山居秋暝

~ 王維

空山新雨後，
天氣晚來秋。
明月松間照，
清泉石上流。
竹喧歸浣女，
蓮動下漁舟。
隨意春芳歇，
王孫自可留。

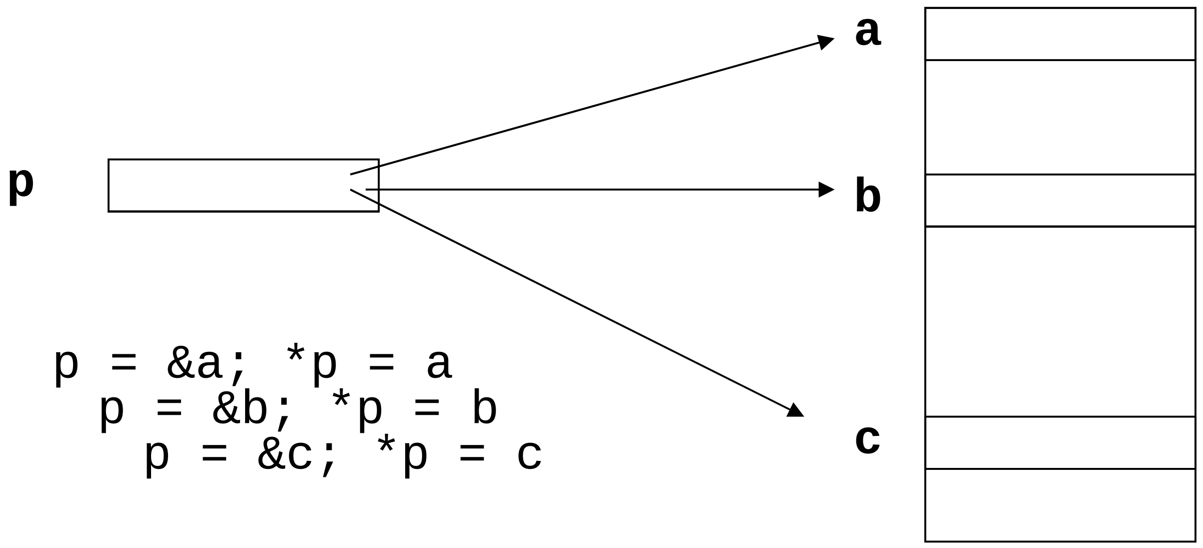
山居秋暝
王維
空山新雨後
天氣晚來秋
明月松間照
清泉石上流
竹喧歸浣女
蓮動下漁舟
隨意春芳歇
王孫自可留

Chapter 6

More about Functions

Pointers (Chapter 4)

- A pointer stores an address
  - which point to a variable of some type
- A single pointer can point to different variables at different times



```
p = &a; *p = a
  p = &b; *p = b
    p = &c; *p = c
```

Pointers to Functions

- A pointer to functions also provide you the flexibility.
  - It will call the function whose address was last assigned to the pointer.
- A pointer to a function must contain
  - The memory address of the function
  - The parameter list
  - The return type

Declaring Pointers to Functions

- `double (*pfun) (char*, int);`
  - The parentheses around the pointer name, `pfun`, and the asterisk are necessary.
  - Otherwise, `double *pfun (char*, int)` would be a function returning a pointer to a double value.

- `long sum(long num1, long num2);`
- `long (*pfun)(long, long) = sum;`

- `long product(long, long);`
- `pfun = product;`

Ex6_01.cpp on P.297

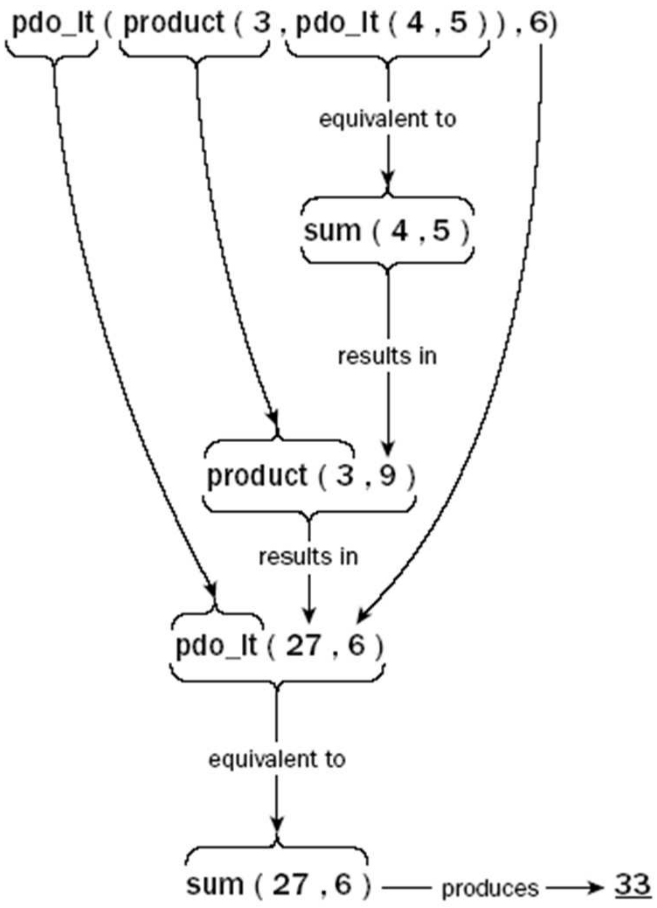


Figure 6-1

A Simpler Example

- As a matter of fact, I think Ex6_01.cpp is too complicated. I prefer the following example:
  - `pdo_it = product;`
  - `cout << pdo_it(3,5) << endl;`
  - `pdo_it = sum;`
  - `cout << pdo_it(3,5) << endl;`

A Pointer to a Function as an Argument

- Ex6_02.cpp on P.300

```
int main(void)
{
  double array[] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };
  int len(sizeof array/sizeof array[0]);

  cout << endl << "Sum of squares = " << sumarray(array,
   len, squared);
  cout << endl << "Sum of cubes = " << sumarray(array, len,
   cubed);
  cout << endl;
  return 0;
}
```

sumarray()

```
double sumarry(double array[], int
  len, double (*pfun) (double))
{
    double total(0.0);

    for (int i=0; i<len; i++)
        total += pfun(array[i]);

    return total;
}
```

Arrays of Pointers to Functions

- double sum(double, double);
- double product(double, double);
- double difference(double, double);
- double (*pfun[3]) (double, double) = { sum, product, difference } ;
  - pfun[1](2.5, 3.5);
    - product(2.5, 3.5)
  - (*pfun)(2.5, 3.5);
    - sum(2.5, 3.5);
  - (*(pfun+2)) (2.5, 3.5);
    - difference(2.5, 3.5)

Initializing Function Parameters

- You may declare the default value of some parameters:
  - `void showit(char msg[] = "I know the default!");`
- When you omit the argument in calling the function, the default value will be supplied automatically.
  - `showit("Today is Wednesday.");`
  - `showit();`
- Ex6_03.cpp on P.302
- Note that in P.303: Only the last argument(s) can be omitted.
  - do_it(30, 30) is legal.
  - do_it(30,   , 30, 30) is illegal.

Function Overloading (P.310)

- Normally, we need three distinct functions to handle three different data types:
  - `int max_int(int array[], int len);`
  - `long max_long(long array[], int len);`
  - `double max_double(double array[], int len);`
- Function overloading allows you to use the same function name for defining several functions as long as they each have different parameter lists.
- When the function is called, the compiler chooses the correct version according to the list of arguments you supply.

Ex6_07.cpp on P.311

- The following functions share a common name, but have a different parameter list:
  - `int max(int array[], int len);`
  - `long max(long array[], int len);`
  - `double max(double array[], int len);`
- Three overloaded functions of max()
- In main(), C compiler inspect the argument list to choose different version of functions.

Signature

- Overloaded functions can be differentiated by
  - having corresponding parameters of different types, or
  - having a different number of parameters.
- The signature of a function is determined by its name and its parameter list.
- All functions in a program must have unique signatures.

- The following example is not valid overloading
  - `double max(long array[], int len);`
  - `long max(long array[], int len);`
- A different return type does not distinguish a function, if the signatures are the same.

If Signature Is Not Unique …

```
#include <iostream>
using std::cout;
using std::endl;

int sum(int a, float b)
{ return a+ static_cast<int>(b); }

float sum(int a, float b)
{ return static_cast<float>(a) + b; }

int main()
{
    cout << sum(1, 2.5) << endl;
    return 0;
}
```

Which version do you invoke?

Function Templates

- In Ex6_07.cpp, you still have to repeat the same code for each function (in P.312), with different variable and parameter types.

- You may define a function template to ask C compiler automatically generate functions with various parameter types.

Defining a Function Template

```
template<typename T> T max(T x[], int len)
{
    T max = x[0];
    for (int i = 1; i < len; i++)
        if (max < x[i])
            max = x[i];
    return max;
}
```

Using a Function Template

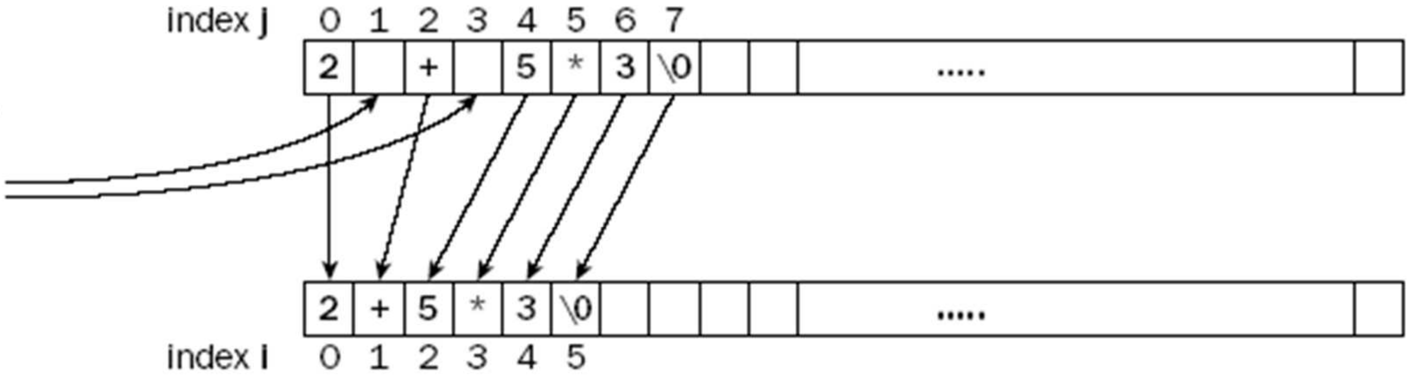
- Each time you use the function `max()` in your program, the compiler checks to see if a function corresponding to the type of arguments that you have used in the function call already exists.
  - If the function does not exist, the compiler creates one by substituting the argument type in your function call to replace the parameter T.
- Compare Ex6_08.cpp and Ex6_07.cpp to see how the source code is reduced.
  - Note that using a template doesn't reduce the size of your compiled program.
- Q: Can we calculate the length of the array inside the function?

Case Study: Implementing a Calculator

- Goal
  - Design a program which acts as a calculator.
  - It will take an arithmetic expression, evaluate it, and print out the result.
  - For example, taking the input string
    “2 * 3.14159 *   12.6   * 12.6 /2   + 25.2 * 25.2”
    will obtain the result “1133.0”.
- To make it simple at the first stage,
  - The whole computation must be entered in a single line.
  - Spaces are allowed to be placed anywhere.
  - Parentheses are not allowed in the expression.
  - Only unsigned numbers are recognized.

Step 1: Eliminating Blanks from a String

The buffer array before copying its contents to itself



Index i is not incremented at these positions because they contain a space.
These spaces are overwritten by the next non-space character that is found in **buffer**.

The buffer array after copying its contents to itself

Figure 6-2

An Intuitive Code for Eliminating Blanks

```
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    const int MAX = 80;
    char str1[MAX] = "NCNU is a good university.";
    char str2[MAX];
    int i=0, j=0;
    do {
        if (str1[i] != ' ')
            str2[j++] = str1[i];
    } while (str1[i++] != '\0');

    cout << str2 << endl;
    return 0;
}
```

P.322

```
// Function to eliminate spaces from a string
void eatspaces(char* str)
{
   int i = 0;     // ‘Copy to’ index to string
   int j = 0;     // ‘Copy from’ index to string

   while ((*(str + i) = *(str + j++)) != ‘\0’)
      if (*(str + i) != ‘ ‘)
         i++;
   return;
}
```

- Now, we obtain an expression with no embedding spaces.

Step 2: Evaluating an Expression

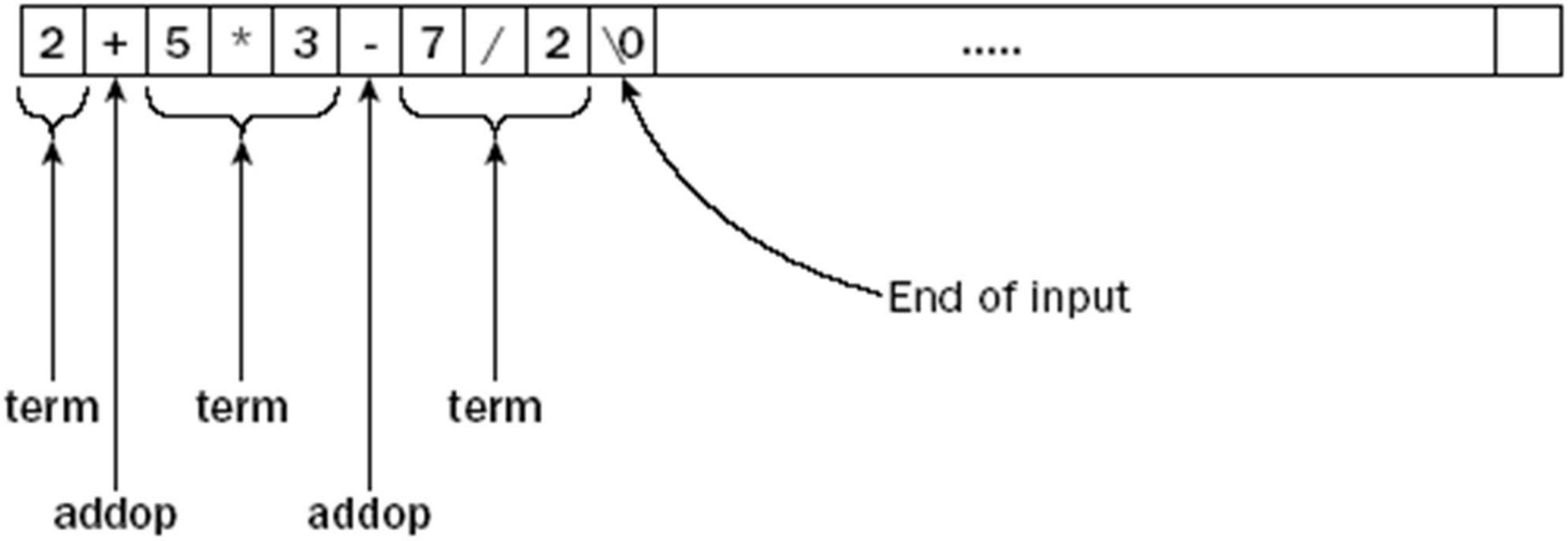
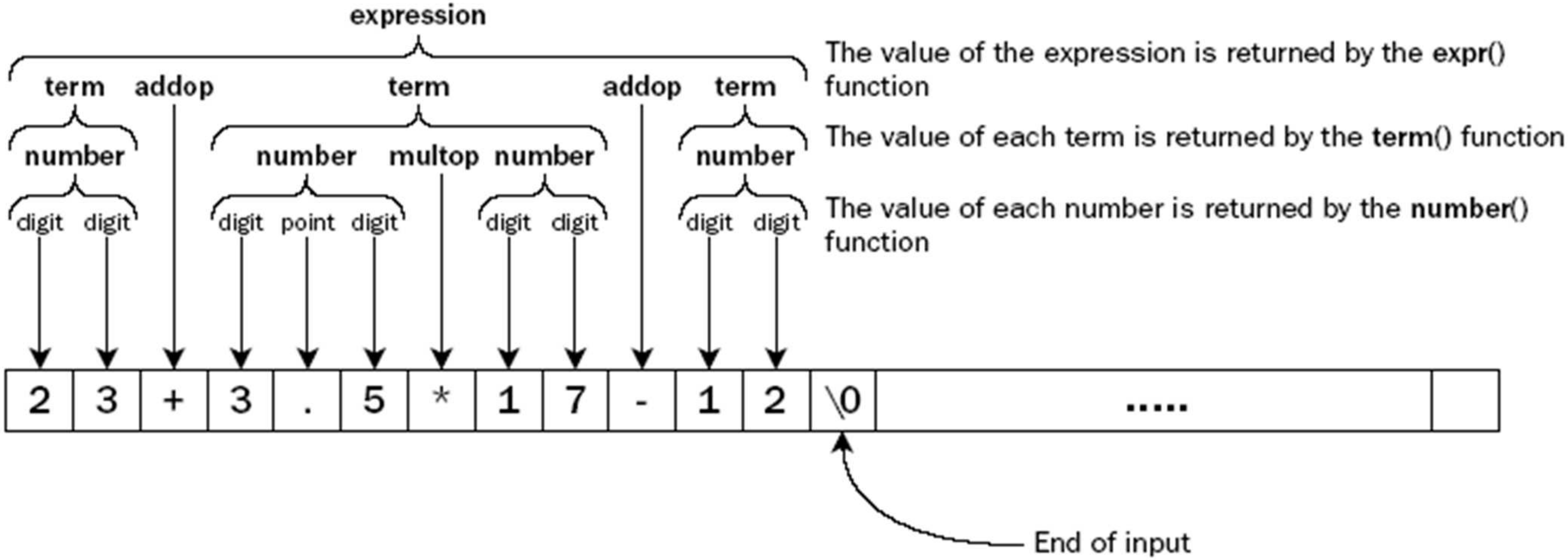


FIgure 6-3

expression: term addop term addop … term

Breaking Down an Expression into Terms and Numbers



The value of the expression is returned by the **expr()** function

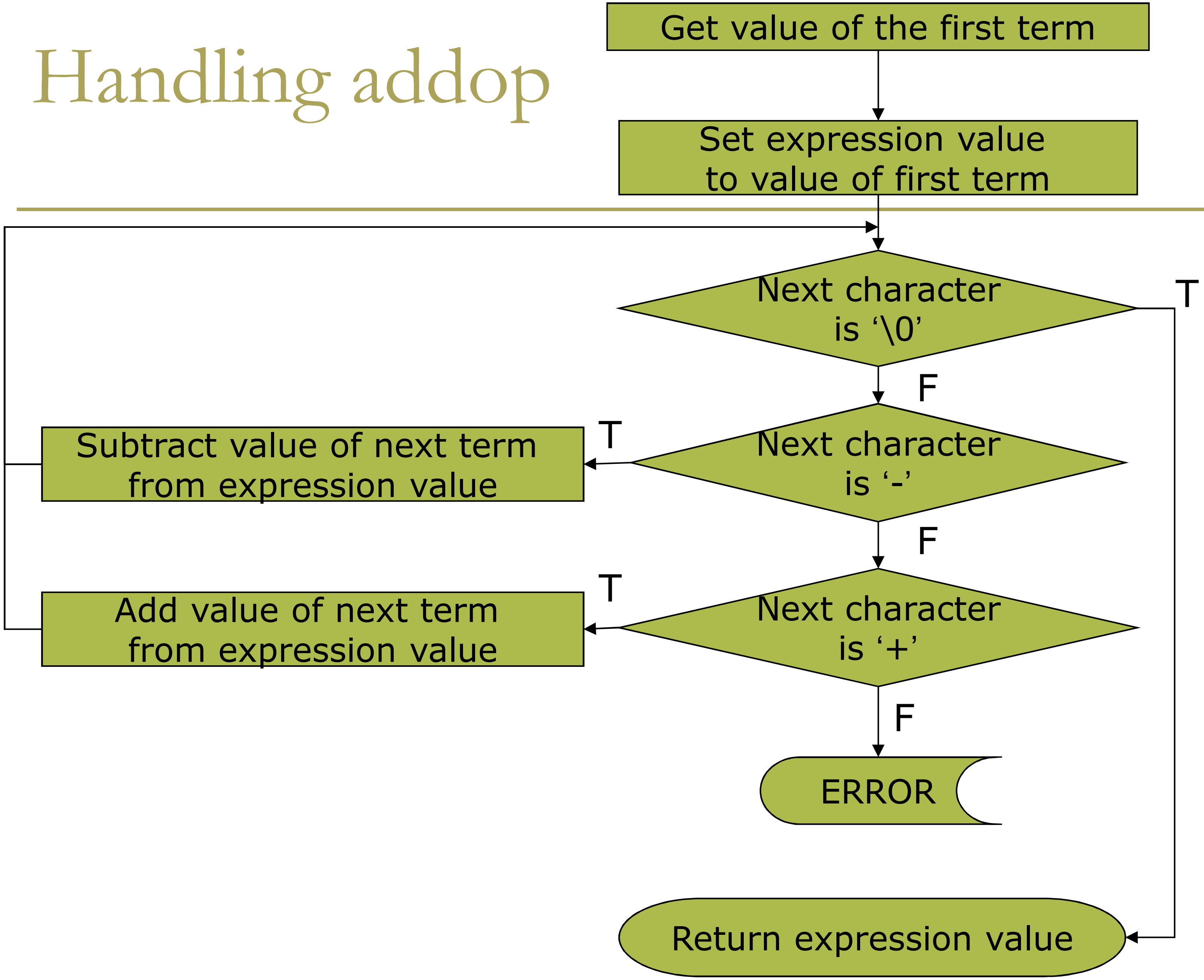
The value of each term is returned by the **term()** function

The value of each number is returned by the **number()** function

End of input

Figure 6-4

Handling addop

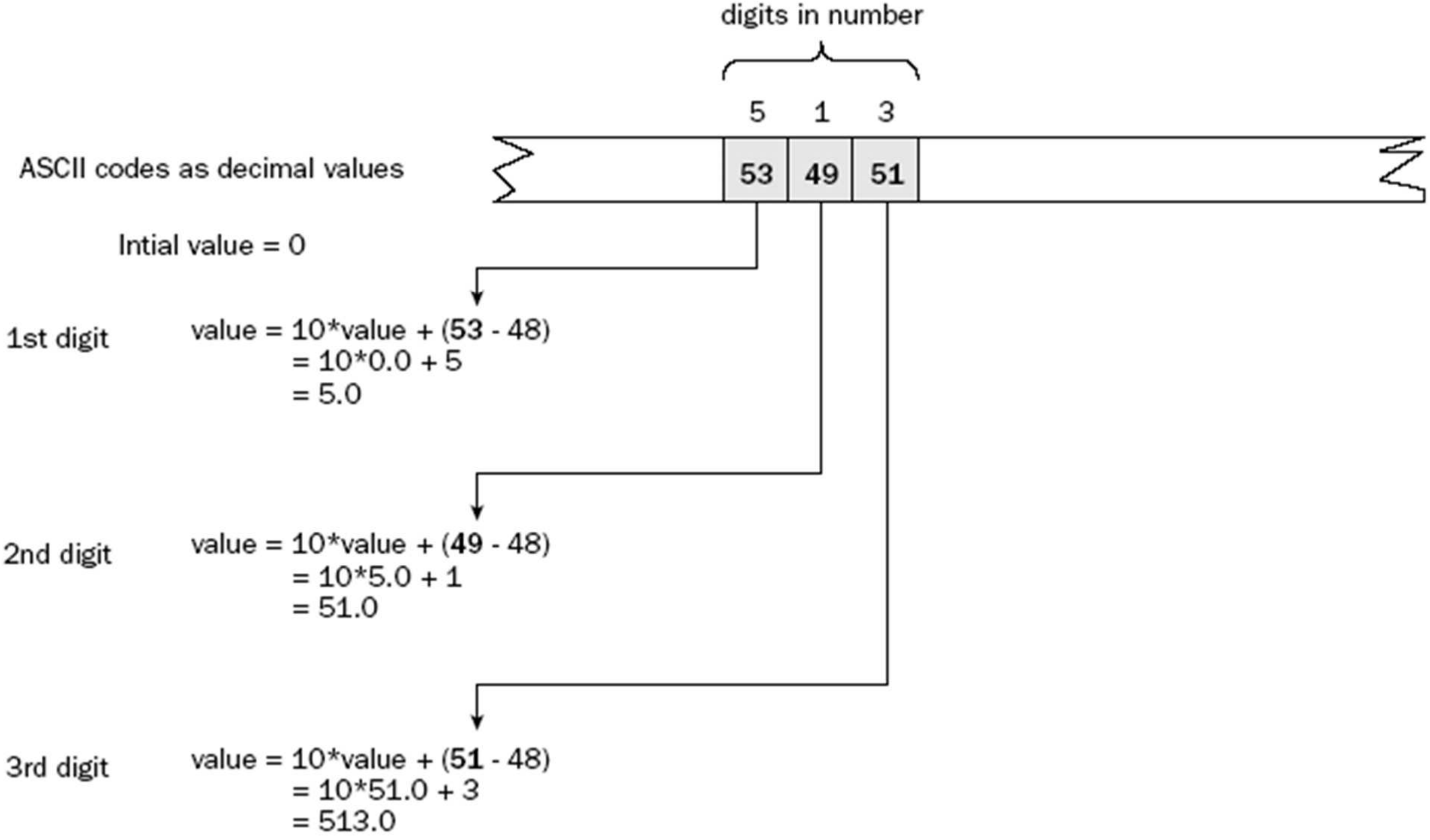

Get value of the first term
Set expression value to value of first term
Next character is '\0'
T
F
Next character is '-'
T
Subtract value of next term from expression value
F
Next character is '+'
T
Add value of next term from expression value
F
ERROR
Return expression value

```
double expr(char* str)

{
    double value = 0.0;
    int index = 0;

    value = term(str, index);

    for (;;)
    {
       switch (*(str + index++))
       {
          case ‘\0’:
             return value;
          case ‘-’:
             value -= term(str, index);
          case ‘+’:
             value += term(str, index);
          default:
             cout << endl << “Arrrgh!*#!! There’s an error” <<
      endl;
             exit(1);
       }
    }
}
```

Getting the value of a Term (P.325)

```
// Function to get the value of a term
double term(char* str, int& index)
{
    double value(0.0);                // Somewhere to accumulate the result

    value = number(str, index);       // Get the first number in the term

    // Loop as long as we have a good operator
    while (true)
    {
        if (*(str + index) == '*')              // If it's multiply,
            value *= number(str, ++index);      // multiply by next number
        else if (*(str + index) == '/')         // If it's divide,
            value /= number(str, ++index);      // divide by next number
        else
            break;
    }
    return value;
}
```

Analyzing a Number



digits in number

5 1 3

ASCII codes as decimal values

53 49 51

Intial value = 0

1st digit value = 10*value + (**53** - 48)
= 10*0.0 + 5
= 5.0

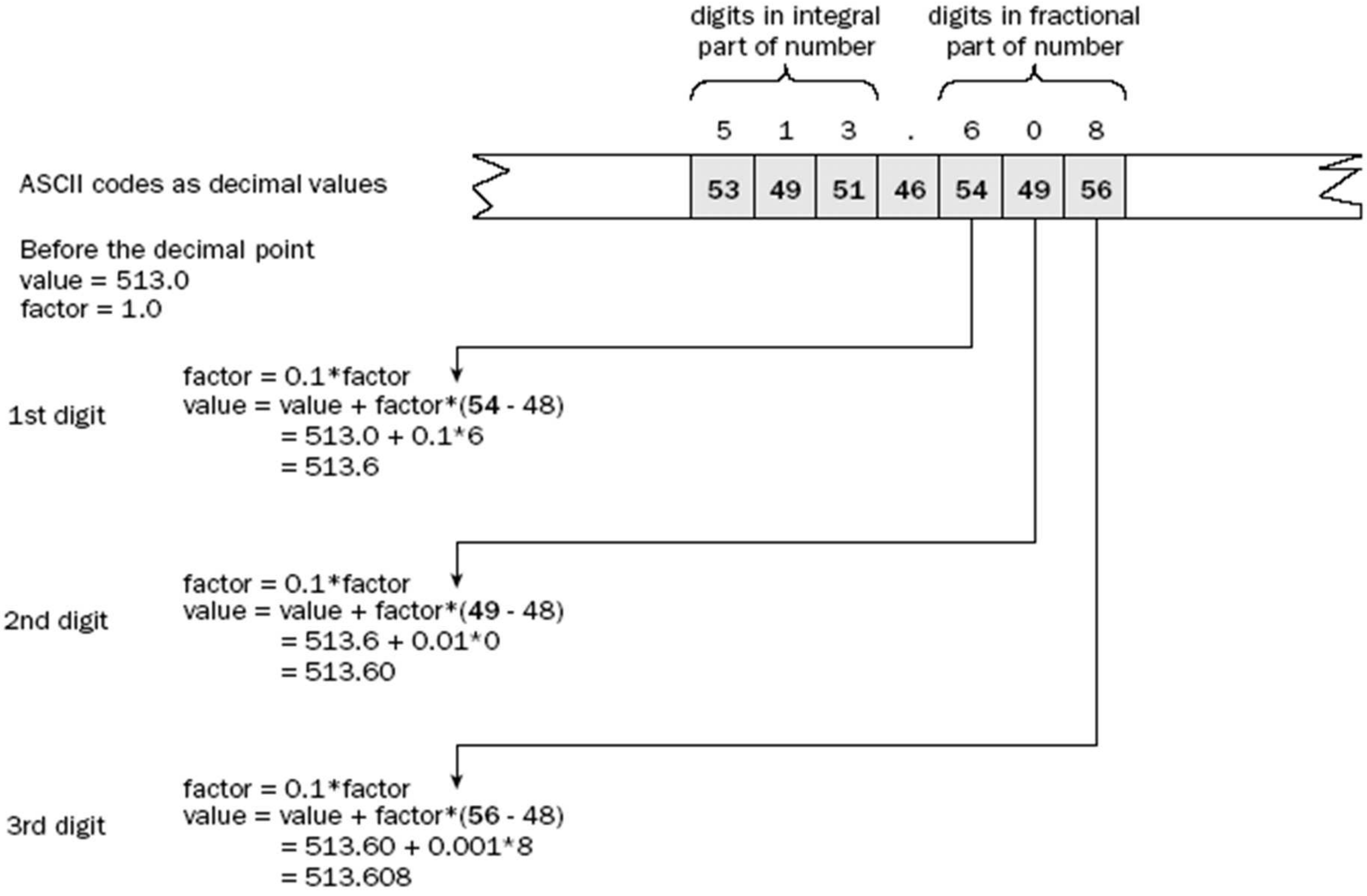
2nd digit value = 10*value + (**49** - 48)
= 10*5.0 + 1
= 51.0

3rd digit value = 10*value + (**51** - 48)
= 10*51.0 + 3
= 513.0

Figure 6-6

```
double number(char* str, int& index)
{
   double value = 0.0;

   while (isdigit(*(str + index)))
        value = 10 * value + ( *(str + index++) – ‘0’);

   if (*(str + index) != ‘.’)
        return value;

   double factor = 1.0;
   while (isdigit(*(str + (++index))))
   {
        factor *= 0.1;
        value = value + ( *(str + index) – ‘0’) * factor;
   }

   return value;
}
```

Handling the fractional part after the decimal point



digits in integral part of number

digits in fractional part of number

ASCII codes as decimal values

5	1	3	.	6	0	8
53	49	51	46	54	49	56

Before the decimal point
value = 513.0
factor = 1.0

1st digit

```
factor = 0.1*factor
value = value + factor*(54 - 48)
         = 513.0 + 0.1*6
         = 513.6
```

2nd digit

```
factor = 0.1*factor
value = value + factor*(49 - 48)
         = 513.6 + 0.01*0
         = 513.60
```

3rd digit

```
factor = 0.1*factor
value = value + factor*(56 - 48)
         = 513.60 + 0.001*8
         = 513.608
```

Figure 6-7

Putting the Program Together

- P.330 Ex6_10.cpp
  - `#include <iostream>// For stream input/output`
  - `#include <cstdlib> // For exit() function`
  - `#include <cctype>  // For isdigit() function`

- Use `cin.getline()` so that the input string can contain spaces.
  - See P.175

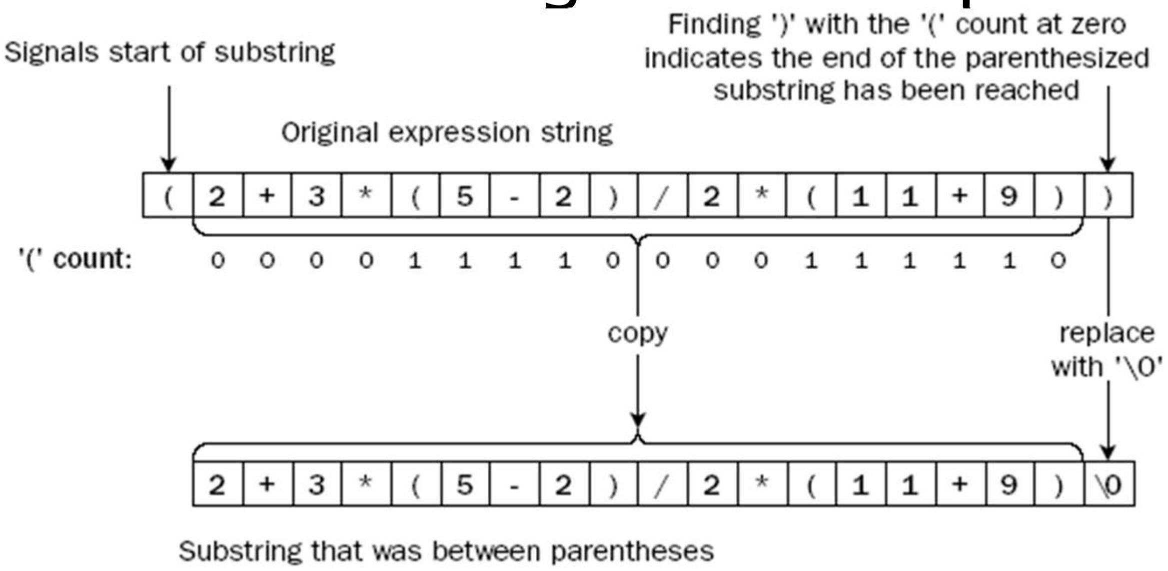
Extending the Program

- Let us try to extend it so that it can handle parentheses:
  - 2* (3 + 4) / 6 – (5 + 6) / (7+ 8)

- Idea: treat an expression in parentheses as just another number.
  - P.332
  - `expr()` recursively calls itself
    - `expr()` → `term()` → `number()` → `expr()`
  - The string pointed by `psubstr` is allocated in `extract()`, and must be freed as an array.

extract()

- Extract a substring between parentheses



Signals start of substring

Finding ')' with the '(' count at zero indicates the end of the parenthesized substring has been reached

Original expression string

(2 + 3 * (5 - 2) / 2 * (1 1 + 9))

'(' count: 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0

copy

replace with '\0'

2 + 3 * (5 - 2) / 2 * (1 1 + 9) \0

Substring that was between parentheses

Figure 6-8

- P.334
  - Utilize `strcpy_s()` which is defined in `<cstring>` header file

Exercise: Modify `number()`

- Implement the simple calculator defined in P.318—336.
- Modify the `number()` function defined in P.327 so that it will take hexadecimal strings as input. The return type will thus become an integer.
  - You may test the modified program by providing an input string “A + B”, which should result in “15”. Another expression “B – A” will result in “1”.