→ • Expressions and Operators (Chapter 4).

→ • Statements (Chapter 5).

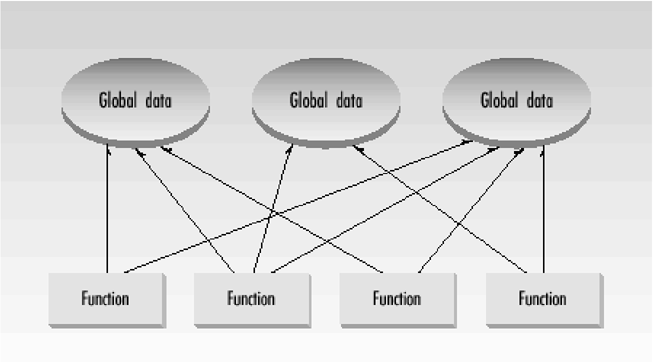
→ • Functions (Chapter 6).

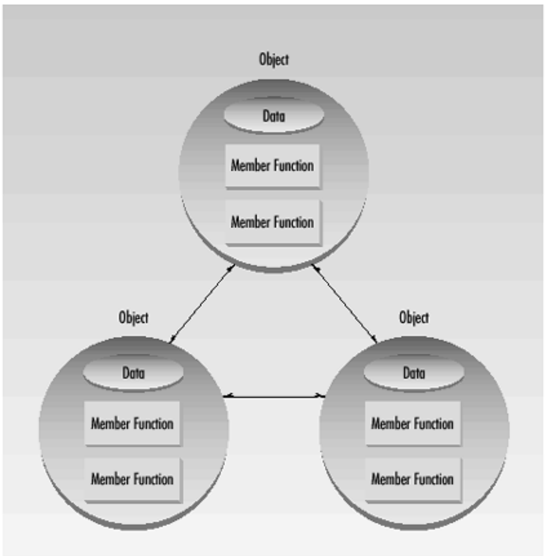
Concise!

Skip!!

Functions

- A function can be thought of as a programmer-defined operations.
- Functions play a key role in procedural programming and an important role in object-oriented programming.





```
// return the greatest common divisor
int gcd(int v1, int v2)
{    while (v2) {
          int temp = v2;
          v2 = v1 % v2;
          v1 = temp;
     }
     return v1;
}
```

- A function is uniquely defined by
  - its name
  - its operand types (parameters).
- The actions of function are specified in a block, referred to as the function body.
- Every function has an associated return type.

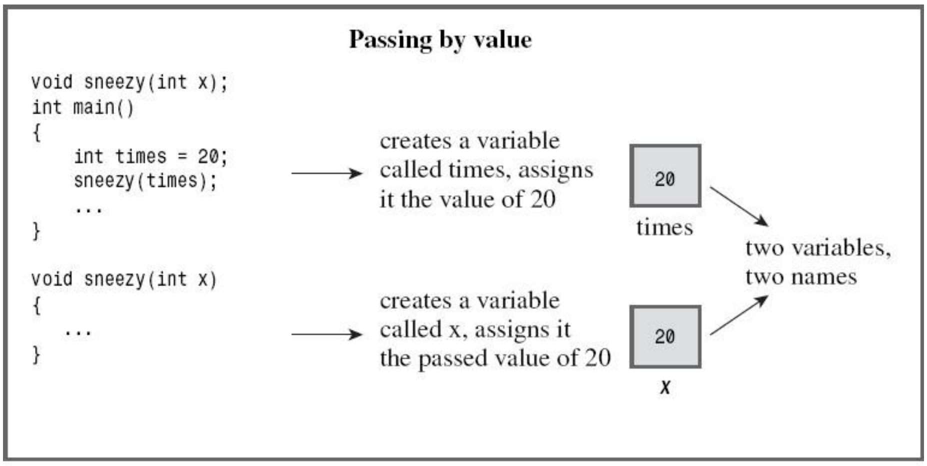
```
// get values from standard input
cout << "Enter two values: \n";
int i, j;
cin >> i >> j;
// call gcd on arguments i and j
// and print their greatest common divisor
cout << "gcd: " << gcd(i, j) << endl;
```

- We use call operator (a pair of parentheses) to invoke a function.

Functions: Argument Passing

- Parameters and passing arguments
  - Pass nonreference and reference parameters.
  - Pass const reference parameters.
  - Pass pointer and array

Pass nonreference and reference parameters

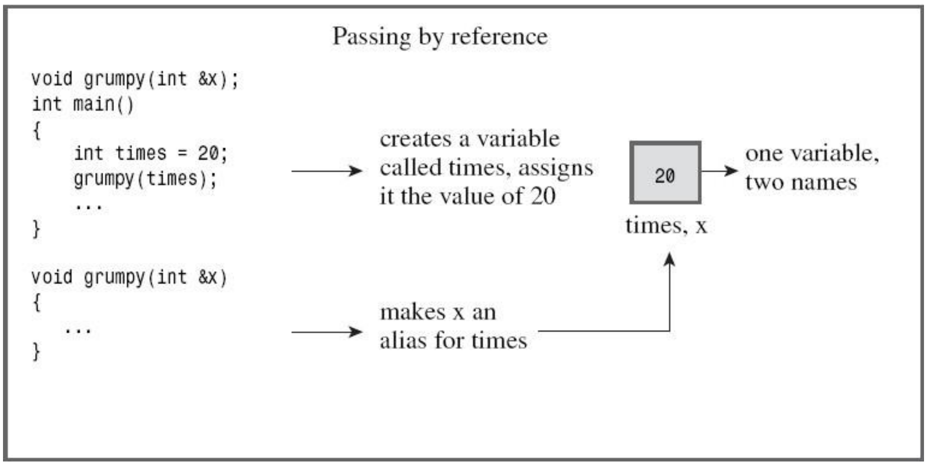


Passing by value

```
void sneezy(int x);
int main()
{
    int times = 20;
    sneezy(times);
    ...
}

void sneezy(int x)
{
   ...
}
```

creates a variable called times, assigns it the value of 20
20
times
two variables, two names
creates a variable called x, assigns it the passed value of 20
20
x



Passing by reference

```
void grumpy(int &x);
int main()
{
    int times = 20;
    grumpy(times);
    ...
}

void grumpy(int &x)
{
   ...
}
```

creates a variable called times, assigns it the value of 20
20
one variable, two names
times, x
makes x an alias for times

- We also use reference parameters when passing a large object to a function to avoid copy. For example, objects of most class types or large arrays.
- When the only reason to make a parameter a reference is to avoid copying the argument, the parameter should be const reference. (why?)

```
// compare the length of two strings
// avoid copies of strings because it could be long
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

See Note

Array and Function

- We often want to write a function to process the data in an array.
- In those cases, array is a function parameter.
- Array parameter is a very special case in C++. The array name ALWAYS be followed by **an empty bracket**.

```
void set_data(int numbs[], int size);

void get_data(const int numbs[], int size);
```

- The effect practically looks like pass-by-reference.

See Note

Functions: Return

- Every return in a function with a return type other than void must return a value.
- Return a nonreference type
  - Value returned by a function initializes a temporary (object) created at the point when the call was made.
  - Return value is copied into the temporary at the calling site
- Return a reference type
  - When a function returns a reference type, the return value is not copied. Instead, the object itself is returned.
- See note.

```
// Disaster: Function returns a reference to a local object
string &manip(const string& s)
{
     string ret = s;
     // transform ret in some way
     return ret; // Wrong: Returning reference to a local object!
}
```

-- This function will fail at run time because it returns a reference to a local object.
-- When the function ends, the storage in which ret resides is freed. The return value refers to memory that is no longer available to the program.

Never Return a Reference to a Local Object!

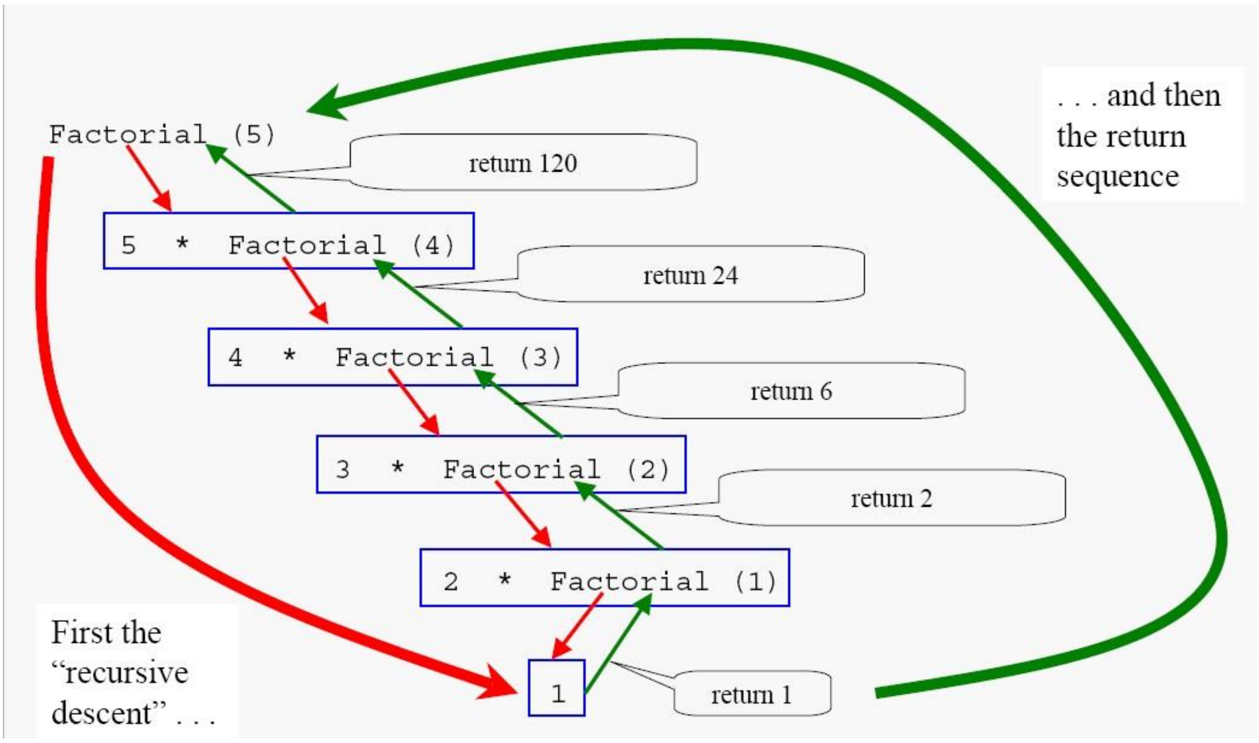
(EFC++ Item 23: Don't try to return a reference when you must return an object)

Recursive Functions

- Recursive function: a function that calls itself.
- Factorial of a number n (→ n!)

```
int Factorial (int val)
{
    if (val > 1)
        return Factorial(val-1)*val;
    return 1;
}
```

Q: when will the program stop?
Q: what happens when we invoke Factorial(5)?



```
Factorial (5)
5 * Factorial (4)
4 * Factorial (3)
3 * Factorial (2)
2 * Factorial (1)
1
```

return 120

return 24

return 6

return 2

return 1

. . . and then the return sequence

First the "recursive descent" . . .

Overloaded Functions

```
// return the greatest common divisor
int gcd(int v1, int v2)
{    while (v2) {
         int temp = v2;
         v2 = v1 % v2;
         v1 = temp;
     }
     return v1;
}
```

- A function is uniquely defined by
  - its name
  - its operand types (parameters).
- The actions of function are specified in a block, referred to as the function body.
- Every function has an associated return type.

- Functions that share the same name are said to be overloaded.
- Function overloading allows two or more functions that perform different versions of essentially the same task.

See Note

Functions with Default Arguments

```
void f() {
    print(31);
    print(31, 10);
    print(31, 16);
}
```

```
void print(int value, int base=10);
```

- Default arguments are the language facility in C++ that allow functions to have default values.

- A default argument is type checked at the time of compilation and evaluate at the time of the call. The default arguments can only be provided for tailing arguments only.

```
int f(int, int=0, int=0); //ok
int f(int=0, int=0, int); //error
int f(int=0, int, int=0); //error
int f(int, int, int=0); //ok
```

- When designing a function with default arguments, you should order the parameters so that those most likely to be used as default appear last.

See Note

Until Next Time

- HW3
- Lab4
- [Reading] Chapter 7.