

More on Functions

- Overloaded Functions
- Use pointers and references as parameters
- Template functions
- Function pointers
- Recursive functions

Function overloading

- With function overloading, you can have several functions in a program with the same name
- Two functions with the same name are different if one of the following is true
 - The number of parameters for each function is different
 - The number of parameters is the same, but at least one pair of corresponding parameters has different types

The Signature of a Function

- The combination of the name of a function, together with its parameter types defines unique characteristic called the function signature
- The return type is not part of the function signature
- Overloaded functions with pointer or reference parameters are different
- Program 9.1

Overloading and Pointer Parameters

- Different

```
int larger(int*pV1, int* pV2);  
int larger(float*pV1, float*pV2);
```

- Identical

```
int largest(int values[], int count);  
int largest(int* values, int count);
```

Overloading and Reference Parameters

- It is not allowed to overload a function with a parameter of a given type *data_type*, with a function that has a parameter of type *reference to data_type*
- Be careful that it may cause unpredictable results

```
int dolt(int number); // These are not  
int& dolt(int &number); // distinguishable  
dolt(value);           // calling
```

- Program 9.2
- The compiler is not prepared to use a temporary address to initialize a reference

Overloading and const Parameters

- Functions with constant parameters and non-constant parameters are the same signature
 - `long& larger(long a, long b);`
 - `long& larger(const long a, const long b);`
- The compiler ignores the const aspect of the parameters
- Functions with pointer to constant parameter and pointer to non-constant parameter are different
 - `long* larger(long* a, long* b);`
 - `long* larger(const long* a, const long* b);`

Overloading with const Pointer Parameters

- Different

```
long* larger(long* a, long* b);           // Pointer parameter  
const long* larger(const long* a, const long* b); // Pointer to const
```

```
long num1 = 1;  
long num2 = 2;  
long num3 = *larger(num1, num2);
```

```
const long num10 = 1;  
const long num20 = 2;  
const long num30 = *larger(num10, num20);
```

Overloading with const Pointer Parameters

- Identical

```
long* larger(long* a, long* b);           // These are  
long* const larger(long* const a, long* const b); // identical
```


Overloading and const Reference Parameters

- Different

```
long& larger(long& a, long& b);  
long larger(const long& a, const long& b);
```

- Type `int&` is always different from type `const int&`
- The return value has no effect on the overloading

Pointer to const and non-const Parameters

- ```
long* larger(long*, long*); // prototype
long num1=1, num2=2; // in main()
long num3 = *larger(&num1, &num2); // calling
```
- ```
long* larger(const long*, const long*); //prototype  
const long num10=1, num20=2; // in main()  
long num30=*larger(&num10,&num20); //calling
```

Pointer to const Type & Constant Pointer to non-const Type

- Reference to “補充Pointer.doc”
- Identical signature

```
long* larger(long* a, long* b);
```

```
long* larger(long* const a, long* const b);
```

const Reference Parameter

- It is much more easier to recognize a constant reference parameter than a constant pointer parameter

- Different

```
long& larger(long& a, long& b);
```

```
const long& larger(const long& a, const  
long& b);
```

- Program ft_sig.cpp

Overloading and Default Argument Values

- `const char*`

```
void show_error(const char* message) {  
    std::cout << std::endl << message << std::endl;  
}
```

- `const string&`

```
void show_error(const string& message) {  
    std::cout << std::endl << message << std::endl;  
}
```

- You can't define a default argument for both functions due to ambiguity

Template Functions

The keyword `template` identifies this code as a template

The keyword `class` identifies `T` as a type. You put the template parameters between angled brackets after the keyword `template`. They are separated by commas if there is more than one.

This `T` is the parameter for the template. It serves to identify where the type for a particular instance has to be substituted in the code. In this case, it is the return type and both parameter types.

```
template <class T> T larger( T a, T b ) {  
    return a>b ? a : b;  
}
```

The symbol `T` stands for the type that is to be replaced when a specific instance of a function is created. Wherever `T` appears in the template definition, it will be replaced by a specific type.

Defining a Template Specialization

- Explicitly specifying a template parameter

```
cout << "Larger of " << a_int << " and " << b_int << " is "  
    << larger<long>(a_int, b_int)  
    << endl;
```

- Program 9.4

Function Templates and Overloading

- Overloaded function

```
long* larger(long* a, long* b);
```

```
long* larger(long* a, long* b) {  
    cout << "overloaded version for long* " << endl;  
    return *a>*b ? a : b;  
}
```


Overloaded template

```
template <class T> T larger (const T array[], int count) {  
    cout << "template overload version for arrays " << endl;  
    T result = array[0];  
    for(int i = 1 ; i < count ; i++)  
        if(array[i] > result)  
            result = array[i];  
    return result;  
}
```

```
double x[] = { 10.5, 12.5, 2.5, 13.5, 5.5 };  
cout << "Largest element has the value "  
    << larger(x, sizeof x/sizeof x[0])  
    << endl;
```

Templates with Multiple Parameters

```
// Template for functions to return the larger of two values
template <class TReturn, class TArg> TReturn larger(TArg a, TArg b) {
    return a>b ? a : b;
}
```

```
cout << "Larger of 1.5 and 2.5 is "
      << larger<int>(1.5, 2.5)
      << endl;
```

```
cout << "Larger of 1.5 and 2.5 is "
      << larger<double, double>(1.5, 2.5)
      << endl;
```

Non-Type Template Parameters

```
template <class T, int upper, int lower> bool is_in_range(T value) {  
    return (value <= upper) && (value >= lower);  
}
```

```
double value = 100.0;  
std::cout << is_in_range(value);    // Won't compile - incorrect usage
```

```
cout << is_in_range<double,0,500>(value); // OK
```

Pointers to Functions

- A pointer can point to the address of a function. Such a pointer can point to different functions at different times during execution of your program.
- A pointer point to different functions with the same data types and the return type

Declaring Pointers to Functions

```
return_type (*pointer_name)(list_of_parameter_types);
```

```
long (*pfun)(long*, int);    // Pointer to function declaration
```

```
long max_element(           long* array, int count);    // Function prototype
```

```
long (*pfun)(long*, int) = max_element;
```

- Program 9.5

Passing a Function as an Argument

- A function passed to another function as an argument is sometimes referred to as a **callback function**
- Program 9.6

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};
```

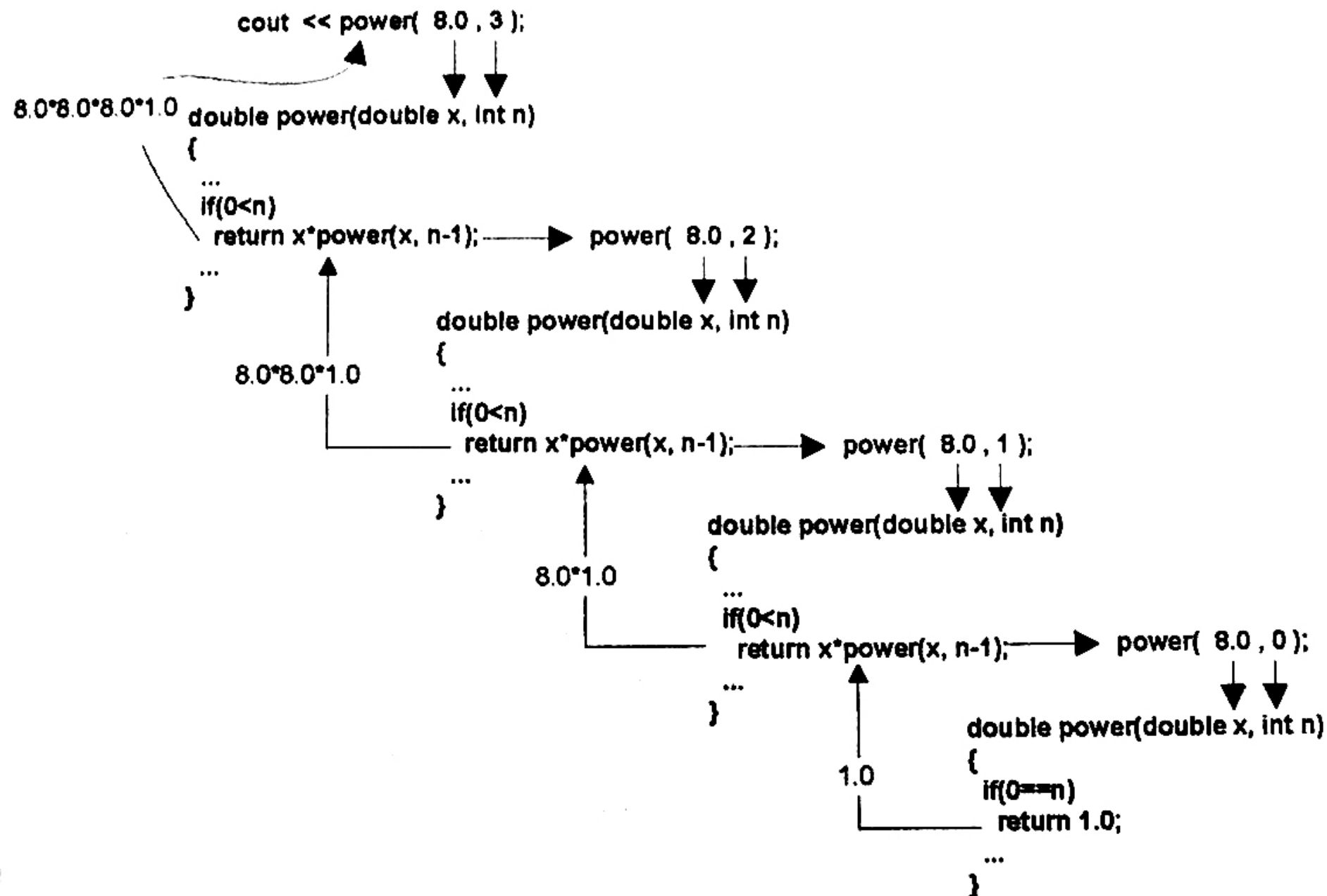
Recursion

- When a function contains a call to itself, it is referred to as a **recursive function**
- Indirect recursive function call
 - Function1 calls function2, and function2 calls function1, alternatively.
- A prerequisite for avoiding a loop of unlimited duration is that the function must contain some means of stopping the process
- Program 9.7

Loop and Recursion

```
double power(double x, int n) {  
    return 0==n ? 1.0 : (0>n ? 1.0/power(x,-n) :  
        x*power(x,n-1));  
} // same as followings
```

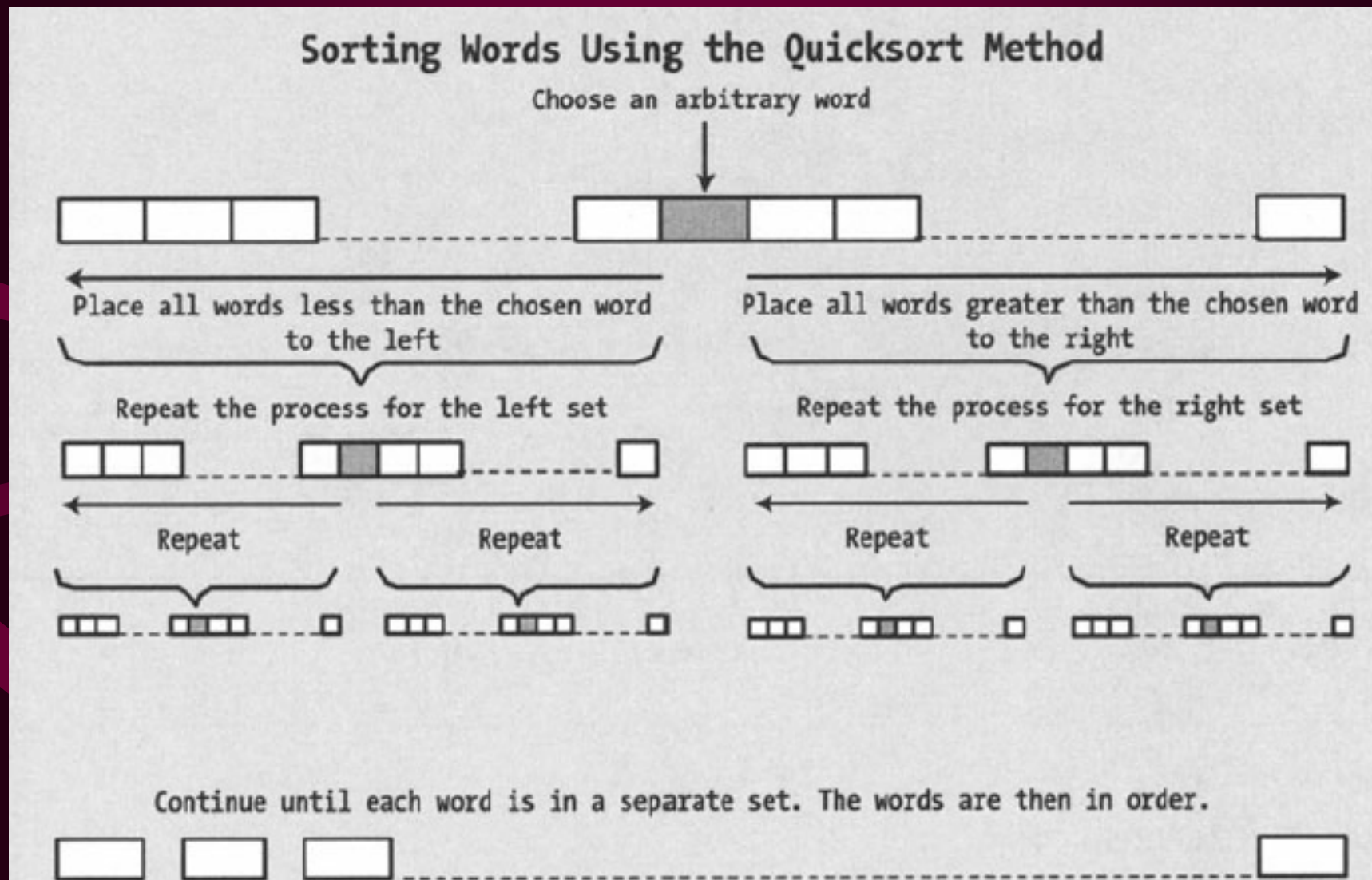
```
double power(double x, int n) {  
    if (0==n) return 1.0;  
    if (0<n) return x*power(x,n-1);  
    return 1.0/power(x, -n);  
}
```



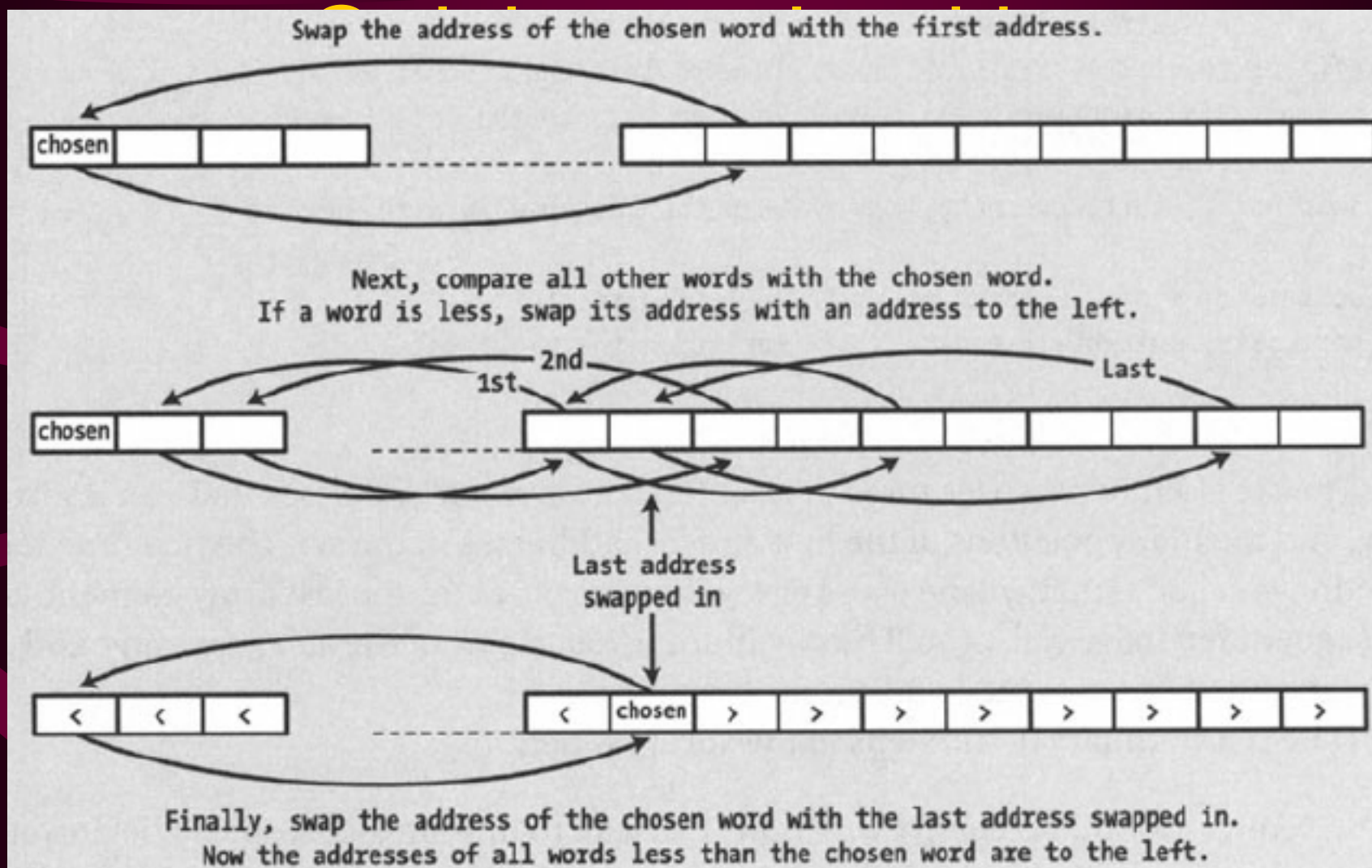
More Efficient Loop

```
double power(double x, int n)
{
    if (n==0) return 1.0;
    if (n<0)
        { x = 1.0/x;  n = -n; }
    double result = x;
    for (int i =1; i < n; i++)
        result *=x;
    return result;
}
```

Quicksort Using Recursion



Swapping addresses in the



變數 current 記錄比 chosen 小的個數做為 Index

Swapping Addresses

```
void swap(string* pStr[], int first, int second)
{
    string* temp = pStr[first];
    pStr[first] = pStr[second];
    pStr[second] = temp;
}
```

Counting Words

```
int count_words(const string&text, const string& separators) {  
    size_t start = text.find_first_not_of(separators);  
    size_t end = 0;  
    int word_count = 0;  
    while (start != string::npos) {  
        end = text.find_first_of(separators, start+1);  
        if (end == string::npos) end = text.length();  
        word_count++;  
        start = text.find_first_not_of(separators, end+1);  
    }  
    return word_count;  
}
```

Recursive function

```
void sort(string*pStr[], int start, int end) {  
    if (!(start<end)) return;  
    swap(pStr, start, (start+end)/2);  
    int current = start;  
    for (int i = start+1; i<= end; i++)  
        if (*(pStr[i]) < *(pStr[start]))  
            swap(pStr, ++current, i);  
    swap(pStr, start, current);  
    sort(pStr, start, current-1);  
    sort(pStr, current+1, end);  
}
```


Extracting Words

```
void extract_words(string* pStr[], const string& text,  
    const string& separators) {  
    size_t start = text.find_first_not_of(separators);  
    size_t end = 0, index = 0;  
    while(start != string::npos) {  
        end = text.find_first_of(separators, start+1);  
        if (end == string::npos) end = text.length();  
        pStr[index++] = new string(text.substr(start,  
            end-start));  
        start = text.find_first_not_of(separators, end+1);  
    }  
}
```

Quick Sort

- Program 9.8