Chapter 13

Functions and Parameter Passing (Part 2)

13.1 Passing Arguments to Functions

13.1.1 Passing Pointers

C++ allows to pass a pointer to a function. Then the parameter has to be declared as a pointer type.

```
void f(int *j) { *j = 100; }

int main()
{
  int i; int *p;

  p = &i; // p now points to i
  f(p);
  cout << i; // i is now 100
  return 0; }
```

The pointer variable `p` is actually not necessary:

```
void f(int *j);

int main()
{
  int i;

  f(&i);

  cout << i;
  return 0;
}

void f(int *j)
{
  *j = 100;
}
```

13.1.2 Passing Arrays

When an array is an argument to a function, only the **address** of the **first** element of the array is passed, not a copy of the entire array.

Note: In C++, an array name without any index is a **pointer** to the first array element. This means that the formal parameter declaration has to be of a compatible type.

There are three ways to declare a parameter that is to receive an array pointer:

- as an array of the same type and size as that used to call the function
- as an unsized array
- as a pointer

The following three example programs illustrate these three possibilities.

Passing Arrays (1): with size specification

```
void display(int n[10]);

int main()
{
  int t[10], i;

  for(i=0; i<10; ++i) t[i] = i;
  display(t); // pass array t to function
  return 0;
}

void display(int n[10])
{
  int i;
  for(i=0; i<10; i++) cout << n[i] << ' ';
}
```

Passing Arrays (2): as an unsized array

```
void display(int n[]);

int main()
{
  int t[10], i;

  for(i=0; i<10; ++i) t[i] = i;
  display(t); // pass array t to function
  return 0;
}

void display(int n[])
{
  int i;
  for(i=0; i<10; i++) cout << n[i] << ' ';
}
```

Passing Arrays (3): using a pointer

```
void display(int *n);

int main()
{
  int t[10], i;

  for(i=0; i<10; ++i) t[i] = i;
  display(t); // pass array t to function
  return 0;
}

void display(int *n)
{
  int i;
  for(i=0; i<10; i++) cout << n[i] << ' ';
}
```

Passing Arrays (4): using a pointer and array size information

```
void display(int *n, int s);

int main()
{
  int t[10], i;

  for(i=0; i<10; ++i) t[i] = i;
  display(t, 10);
  return 0;
}

void display(int *n, int s)
{
  int i;
  for(i=0; i<s; i++) cout << n[i] << ' ';
}
```

Important to remember: As for arrays, only addresses are passed to a function, the function will potentially alter the contents of the array.

```
void cube(int *n, int num)
{
  while(num) {
    *n = *n * *n * *n;
    num--;
    n++;
} }

void main()
{
  int i, nbs[10];

  for(i=0;i<10;i++) nbs[i] = i+1;
  cube(nbs,10);
  for(i=0;i<10;i++) cout << nbs[i] << ' ';}
```

13.1.3 Passing Strings

Strings are simply character arrays that are null-terminated.

When a string is passed to a function, only a pointer to the beginning of the string (of type **char ***) is actually passed.

The following code fragment defines a function that converts strings to uppercase:

```
void stringToUpper(char *str);

int main()
{
  char str[80];
  strcpy(str, “this is a test”);
  stringToUpper(str);
  cout << str;
  return 0; }

void stringToUpper(char *str)
{
  while(*str) {
    *str = toupper(*str); // one character
    str++; /* move on to next char */ }}
```

13.2 Parameter Passing Mechanisms

13.2.1 General Parameter Passing Mechanisms

- **Call by Value**

  The *value* of an argument is copied into the formal parameter of the subroutine.

  --> Changes made to the parameters of the subroutine will not affect the arguments used to call it.

- **Call by Name**

  The formal parameter is textually replaced by the calling argument.

- **Call by Reference**

  The *address* of an argument is copied into the parameter.

  --> Changes made to the parameter will affect the argument used to call the subroutine.

Example: We use the following pseudo-code for defining a function, two array elements, and a variable:

f(p) = { i = 2; a[1] = 12; x = p }

a[1] = 10; a[2] = 11; i = 1;

Now consider a call to **f(a[i])**:

- **Call by Value:**

  → f(**a[1]**) = f(**10**) = { i = 2; a[1] = 12; x = **10** }

  → x = 10

- **Call by Reference:**

  → f(&**a[1]**) = { i = 2; a[1] = 12; x = **a[1]** }

  → x = 12

Example (con't): We use the following pseudo-code for defining a function, two array elements, and a variable:

f(p) = { i = 2; a[1] = 12; x = p }

a[1] = 10; a[2] = 11; i = 1;

Now consider a call to **f(a[i])**:

- **Call by Name:**

→ f(**a[i]**) = { i = 2; a[1] = 12; x = **a[i]** }

→ x = a[2]

→ x = 11

The call-by-name parameter passing is not provided in C++!

13.2.2 C++ Approaches to Parameter Passing

In general, there are several ways that a computer language can pass an argument to a subroutine.

In C++, there are two methods of parameter passing:

- **Call-by-Value**

  The *value* of an argument is copied into the formal parameter of the subroutine.

  → Changes made to the parameters of the subroutine will not affect the arguments used to call it.

- **Call-by-Reference**

  The *address* of an argument is copied into the parameter.

  Changes made to the parameter will affect the argument used to call the subroutine.

Call-by-Value:

```
int sqr_it(int x);

int main()
{
  int t = 10;

  cout << sqr_it(t) << ' ' << t;

  return 0;
}

int sqr_it(int x)
{
  x = x * x;
  return x;
}
```

13.2.3 Call-by-Reference Using Pointers

One can manually create a **call-by-reference** by passing the address of an argument, i.e., a pointer, to a function.

The following function, which exchanges the values of two variables, uses explicit pointers in its formal parameters.

```
void swap(int *x, int *y)
{
  int temp;

  temp = *x;  // save the value at address x

  *x = *y;    // put y into x

  *y = temp;  // put temp, i.e., x, into y
}
```

Illustration of the swap( ) function — Step 0

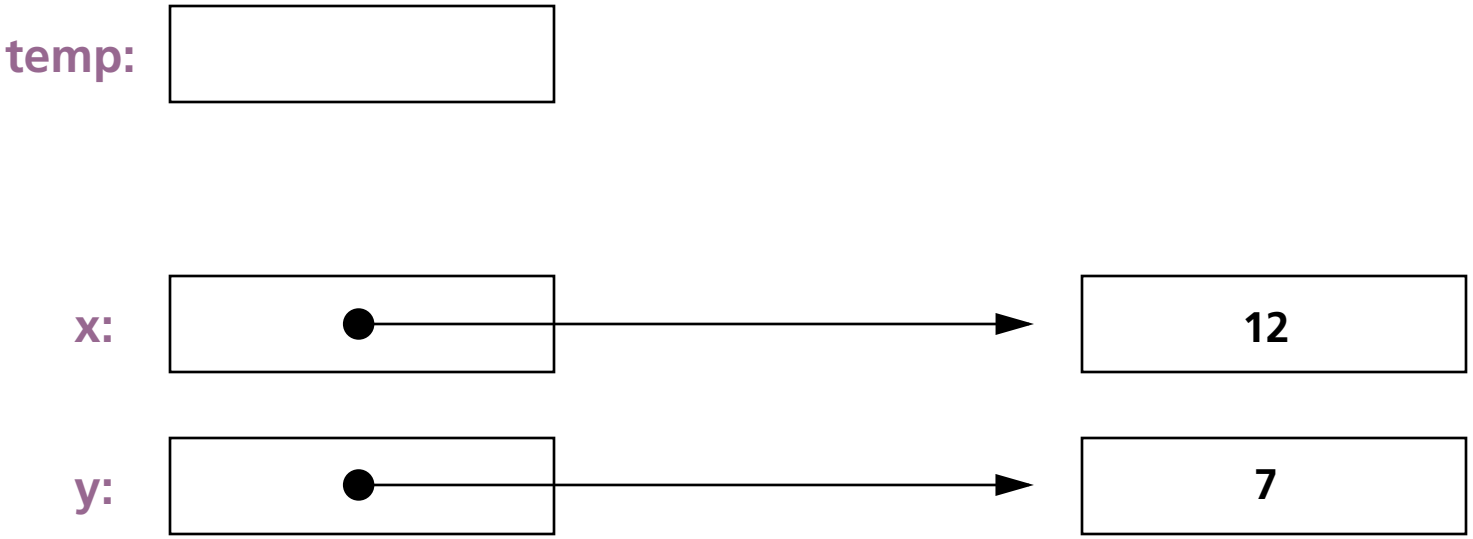
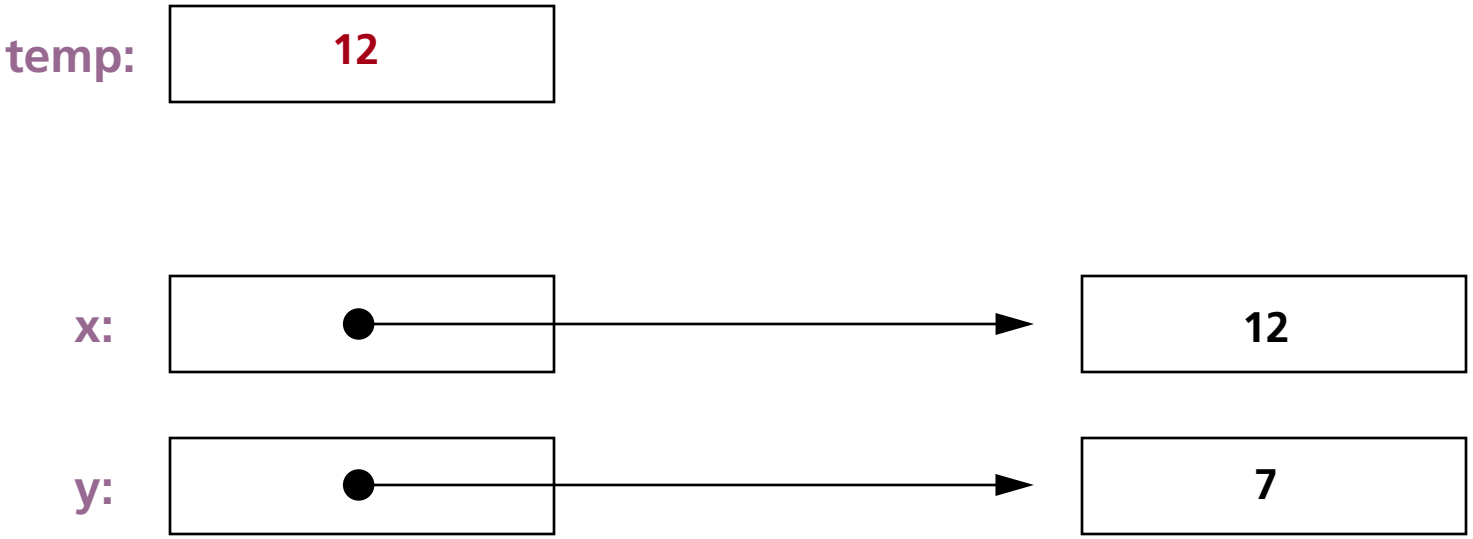


Illustration of the swap( ) function — Step 1

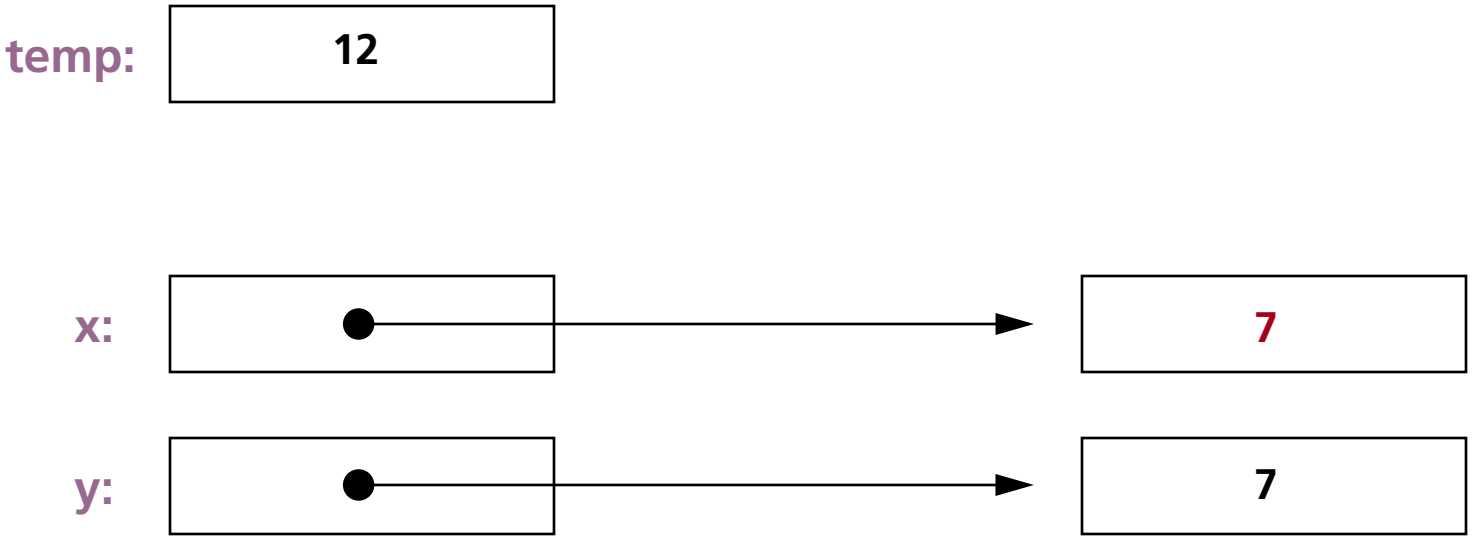



```
temp = *x;   // save the value at address x

*x = *y;     // put y into x

*y = temp;   // put temp into y
```

Illustration of the swap( ) function — Step 2

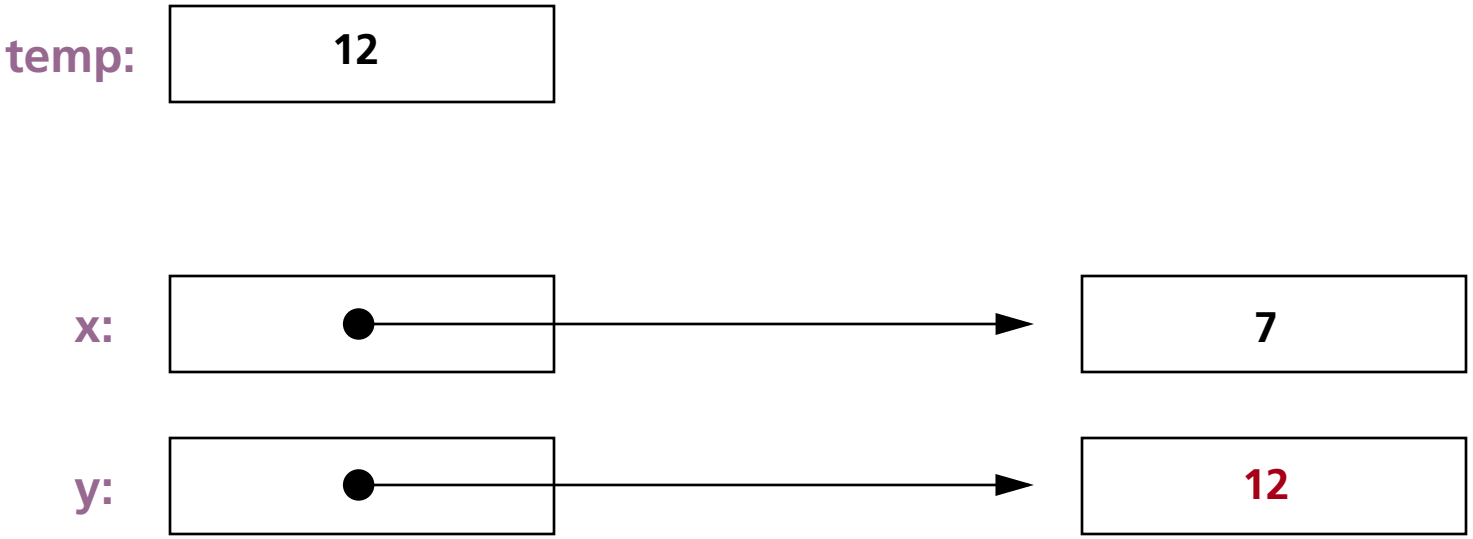



```
temp = *x;   // save the value at address x

*x = *y;     // put y into x

*y = temp;   // put temp into y
```

Illustration of the swap( ) function — Step 3




```
temp = *x;   // save the value at address x

*x = *y;     // put y into x

*y = temp;   // put x into y
```

Here is an example code fragment how to use the swap function:

```
int main()
{
  int i, j;

  i = 10;
  j = 20;

  cout << "initial values of i and j: ";
  cout << i << ' ' << j << '\n';

  swap(&i, &j);

  cout << "swapped values of i and j: ";
  cout << i << ' ' << j << '\n';

  return 0; }
```

Note: A less desireable solution is using **swap( )** without any references to variables, because it will work only on global variables, and only on these particular variables:

```
a = 12; b = 7;

void swap()
{
  int temp;

  temp = a;

  b = a;

  a = temp; }
```

13.2.4 Reference Parameters

Instead of using pointers to manually achieve a call-by-reference, it is possible to tell the C++ compiler to **automatically use call-by-reference** for the formal parameters of a particular function.

- A reference parameter is declared by preceding the parameter name in the function´s declaration with an **&**.
- When a reference parameter is created, that parameter automatically refers to the argument used to call the function.
- The reference parameter implicitly points to the argument.

Operations performed on a reference parameter affect the argument used to call the function, not the reference parameter itself.

There is no need to apply the **&** operator to an argument.

Inside the function, the reference parameter is used directly; i.e., the * operator is not necessary. In fact, it is not correct to use the * operator for reference parameters.

Here is a simple example to understand the use of reference parameters:

```
void f(int &i)
{
  i = 10;  // modifies the calling argument
}

int main()
{
  int val = 1;

  cout << "Old val: " << val << '\n';

  f(val); // pass address of val to f()

  cout << "New val: " << val << '\n';
  return 0; }
```

Here is the version of the **swap example**, now using reference parameters:

```
void swap (int &x, int &y)
{
  int temp;

  temp = x;  // save the value at address x

  x = y;     // put y into x

  y = temp;  // put x into y
}
```

```
int main()
{
  int i, j;

  i = 10;
  j = 20;

  cout << “initial values of i and j: “;
  cout << i << ‘ ‘ << j << ‘\n’;

  swap(i, j);

  cout << “swapped values of i and j: “;
  cout << i << ‘ ‘ << j << ‘\n’;

  return 0;
}
```

13.2.5 Returning References

A function can return a reference, which means, it returns an implicit pointer to its return value.

→ The function can be used on the **left side** of an assignment statement!

```
double val = 100;


double &f()
{
  return val; // return reference to val
}
```

```
double val = 100;

double &f()
{
  return val; // return reference to val
}

int main()
{
  double newval;

  newval = f(); // assign value of val

  f() = 99.1;   // change val´s value
                // reference to val becomes
                // target of assignment
  return 0; }
```

Here is another example program using a reference return type:

```
double v[] = {1.1, 2.2, 3.3, 4.4, 5.5};

double &change_element(int i)
{
  return v[i]; // reference to i-th element
}

void main()
{
  int i;
  for(i=0; i<5; i++) cout << v[i] << ' ';

  change_element(1) = 523.9; // second element
  change_element(3) = -98.7; // 4th element

  for(i=0; i<5; i++) cout v[i] << ' '; }
```

Mind the Scope of References!

When returning a reference, one has to be careful that the object being referred to does not go out of scope:

```
// Error: cannot return reference to local var

int &f()
{
  int i = 10;
  return i;
}
```

13.2.6 Restrictions When Using References

There are some restrictions that apply to reference variables:

- You cannot reference a reference variable.
- You cannot create arrays of references.
- You cannot create a pointer to a reference.
- You cannot apply the **&** operator to a reference.
- References are not allowed on bit-fields.

13.3 Command Line Arguments for *main()*

To pass information into a program when you call it, is accomplished by passing **command line arguments** to the program.

A command line argument is the information that follows the program´s name on the command line of the operating system.

Examples:

> ls -a

> ls -la

> diff *file-1 file-2*

> CC *program-name.cc*

> CC -out *prog-ex-name program-name.cc*

> add 123 745

13.3.1 *argc* and *argv*

C++ defines two built-in, but optional, parameters to **main( )**, which receive the command line arguments:

- argc: an **integer**

  The **argc** parameter is an integer that holds the number of arguments on the command line.

  It will always be at least 1, because the name of the program also counts.

- **argv**: a **pointer**

  The **argv** parameter is a pointer to an array of character pointers.

  Each pointer in the **argv** array points to a string containing a command line argument:

  argv[0]: the program´s name

  argv[1]: the first argument

  argv[2]: the second argument …

The following program demonstrates how to access command line arguments. It prints **hello**, followed by the string entered as the first command line argument.

```
int main(int argc, char *argv[])
{
  if(argc != 2) {
    cout << “You forgot to type your
name.\n”;
    return 1;
  }

  cout << “Hello “ << argv[1] << ‘\n’;

  return 0;
}
```

Output of this program:

```
>name Jessica
 Hello Jessica
>
```

Command line arguments should be separated by spaces or tabs.

The following program prints all command line arguments it is called with, one character at a time.

```
int main(int argc, char *argv[])
{
  int t, i;

  for(t=0; t < argc; ++t) {
    i = 0;
    while( argv[t][i] ) {
      cout << argv[t][i];
      ++i;
    }
    cout << ' ';
  }
  return 0; }
```

13.3.2 Passing Numeric Command Line Arguments

All command line arguments are passed to the program as strings.

Hence, numeric arguments have to be converted into their proper internal formats. The following program displays the sum of two numeric command line arguments.

```
int main(int argc, char *argv[])
{
  double a, b;
  if(argc != 3) {
    cout << “Usage: add num num\n”; }

  a = atof(argv[1]);
  b = atof(argv[2]);

  cout << a + b; return 0; }
```

13.3.3 Converting Numeric Strings to Numbers

The C++ standard library, **cstdlib**, includes several functions that allow conversions from the string representation of a number into its internal numerical format:

- atoi( ): numeric string --> integer
- atol( ): numeric string --> long integer
- atof( ): numeric string --> double floating-point

```
int main()
{
  int i; long j; double k;
  i = atoi(“100”);
  j = atol(“100000”);
  k = atof(“-0.123”);

  cout << i << ‘ ‘ << j << ‘ ‘ << k << ‘\n’;
  return 0; }
```

13.4 References

- G. Blank and R. Barnes, *The Universal Machine*, Boston, MA: WCB/ McGraw-Hill, 1998. Chapter 9.
- H. Schildt, C++ from the Ground Up, McGraw-Hill, Berkeley, CA, 1998. Chapter 7.