// inlineFunction: inline Function
// sum() is NOT inline

#include <iostream>

using namespace std;

int sum(int a, int b); // Prototype

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

    x = sum(2, 3);
    cout << "x is " << x << endl;

    y = sum(4, 2 * x);
    z = sum(x, y) + 20;
    cout << "y is " << y << " and z is " << z << endl;

    cout << "sum(z, 3) is " << sum(z, 3) << endl;

    return 0;
}

int sum(int a, int b) // Definition
{
    return a + b;
}

Output:
x is 5
y is 14 and z is 39
sum(z, 3) is 42
// inlineFunction: inline Function Version 2
// sum() is inline

#include <iostream> // cout

using namespace std;

inline int sum(int a, int b) { return a + b; } // Definition

/* An inline function must be defined before its first call. The definition also serves as the prototype. In other words, the compiler can use the header when checking the actual arguments for number and type. */

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

    x = sum(2, 3);
    cout << "x is " << x << endl;

    y = sum(4, 2 * x);
    z = sum(x, y) + 20;
    cout << "y is " << y << " and z is " << z << endl;

    cout << "sum(z, 3) is " << sum(z, 3) << endl;

    return 0;
}

// inlineFunction: inline Function Version 3
// With compiler substitutions of inline function

#include <iostream> // cout

using namespace std;

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

    x = 2 + 3; // Substitutions shown in bold
    cout << "x is " << x << endl;

    y = 4 + 2 * x;
    z = x + y + 20;
    cout << "y is " << y << " and z is " << z << endl;

    cout << "sum(z, 3) is " << z + 3 << endl;

    return 0;
}

All 3 versions of this program produce the same output as shown on page 12-1.
1. Added with ANSI/ISO standard; not in C

2. Enhancement designed to speed up program execution:

Difference between *normal functions* and *inline functions* is how the compiler incorporates them into the executable program.

Program instructions have addresses just as variables have addresses. The address register stores the address of the next instruction to be executed.

When there is a transfer of control, the computer transfers to an address other than the next sequential statement.

**Overhead** of a normal function call, representing an increase in execution time:

a. store address of next sequential instruction

b. copy arguments to stack and/or set up references; allocate local variables

c. transfer control to beginning of called function

d. execute the function, storing a return value somewhere if the function is not void

e. transfer control back to caller; deallocate the stack that was used by the callee

3. **inline functions:** Compiler replaces the function call with the function code.

Requires additional compile time to save run time.

**Advantage:**
No transfer of control to and from the function.
Thus, increased execution speed.

**Disadvantage:**
Memory penalty
If a function is called 10 times, there are 10 copies of the function's code, whereas a non-inline function is stored only once.
4. Compiler does not have to honor your request to inline a function.
   Function may be too large or recursive.

5. Generally, if function cannot be coded on one line, it should not be inlined.
   Avoids "code bloat"
   Function definition serves as a prototype.

6. **inline** is a function specifier
   When prototype and definition are in same file, both need keyword **inline**.
// Default Arguments

#include <iostream>

using namespace std;

void print_string(char ch = '*', int length = 10); // Default arguments

int main()
{
    print_string(); // same as print_string('*', 10);
    print_string('+'); // same as print_string('+', 10);
    print_string('-', 5);

    return 0;
}

void print_string(char ch, int length) // No default values in header
{
    for (int i = 1; i <= length; ++i)
        cout << ch;
    cout << endl;
}

Output:
**********
++++++++++
-----

```cpp
// More default arguments

#include <iostream>
#include <cstring> // strlen()

using namespace std;

char* substring(const char* str, int n = 1); // Default argument

/* Returns a substring consisting of the n leftmost characters of the string argument. If n is out of range, the string argument is returned. n defaults to 1. */

int main()
{
  char test_string[] = "polymorphism";
  char* ps = substring(test_string);
  cout << ps << endl;
  delete [] ps;
  ps = substring(test_string, 4);
  cout << ps << endl;
  delete [] ps;
  ps = substring(test_string, 0);
  cout << ps << endl;
  delete [] ps;
  ps = substring(test_string, 20);
  cout << ps << endl;
  delete [] ps;
  return 0;
}

char* substring(const char* str, int n) // No default argument in header
{
  int length = strlen(str);
  if (n < 1 || n > length)
    n = length;
  char* pstr = new char[n + 1]; // Create space for new string
  int i;
  for (i = 0; i < n; ++i) // Copy n characters to new string
    pstr[i] = str[i];
  pstr[i] = '\0';
  return pstr;
}

Output:
poly
polymorphism
polymorphism
```
Stack during `substring()` from line 51:

```
<table>
<thead>
<tr>
<th>0</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>●</td>
<td>pstr</td>
</tr>
<tr>
<td>12</td>
<td>length</td>
</tr>
<tr>
<td>●</td>
<td>str</td>
</tr>
</tbody>
</table>
```

copy of 4

```
<table>
<thead>
<tr>
<th>4</th>
<th>n</th>
<th>substring()</th>
</tr>
</thead>
<tbody>
<tr>
<td>●</td>
<td>ps</td>
<td></td>
</tr>
</tbody>
</table>
```

main() test_string polymorphism

```c
char* substring(const char* str, int n)
{
    int length = strlen(str);
    if (n < 1 || n > length)
        n = length;
    char* pstr = new char[n + 1];
    int i;
    for (i = 0; i < n; ++i)
        pstr[i] = str[i];
    pstr[i] = '\0';
    return pstr;
}
```

**Note:**

```c
char* ps;
for (int i = 1; i <= strlen(test_string); i++)
{
    ps = substring(test_string, i);
    cout << ps << endl;
}
```

```
delete [] ps;
```

Outputs:
p
po
pol
poly
...
polymorphism
polymorphism

Problem with above code?
1. Added with ANSI/ISO standard; not in C.

2. **default argument**: value that is used automatically if the corresponding actual argument is omitted from the function call.

   Established in prototype because compiler uses prototype to check actual arguments. If an argument is missing, the compiler can supply the default.

   a)  
   ```cpp
   istream& ignore(int = 1, int = EOF); // Prototype
   ```
   ```cpp
   cin.ignore(); // same as cin.ignore(1, EOF);
   ```
   ```cpp
   cin.ignore(1024, \n'); // Discards next 1024 characters or up thru first newline, whichever comes first
   ```

   b)  
   ```cpp
   void fun(int n = 1); // default value is an initialization value
   ```
   ```cpp
   fun(); // same as fun(1);
   ```
   ```cpp
   fun(2); // Actual argument overrides the default
   ```

3. If an argument has a default value, then all arguments to its right must have defaults.

   ```cpp
   void fun1(int a = 10, int b = 20, int c = 30); // Valid
   ```
   ```cpp
   void fun2(int a, int b = 20, int c = 30); // Valid
   ```
   ```cpp
   void fun3(int a, int b = 20, int c); // Invalid
   ```

   ```cpp
   fun1(); // fun1(10, 20, 30);
   ```
   ```cpp
   fun1(2); // fun1(2, 20, 30);
   ```
   ```cpp
   fun1(2, 1); // fun1(2, 1, 30);
   ```
   ```cpp
   fun1(1, 1, 1); // No default arguments used
   ```
   ```cpp
   fun1(2, , 3); // Invalid; b is not set to 20;
   ```
   ```cpp
   // cannot skip arguments
   ```

4. Important in **class** design: To reduce the number of constructors
// OverloadedFun: Overloaded Functions
// function name, min, is overloaded

#include <iostream> // cout
#include <iomanip> // setprecision()

using namespace std;

int min();
int min(int);
int min(int, int);
int min(int, int, int);
double min(double);
double min(double, double);
double min(double, double, double);

int main()
{
    cout << fixed << setprecision(3)
    << "min() is " << min() << 'n'
    << "min(2) is " << min(2) << 'n'
    << "min(2, 5) is " << min(2, 5) << 'n'
    << "min(5, 1, 3) is " << min(5, 1, 3) << 'n'
    << "min(2.578) is " << min(2.578) << 'n'
    << "min(2.578, 5.53) is " << min(2.578, 5.53) << 'n'
    << "min(5.53, 1.223, 3.89) is " << min(5.53, 1.223, 3.89) << 'n';
    return 0;
}

int min() // Good candidates for inlining
{
    return 0;
}

int min(int a)
{
    return a;
}

int min(int a, int b)
{
    return a < b ? a : b;
}

int min(int a, int b, int c)
{ // Nested conditional expression; () are optional
    return a < b ? (a < c ? a : c) : (b < c ? b : c);
}

double min(double a)
{
    return a;
}
double min(double a, double b)
{
    return a < b ? a : b;
}

double min(double a, double b, double c)
{
    return a < b ? (a < c ? a : c) : (b < c ? b : c);
}

\textbf{Output:}

\begin{verbatim}
min() is 0
min(2) is 2
min(2, 5) is 2
min(5, 1, 3) is 1
min(2.578) is 2.578
min(2.578, 5.53) is 2.578
min(5.53, 1.223, 3.89) is 1.223
\end{verbatim}
1. \texttt{int get();} // Prototype

   char ch;
   ch = cin.get(); // Gets next character (including WS)
   // and returns it as \texttt{int}

\texttt{istream& get(char&);} // Prototype

   char ch;
   cin.get(ch); // Gets next character (including WS)
   // and assigns it to its argument

\texttt{istream& get(char*, int, char = '\n');} // Prototype

   char name[80];
   cin.get(name, 80); // Gets chars. (including WS) until it reads 79
   // or until it reaches newline.
   // Leaves \texttt{\n} in buffer

2. \textbf{Function Polymorphism} ("many forms") or \textbf{Function Overloading}: Multiple functions sharing the same name

   "function having many forms" \rightarrow \textbf{overloading the name}
   in this case \texttt{get()}

   Not in C; not new with the standard

3. Analogous to verbs with different meanings: run a marathon
   run a C++ program
   run a bath
   run a risk

   True meaning is derived from the context.

Compiler uses the function's \texttt{signature} (argument list) to pick the function; It's as if the signature is part of the name.

Can define two or more functions with the same name provided they have different signatures, that is, signatures that differ in number of arguments or in type of arguments, or both. When a function is called, the compiler matches the actual arguments to a signature.
4. If there is no matching prototype, C++ will try a standard type conversion to force a match.

```cpp
void fun(double); // fun() is not overloaded
fun(2);          // int 2 is coerce to a double

BUT
void fun1(double); // fun1() is overloaded
void fun1(float);
fun1(2);          // ERROR - ambiguous call
```

Compiler doesn't know which function to use since int 2 could be promoted to double or float.

5. int fun2(int); // fun2() is overloaded
   int fun2(int&);

```cpp
int x = 2;
fun2(x);       // x matches int and int&
```

Compiler has no way of knowing which function to use. Thus, it considers a reference to a type and the type itself to be the same in a signature. This kind of overloading should not be done.

6. Function matching process does discriminate between const & non-const.

```cpp
void fun3(int);   // Different signatures
void fun3(const int);
```
7. Overloaded functions with default arguments:

```c
void fun4();
void fun4(int a = 1, int b = 2);

fun4();                  // Ambiguous call
fun4(5, 6);               // Valid
fun4(5);                  // Valid

void fun5();
void fun5(int a, int b = 2);

fun5();                  // not Ambiguous
```

8. Return value is not part of signature.

```c
int sum(int, int);
double sum(int, int);       // Overloading will not work
```

9. Function names should be overloaded only when appropriate; that is, when functions perform the same operations on different types.

10. Been using overloading with the arithmetic operators:

```c
10.0 / 4.0 yields 2.5 using floating-point arithmetic based on operands
10 / 4 yields 2 using integer arithmetic based on operands
```