C++ Basics
functions

Default Arguments

- It's common for a program to invoke a function <u>repeatedly with the same argument value for a particular parameter.</u>

- In such cases, you can specify that such a parameter has a **default argument**, i.e., a default value to be passed to that parameter.

- When a program **omits** an argument for a *parameter with a default argument* in a function call, **the compiler rewrites the function call and inserts the default value of that argument**.

- Default arguments *must* be the **rightmost** (trailing) arguments in a function's parameter list.

- Default arguments must be specified with the *first* occurrence of the function name—typically, in the function prototype

- *Using default arguments can <u>simplify</u> writing function calls. However, some programmers feel that explicitly specifying all arguments is <u>clearer</u>.*

```cpp
#include <iostream>
using namespace std;

// function prototype that specifies default arguments
unsigned int boxVolume(unsigned int length = 1, unsigned int width = 1,
   unsigned int height = 1);

int main() {
   // no arguments--use default values for all dimensions
   cout << "The default box volume is: " << boxVolume();

   // specify length; default width and height
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 1 and height 1 is: " << boxVolume(10);

   // specify length and width; default height
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 5 and height 1 is: " << boxVolume(10, 5);

   // specify all arguments
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 5 and height 2 is: " << boxVolume(10, 5, 2)
      << endl;
}

// function boxVolume calculates the volume of a box
unsigned int boxVolume(unsigned int length, unsigned int width,
   unsigned int height) {
   return length * width * height;
}
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Function Overloading

- C++ enables several functions of **the same name to be defined**, *as long as they have different signatures*. This is called **function overloading**.

- Function overloading is used to create several functions of the same name that perform *similar tasks*, but **on different data types**.

- *Overloading functions that perform closely related tasks can make programs more readable and understandable.*

Function Overloading

```cpp
#include <iostream>
using namespace std;

// function square for int values
int square(int x) {
   cout << "square of integer " << x << " is ";
   return x * x;
}

// function square for double values
double square(double y) {
   cout << "square of double " << y << " is ";
   return y * y;
}

int main() {
   cout << square(7); // calls int version
   cout << endl;
   cout << square(7.5); // calls double version
   cout << endl;
}
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Function Overloading

- Overloaded functions are *distinguished* by their **signatures**. *A signature is a combination of a function's name and its parameter types (in order)*.

- The compiler encodes each function identifier with the types of its parameters (sometimes referred to as **name mangling** or **name decoration**) to enable **type-safe linkage**.

- Type-safe linkage ensures that the proper overloaded function is called and *that the types of the arguments conform to the types of the parameters*.

Function Overloading

```cpp
// function square for int values
int square(int x) {
   return x * x;
}

// function square for double values
double square(double y) {
   return y * y;
}

// function that receives arguments of types
// int, float, char and int&
void nothing1(int a, float b, char c, int& d) { }

// function that receives arguments of types
// char, int, float& and double&
int nothing2(char a, int b, float& c, double& d) {
   return 0;
}

int main() { }
```

```
__Z6squarei
__Z6squared
__Z8nothing1ifcRi
__Z8nothing2ciRfRd
main
```

- Function-name mangling is **compiler specific**

Function Overloading

- *Overloaded functions can have different **return types**, but if they do, they must also have <u>different parameter lists.</u>*

- *Creating overloaded functions with <u>**identical** parameter lists</u> and <u>**different** return types</u> is a compilation **error**.*

- *A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error.*

```
int f(int i, double d=5) { ... }
void f(int i) { ... }

f(1,5); // calls f(int, double)
f(1); // error: Call to ‘f’ is ambiguous
```

Function Overloading in “C”

➢ *Remember that “C” language <u>does NOT support function overloading</u>!*

```
void f()
{
}

void f(int i)
{
}

int main() {
    f(); // Error: redefinition of f()

    return 0;
}
```

Function Templates

- Overloaded functions are normally used to perform *similar* operations that involve *different* program logic on different data types.

- If the program logic and operations are ***identical*** for each data type, overloading may be performed more compactly and conveniently by using **function templates.**

- You write a **single** function template definition.

- Given the argument types provided in calls to this function, C++ automatically generates separate **function template specializations** to handle each type of call appropriately.

- Defining a single function template essentially defines a whole family of overloaded functions.

Function Templates

```cpp
template <typename T>  // or template<class T>
T maximum(T value1, T value2, T value3) {
   T maximumValue{value1}; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if (value2 > maximumValue) {
      maximumValue = value2;
   }

   // determine whether value3 is greater than maximumValue
   if (value3 > maximumValue) {
      maximumValue = value3;
   }

   return maximumValue;
}
```

- All function template definitions begin with the **template keyword** followed by a **template parameter list** enclosed in angle brackets (**<** and **>**).

- Every parameter in the template parameter list is preceded by *keyword* **typename** or *keyword* **class.**

- The **type parameters** are placeholders for *fundamental types or user-defined types*.

Function Templates

```
#include <iostream>
#include "maximum.h" // include definition of function template maximum
using namespace std;

int main() {
   // demonstrate maximum with int values
   cout << "Input three integer values: ";
   int int1, int2, int3;
   cin >> int1 >> int2 >> int3;

   // invoke int version of maximum
   cout << "The maximum integer value is: "
      << maximum(int1, int2, int3);

   // demonstrate maximum with double values
   cout << "\n\nInput three double values: ";
   double double1, double2, double3;
   cin >> double1 >> double2 >> double3;

   // invoke double version of maximum
   cout << "The maximum double value is: "
      << maximum(double1, double2, double3);

   // demonstrate maximum with char values
   cout << "\n\nInput three characters: ";
   char char1, char2, char3;
   cin >> char1 >> char2 >> char3;

   // invoke char version of maximum
   cout << "The maximum character value is: "
      << maximum(char1, char2, char3) << endl;
}
```

Function Templates

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Function Templates

The *function template* ***specialization*** created for type **int** replaces each occurrence of **T** with **int** as follows:

```
int maximum(int value1, int value2, int value3) {
   int maximumValue{value1}; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if (value2 > maximumValue) {
      maximumValue = value2;
   }

   // determine whether value3 is greater than maximumValue
   if (value3 > maximumValue) {
      maximumValue = value3;
   }

   return maximumValue;
}
```

- Templates are a means of ***code generation.***

- Not placing keyword **class** or keyword **typename** before every formal type parameter of a function template (e.g., writing < class S, T > instead of < class S, class T >) is a **syntax error**.

Recursion

A **recursive function** is a function that calls **itself**, either *directly*, or *indirectly* (through another function).

- A recursive function is called to solve a problem. The function knows how to solve only the *simplest case(s)*, or so-called **base case(s)**.
- If the function is called with a base case, the function simply returns a result.
- If the function is called with a more complex problem, *it typically divides the problem into two conceptual pieces*—a piece that the function knows how to do and a piece that it does not know how to do.
- To make recursion feasible, the latter piece *must* resemble the original problem, but be a slightly simpler or smaller version.
- The function calls a **copy** of **itself** to work on the *smaller problem*—this is referred to as a **recursive call** and is also called the **recursion step**.

- **Omitting the base case** or writing the recursion step incorrectly so that it does not converge on the base case causes an infinite recursion error, typically causing a **stack overflow**. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

Recursion

In order for the recursion to eventually terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller and smaller problems must eventually *converge* on the **base case**.

Factorial:

$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$ // 0!=1 or 1!=1 is the **base case**

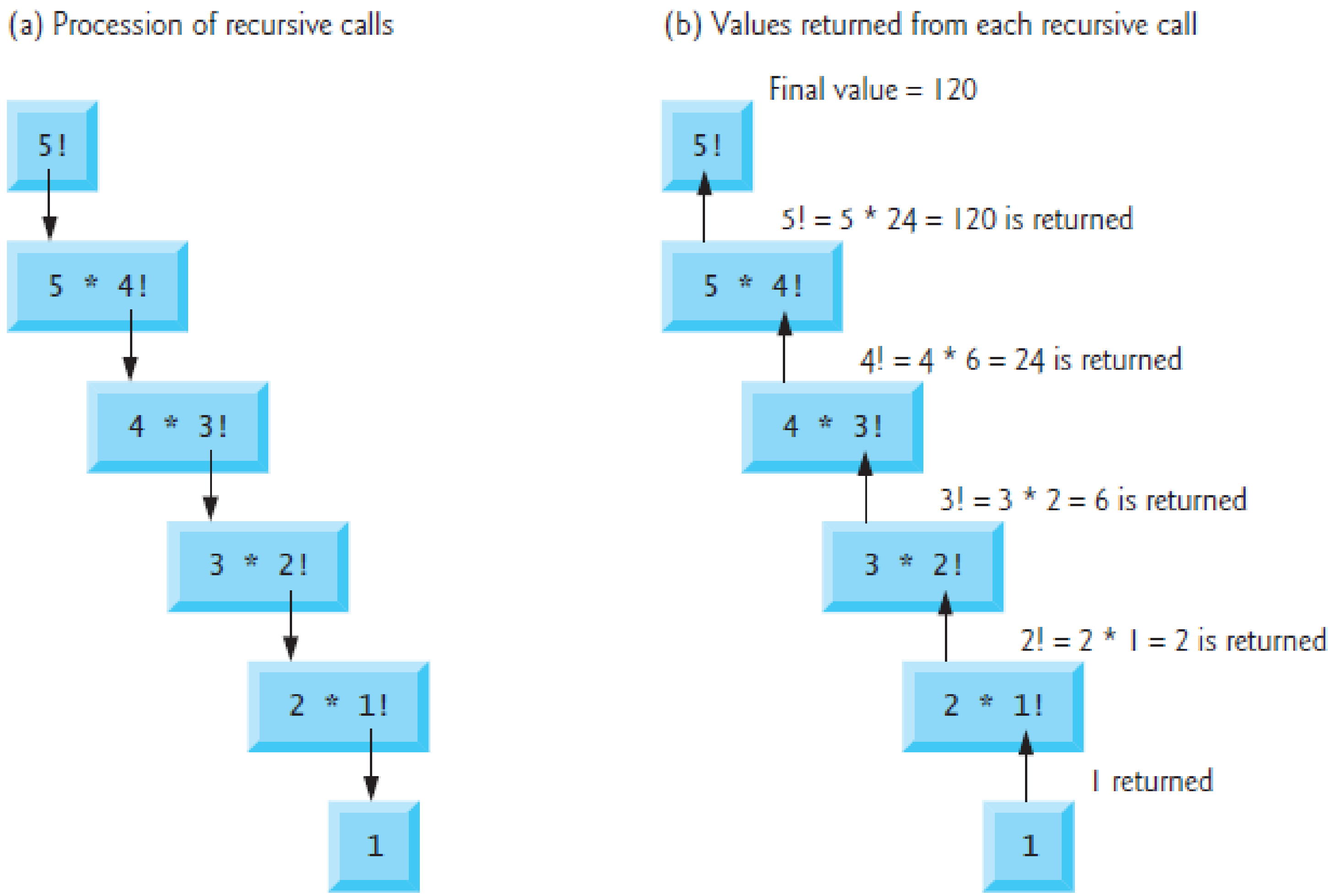
Iterative factorial:

```
factorial = 1;
for (unsigned int counter{number}; counter >= 1; --counter) {
   factorial *= counter;
}
```

Recursive factorial:

$$n! = n \cdot (n - 1)!$$

Recursion


(a) Procession of recursive calls
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
(b) Values returned from each recursive call
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

unsigned long factorial(unsigned long); // function prototype

int main() {
   // calculate the factorials of 0 through 10
   for (unsigned int counter{0}; counter <= 10; ++counter) {
      cout << setw(2) << counter << "! = " << factorial(counter)
         << endl;
   }
}

// recursive definition of function factorial
unsigned long factorial(unsigned long number) {
   if (number <= 1) { // test for base case
      return 1; // base cases: 0! = 1 and 1! = 1
   }
   else { // recursion step
      return number * factorial(number - 1);
   }
}
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Recursion example (Fibonacci series)

Fibonacci series begins with 0 and 1 and has the property that **each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers**:

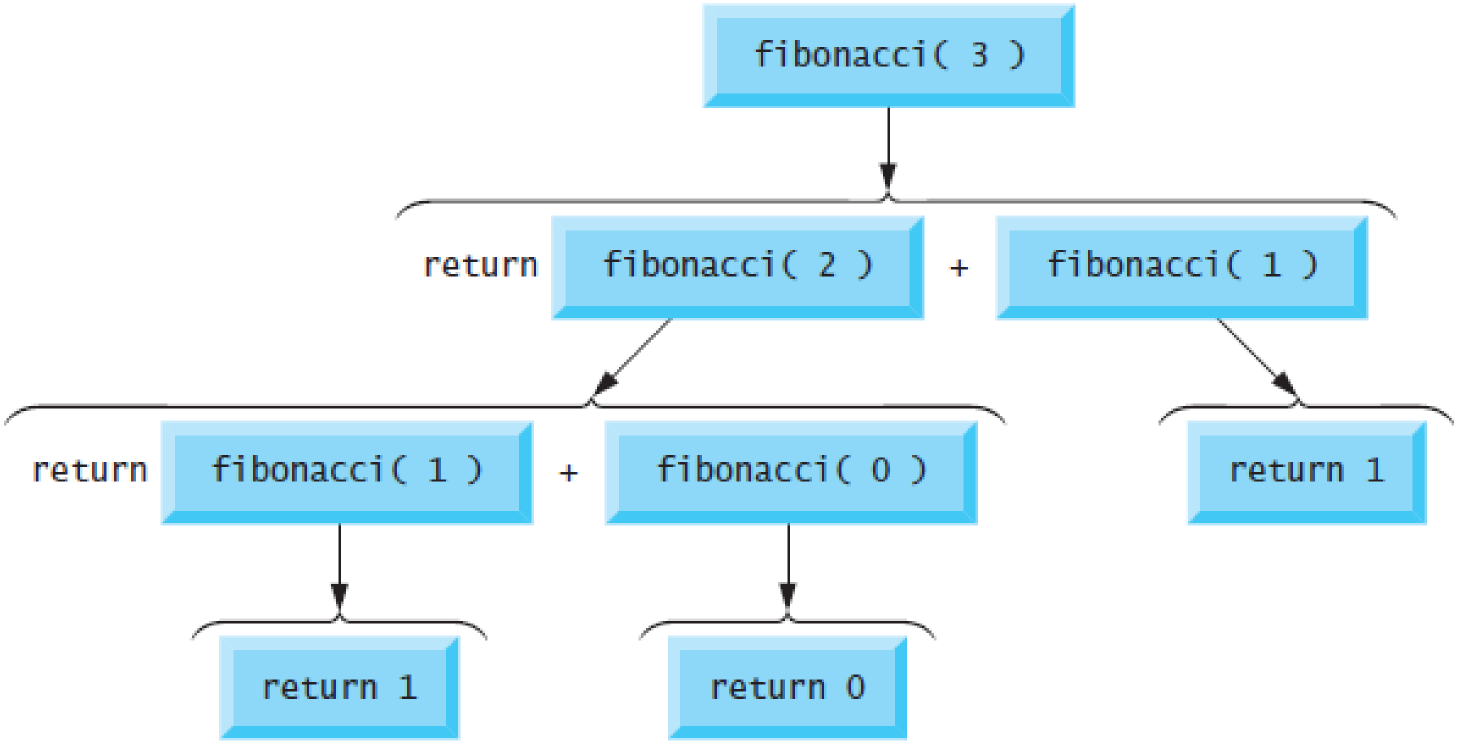
0, 1, 1, 2, 3, 5, 8, 13, 21, …

Recursive Fibonacci definition:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)
```

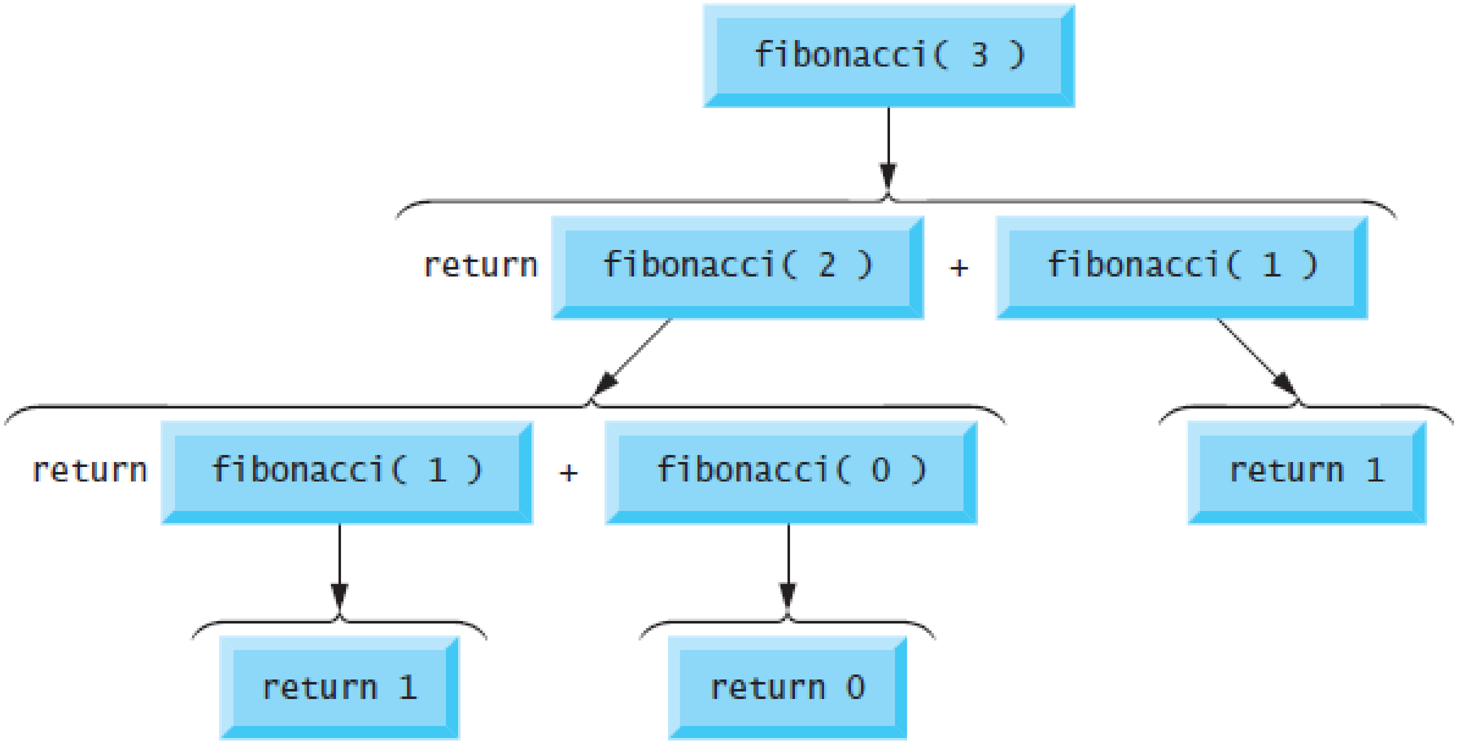
Recursion example (Fibonacci series)

In what order are these calls made?



Recursion example (Fibonacci series)

In what order are these calls made?



C++ does *not* specify the order in which the operands of most operators (including +) **are to be <u>evaluated</u>**.

Order of evaluation

C++ specifies the **order of evaluation** of the operands of **only *four* operators**:

- && (logical AND) – f1() && f2() && f3() - left to right
- || (logical OR) – f1() || f2() || f3() - left to right
- , (comma) – int a=0, b=1, c=2; - left to right

- ?: (ternary conditional) - a() ? b () : c() ? d() : e()

```
if (a()) {
    b();
} else if (c()) {
    d();
} else {
    e();
}
```

➢ But remember, that the **associativity** of the ternary operator is **right to left**!

Examples

```
int x[2] = {0,10};
int* xPtr = x;
```

```
cout << x[0] ? 5 : x[1] ? 555 : 10;
```

Is parsed as

```
cout << (x[0] ? 5 : (x[1] ? 555 : 10)); // 555
```

NOT as

```
cout << ((x[0] ? 5 : x[1]) ? 555 : 10);
```

due to right-to-left associativity.

Order of evaluation

The parts of an expression containing && or || operators are evaluated *only* until it's known whether the condition is *true* or *false:*

```
(gender == FEMALE) && (age >= 65)
```

1. stops immediately if gender *is not* equal to FEMALE
2. continues if gender *is* equal to FEMALE

- This feature of logical AND and logical OR expressions is called **short-circuit evaluation**.

- Programs that depend on the **order of evaluation** of the **operands** of operators other than **&&, ||, ?:** and the **comma (,)** operator can function differently with different *compilers* and can lead to logic errors.

Fibonacci series

```
#include <iostream>
using namespace std;

unsigned long fibonacci(unsigned long); // function prototype

int main() {
   // calculate the fibonacci values of 0 through 10
   for (unsigned int counter{0}; counter <= 10; ++counter)
      cout << "fibonacci(" << counter << ") = "
         << fibonacci(counter) << endl;

   // display higher fibonacci values
   cout << "\nfibonacci(20) = " << fibonacci(20) << endl;
   cout << "fibonacci(30) = " << fibonacci(30) << endl;
   cout << "fibonacci(35) = " << fibonacci(35) << endl;
}

// recursive function fibonacci
unsigned long fibonacci(unsigned long number) {
   if ((0 == number) || (1 == number)) { // base cases
      return number;
   }
   else { // recursion step
      return fibonacci(number - 1) + fibonacci(number - 2);
   }
}
```

Fibonacci series

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8
fibonacci(7) = 13
fibonacci(8) = 21
fibonacci(9) = 34
fibonacci(10) = 55

fibonacci(20) = 6765
fibonacci(30) = 832040
fibonacci(35) = 9227465
```

Fibonacci series

- Each level of recursion in function Fibonacci has a ***doubling*** effect on the number of function calls.

- The number of recursive calls that are required to calculate the ***n*-th** Fibonacci number is on the order of $\mathbf{2^n}$**.**

- Computer scientists refer to this as **exponential complexity**.

❖ **Avoid** Fibonacci-style recursive programs that result in an **exponential** “explosion” of calls.

Recursion vs Iteration

- Both iteration and recursion involve ***iteration.***
- Iteration and recursion each involve a ***termination test***: Iteration terminates when the *loop-continuation condition fails*; recursion terminates when a *base case is recognized*.
- Iteration ***modifies a counter*** until the counter assumes a value that makes the **loop-continuation condition fail**; recursion produces *simpler versions of the original problem* **until the base case is reached**.
- Both iteration and recursion *can occur infinitely.*

Recursion vs Iteration

Negatives of Recursion:

- Repeatedly invokes the mechanism, and consequently the *overhead, of function calls*.
- This can be expensive in both processor time and memory space.
- Each recursive call causes *another copy of the function variables* to be created.
- Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is <u>omitted.</u>

Recursion vs Iteration

Why choose Recursion?

Any problem that can be solved recursively can also be solved iteratively (non-recursively).

- A recursive approach is normally chosen when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug.
- Another reason to choose a recursive solution is that an iterative solution may not be apparent (evident) when a recursive solution is.

❖ Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

Fibonacci series

Can you write the **iterative** version of the Fibonacci series?

0, 1, 1, 2, 3, 5, 8, 13, 21, …

fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

```
// recursive function fibonacci
unsigned long fibonacci(unsigned long number) {
   if ((0 == number) || (1 == number)) { // base cases
      return number;
   }
   else { // recursion step
      return fibonacci(number - 1) + fibonacci(number - 2);
   }
}
```

Fibonacci series

Can you write the **iterative** version of the Fibonacci series?

0, 1, 1, 2, 3, 5, 8, 13, 21, …

fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

```
unsigned long fibonacci(int number) {
    if ( (0 == number) || (1 == number) ) {
        return number;
    }

    unsigned long x = 0, y = 1, res = 0;

    for (int i = 2; i <= number; i++) {
        res = x + y;
        x = y;
        y = res;
    }

    return res;
}
```

Fibonacci series

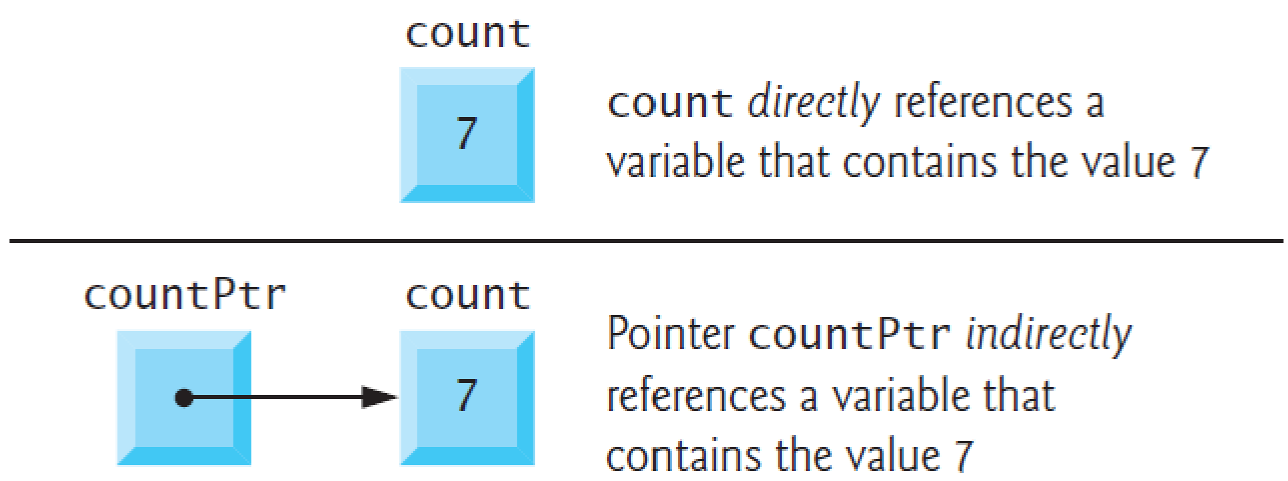
```
int main() {
    for (unsigned int counter{0}; counter <= 10; ++counter) {
        cout << "fibonacci(" << counter << ") = "
             << fibonacci(counter) << std::endl;
    }

    return 0;
}
```

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8
fibonacci(7) = 13
fibonacci(8) = 21
fibonacci(9) = 34
fibonacci(10) = 55
```

C++ Basics
Pointers

Pointers

- Pointer variables contain ***memory addresses* as their values**.
- Normally, a variable *directly* contains a *specific* value.
- A pointer contains the *memory address* of a variable that, in turn, contains a *specific* value.
- In this sense, a variable name **directly references a value**, and a pointer **indirectly references a value**.
- Referencing a value through a pointer is called **indirection**.



Pointers

Pointers, like any other variables, must be declared *before* they can be used:

int* countPtr, count;

OR

int count;
int* countPtr;

and is read as "**countPtr is a pointer to int**"

- *Assuming that the * used to declare a pointer distributes to all names in a declaration's comma-separated list of variables can lead to* ***errors.***

- *Declaring only one variable per declaration helps avoid these types of errors and improves program readability.*

- *Although it's not a requirement, we like to* ***include the letters Ptr*** *in each pointer variable name to make it clear that the variable is a pointer and must be handled accordingly.*

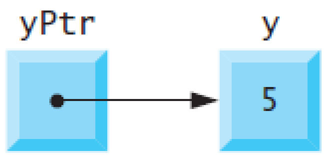
Pointers

- ➢ Pointers should be initialized to **nullptr [C++ 11].**
- ➢ Before [C++ 11] pointers were initialized to **NULL.**
- ➢ A pointer with the value nullptr "points to nothing" and is known as a **null pointer.**

❖ *Initialize all pointers to prevent pointing to unknown or uninitialized areas of memory.*

The **address operator (&)** is a unary operator that *obtains the memory address of its operand:*

int y{**5**}; // declare variable y
int* yPtr{**nullptr**}; // declare pointer variable yPtr
yPtr = **&**y; // assign address of y to yPtr



Pointers

- When declaring a reference, the & is part of the *type*.
- In an expression like &y, the & is the *address operator*

int x{1}; // declare variable x
int y{**5**}; // declare variable y
int* yPtr{**nullptr**}; // declare pointer variable yPtr

yPtr = **&**y; // assign address of y to yPtr
int& yRef = y; // declaring a reference

yPtr = &x; // assign address of x to yPtr. // OK
Int& yRef = x; // ERROR:

Pointers

The unary *** operator**—commonly referred to as the **indirection operator** or **dereferencing operator**—*returns a* lvalue *representing the object to which its pointer operand points*.

cout << *yPtr << endl; // **dereferencing** the yPtr, // OK
*yPtr = 9; // OK
cin >> *yPtr; // OK

- *Dereferencing an **uninitialized pointer results in undefined behavior** that could cause a fatal execution-time error. This could also lead to accidentally modifying important data, allowing the program to run to completion, possibly with incorrect results.*

- ***Dereferencing a null pointer results in undefined behavior** and typically causes a fatal execution-time error. **Ensure that a pointer is not null*** **before** ***dereferencing it:***

```
if (nullptr != yPtr) {
    *yPtr = 9;
}
```

Pointers

```
int a{7}; // initialize a with 7
int* aPtr = &a; // initialize aPtr with the address of int variable a

cout << "The address of a is " << &a
     << "\nThe value of aPtr is " << aPtr
     << "\nThe address of aPtr is " << &aPtr;
cout << "\n\nThe value of a is " << a
   << "\nThe value of *aPtr is " << *aPtr << endl;
```

Result:

The address of a is 0x61febc
The value of aPtr is 0x61febc
The address of aPtr is 0x61feb8

The value of a is 7
The value of *aPtr is 7

Pointers

There are three ways in C++ to pass arguments to a function:

- pass-by-value
- pass-by-reference with a <u>reference</u> argument
- **pass-by-reference with a <u>pointer</u> argument**.

Pointers

```
#include <iostream>
using namespace std;

int cubeByValue(int); // prototype

int main() {
   int number{5};

   cout << "The original value of number is " << number;
   number = cubeByValue(number); // pass number by value to cubeByValue
   cout << "\nThe new value of number is " << number << endl;
}

// calculate and return cube of integer argument
int cubeByValue(int n) {
   return n * n * n; // cube local variable n and return result
}
```

```
The original value of number is 5
The new value of number is 125
```

Pointers

Pass-By-Reference with a Pointer Actually Passes the Pointer By Value

```
#include <iostream>
using namespace std;

void cubeByReference(int*); // prototype

int main() {
   int number{5};

   cout << "The original value of number is " << number;
   cubeByReference(&number); // pass number address to cubeByReference
   cout << "\nThe new value of number is " << number << endl;
}

// calculate cube of *nPtr; modifies variable number in main
void cubeByReference(int* nPtr) {
   *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
}
```

```
The original value of number is 5
The new value of number is 125
```

```
*nPtr = (*nPtr) * (*nPtr) * (*nPtr); // cube *nPtr
```

Step 1: Before `main` calls `cubeByValue`:

```
int main() {
   int number{5};
   number = cubeByValue(number);
}
```

number
5

Step 2: After `cubeByValue` receives the call:

```
int main() {
   int number{5};
   number = cubeByValue(number);
}
```

number
5

```
int cubeByValue( int n ) {
   return n * n * n;
}
```

n
5

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:

```
int main() {
   int number{5};
   number = cubeByValue(number);
}
```

number
5

```
int cubeByValue(int n) {
   return n * n * n;
}
```

125

n
5

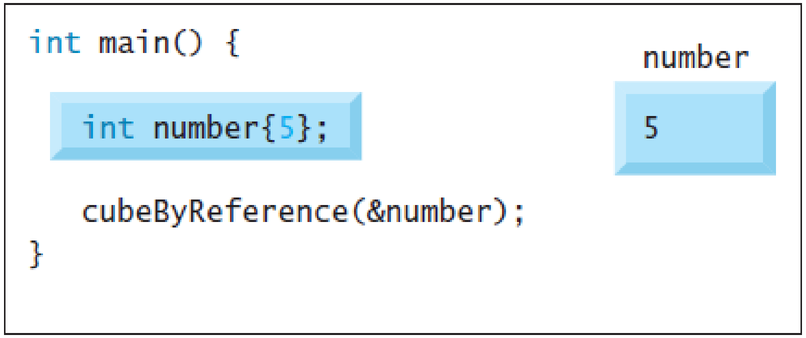
Step 4: After **cubeByValue** returns to **main** and before assigning the result to **number**:

```
int main() {                                   number
                                               5
   int number{5};           125

   number = cubeByValue(number);
}
```

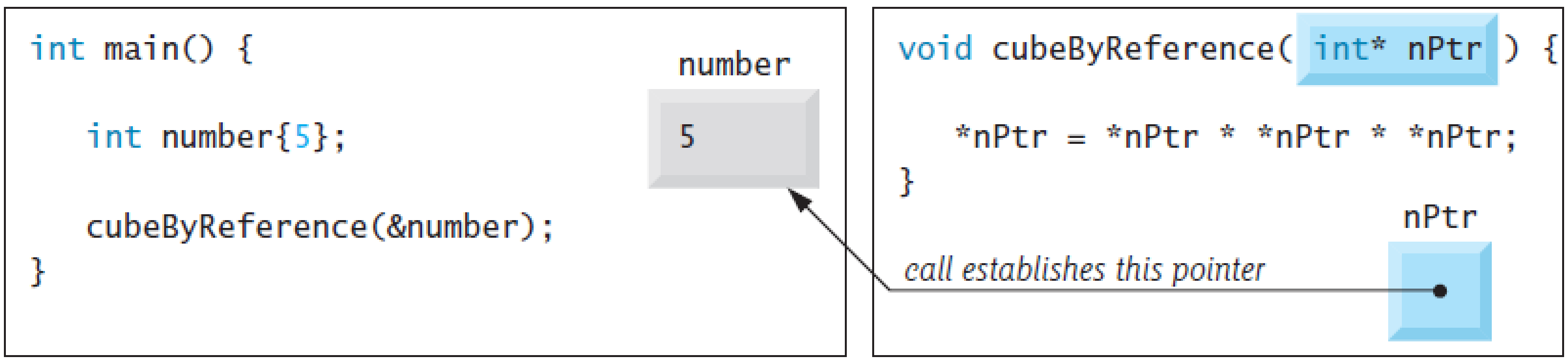
Step 5: After **main** completes the assignment to **number**:

```
int main() {                                   number
                                               125
   int number{5};
    125
   number = cubeByValue(number);
}
```

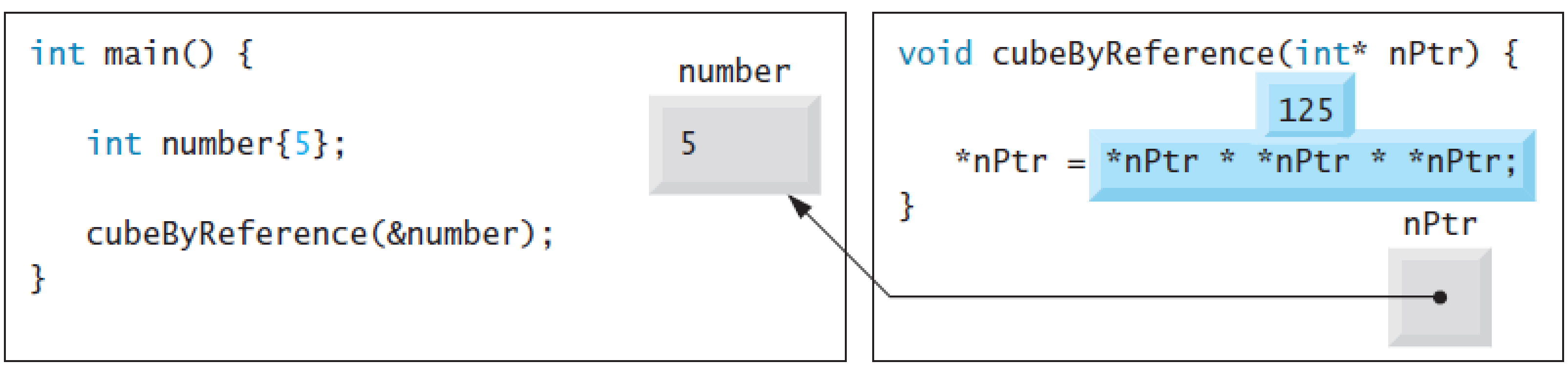
Step 1: Before `main` calls `cubeByReference`:



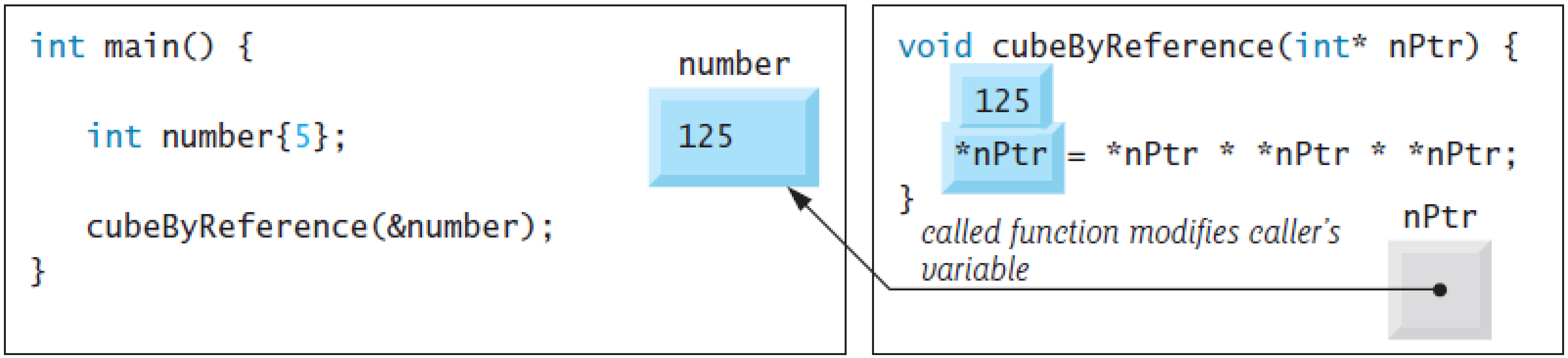
Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:



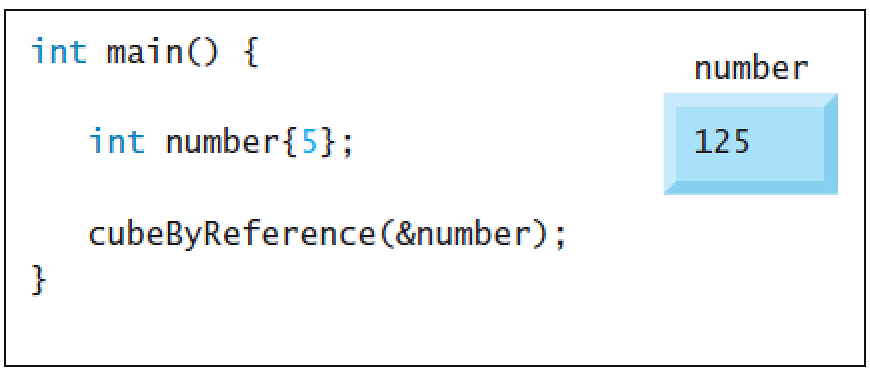
Step 3: Before `*nPtr` is assigned the result of the calculation `5 * 5 * 5`:



Step 4: After `*nPtr` is assigned 125 and before program control returns to `main`:



Step 5: After `cubeByReference` returns to `main`:



Pointer constants

There are four ways to pass a pointer to a function:

- a *nonconstant pointer to nonconstant data*,
- a *nonconstant pointer to constant data*,
- a *constant pointer to nonconstant data,*
- a *constant pointer to constant data*.

nonconstant pointer to nonconstant data

The highest access is granted by a **nonconstant pointer to nonconstant data**:

- the *data can be modified* through the dereferenced pointer, and
- the *pointer can be modified* to point to other data.

```
int* countPtr;
```

nonconstant pointer to constant data

A **nonconstant pointer to constant data** is:

• a pointer that can be modified to point to *any* data of the appropriate type, but

• the data to which it points *cannot* be modified through that pointer.

const int* countPtr;

❖ Some programmers prefer to write this as **int const* countPtr**; to make it obvious that const applies to the int, not the pointer. They'd read this declaration from right to left as "countPtr is a pointer to a constant integer."

nonconstant pointer to constant data

```cpp
void f(const int*); // prototype

int main() {
   int y{0};

   f(&y); // f will attempt an illegal modification
}

// constant variable cannot be modified through xPtr
void f(const int* xPtr) {
   *xPtr = 100; // error: cannot modify a const object
}
```

GNU C++ compiler error message:

```
fig08_10.cpp: In function 'void f(const int*)':
fig08_10.cpp:17:12: error: assignment of read-only location '* xPtr'
```

constant pointer to nonconstant data

A **constant pointer to nonconstant data** is a pointer that:

• always points to the same memory location, and

• the data at that location *can* be modified through the pointer.

```
int* const countPtr;
```

constant pointer to nonconstant data

```
int main() {
   int x, y;

   // ptr is a constant pointer to an integer that can be modified
   // through ptr, but ptr always points to the same memory location.
   int* const ptr{&x}; // const pointer must be initialized

   *ptr = 7; // allowed: *ptr is not const
   ptr = &y; // error: ptr is const; cannot assign to it a new address
}
```

Microsoft Visual C++ compiler error message:

```
'ptr': you cannot assign to a variable that is const
```

constant pointer to constant data

The *minimum* access privilege is granted by a **constant pointer to constant data**:

• such a pointer *always* points to the *same* memory location, and

• the data at that location *cannot* be modified via the pointer.

<u>const</u> int* <u>const</u> countPtr;

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

  // nonconst pointer to nonconst data
  int* xPtr1 = &a;
  *xPtr1 = 10;
  xPtr1 = tmpPtr;
  xPtr1 = tmpPtr2;
}
```

Examples

```
int main() {
   int* tmpPtr = nullptr;
   const int* tmpPtr2 = nullptr;
   int a{7}; // initialize a with 7

   // nonconst pointer to nonconst data
   int* xPtr1 = &a;
   *xPtr1 = 10; // OK
   xPtr1 = tmpPtr; // OK
   xPtr1 = tmpPtr2; // Error
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

  // nonconst pointer to const data
  const int* xPtr2 = &a;
  *xPtr2 = 10;
  xPtr2 = tmpPtr;
  xPtr2 = tmpPtr2;
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

  // nonconst pointer to const data
  const int* xPtr2 = &a;
  *xPtr2 = 10; // Error
  xPtr2 = tmpPtr; // OK
  xPtr2 = tmpPtr2; // OK
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

// const pointer to nonconst data
  int* const xPtr3 = &a;
  *xPtr3 = 10;
  xPtr3 = tmpPtr;
  xPtr3 = tmpPtr2;
}
```

Examples

```
int main() {
   int* tmpPtr = nullptr;
   const int* tmpPtr2 = nullptr;
   int a{7}; // initialize a with 7

// const pointer to nonconst data
   int* const xPtr3 = &a;
   *xPtr3 = 10; // OK
   xPtr3 = tmpPtr; // Error
   xPtr3 = tmpPtr2; // Error
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

// const pointer to const data
  const int* const xPtr4 = &a;
  *xPtr4 = 10;
  xPtr4 = tmpPtr;
  xPtr4 = tmpPtr2;
  int y = *xPtr4;
}
```

Examples

```
int main() {
  int* tmpPtr = nullptr;
  const int* tmpPtr2 = nullptr;
  int a{7}; // initialize a with 7

// const pointer to const data
  const int* const xPtr4 = &a;
  *xPtr4 = 10; // Error
  xPtr4 = tmpPtr; // Error
  xPtr4 = tmpPtr2; // Error
  int y = *xPtr4; // OK
}
```