube_volume(double side_len)
{
  double volume = side_len * side_len * side_len;
  return volume;
}
int main()
{
  double volume = cube_volume(2);
  cout << volume << endl;
  return 0;
}
```

Each `volume` variable is defined in a separate function, so there is not a problem with this code.

Variable Scope

Because of scope, when you are writing a function you can focus on choosing variable and parameter names that make sense for your function.

You do not have to worry that your names will be used elsewhere.

Variable Scope

Names inside a block are called *local* to that block.

A function names a block.

Recall that variables and parameters do not exist after the function is over—because they are local to that block.

But there are other blocks.

Variable Scope

It is *not legal* to define two variables or parameters with the same name in the same scope.

For example, the following is not legal:

```
int test(double volume)
{
   double volume = cube_volume(2);
   double volume = cube_volume(10);
// ERROR: cannot define another volume variable
// ERROR: or parameter in the same scope
...
}
```


Variable Scope – Nested Blocks

However, you can define another variable with the same name in a *nested block*.

```
double withdraw(double balance, double amount)
{
   if (...)
   {
      double amount = 10;
      ...
   }
   ...
}
```

a variable named `amount` local to the `if`'s block
– *and* a parameter variable named `amount`.

Variable Scope – Nested Blocks

The scope of the parameter variable `amount` is the entire function, *except* the nested block.

Inside the nested block, `amount` refers to the local variable that was defined in that block.

You should avoid this *potentially confusing situation* in the functions that you write, simply by renaming one of the variables.

Why should there be a variable with the same name in the same function?

Global Variables

- Generally, global variables are ***not*** a good idea.

But ...

here's what they are and how to use them

(if you must).

Global Variables

Global variables are defined *outside* any block.

They are visible to every function defined after them.

Global Variables

In some cases, this is a good thing:

The `<iostream>` header defines these global variables:

```
cin
cout
```

This is good because there should only be one of each of these and every function who needs them should have direct access to them.

Global Variables

But in a banking program, how many functions should have direct access to a balance variable?

Global Variables

```
int balance = 10000; // A global variable

void withdraw(double amount)
{
   if (balance >= amount)
   {
      balance = balance - amount;
   }
}

int main()
{
   withdraw(1000);
   cout << balance << endl;
   return 0;
}
```


Global Variables

In the previous program there is only one function that updates the `balance` variable.

But there could be many, many, many – written by group of programmers.

Then we would have a problem.

Global Variables

When multiple functions update global variables, the result can be *difficult* to predict.

Particularly in larger programs that are developed by multiple programmers, it is very important that the effect of each function be clear and easy to understand.

Global Variables – Breaking Open the Black Box

Programs with global variables are difficult to maintain and extend because you can no longer view each function as a "black box" that simply receives parameter values and returns a result or does something.

When functions modify global variables, it becomes more difficult to understand the effect of function calls.

Global Variables – Just Say "No"

You should *avoid* global variables in your programs!

Reference Parameters (5.8) Stop

- Suppose you would like a function to get the user's last name and ID number.
- The variables for this data are in your scope.
- But you want the function to change them for you.
- If you want to write a function that changes the value of a parameter, you must use a *reference parameter*.

Reference Parameters

To understand the need for a different kind of parameter, you must first understand why the parameters you now know do not work.

Reference Parameters

Consider a function that simulates withdrawing a given amount of money from a bank account, provided that sufficient funds are available.

If the amount of money is insufficient, a $10 penalty is deducted instead.

The function would be used as follows:

```
double harrys_account = 1000;
withdraw(harrys_account, 100);
     // Now harrys_account is 900
withdraw(harrys_account, 1000);
     // Insufficient funds.
     // Now harrys_account is 890
```