CSC 127 - Modular Programming With C++

Modular Programming

"The process of **splitting** of a **large program** into **small manageable tasks** and **designing** them **independently** is known as Modular **Programming or Divide-&-Conquer Technique.**"

C++ Functions

- **“Set of program statements** that can be processed independently.**”**

- Like in other languages, called **subroutines** or **procedures**.

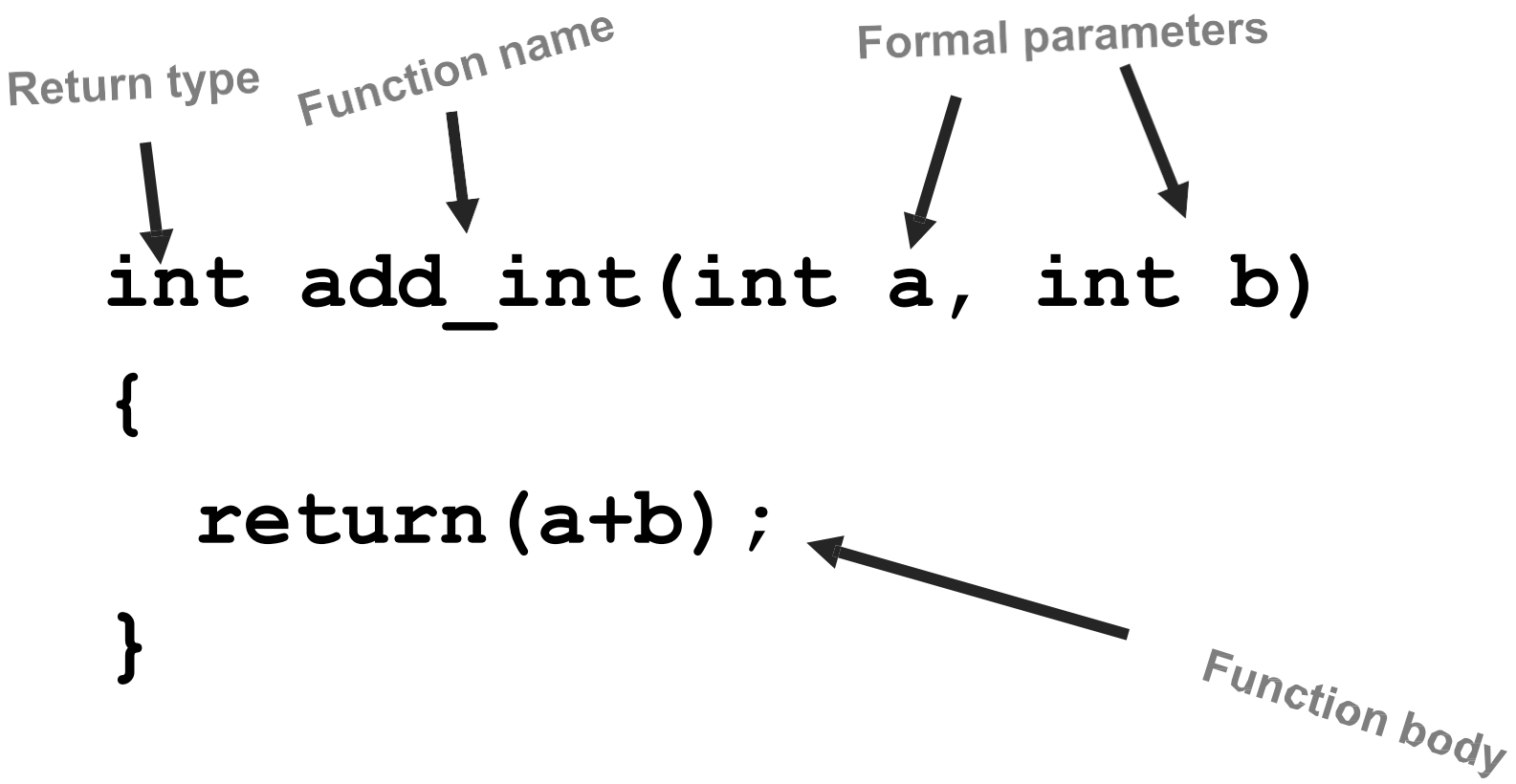
Advantages …?

- Elimination of redundant code
- Easier debugging
- Reduction in the Size of the code
- Leads to reusability of the code
- Achievement of Procedure Abstraction

Function Components

- Function Prototypes
- Function Definition(declaration & body)
- Function Parameters(formal parameters)
- return statement
- Function call(actual parameters)

Sample function



Using Math Library functions

- C++ includes a library of Math functions that we use.
- We have to know how to *call* these functions before we use them.
- We have to know what they return.
- We don’t have to know how they work!
- **`#include <math.h>`**

Math Library Functions

ceil floor

cos sin tan

exp log log10 pow

Etc.,

Function parameters

- The Formal parameters are local to the function.
  - When the function is called they will have the values *passed in*.
  - The function gets *a **copy*** of the values passed in.

Local variables

- Parameters and variables declared **inside** the definition of a function are *local*.
- They only exist inside the function body.
- Once the function returns, the variables no longer exist!

Block Variables

- We can also declare variables that exist only within the *body* of a compound statement *(a block*)*:*

```
{
int res;
    …
    …
}
```

Global variables

- We can declare variables **outside** of any function definition – these variables are ***global variables*.**
- Any function can access/change global variables.
- Example: flag that indicates whether debugging information should be printed.

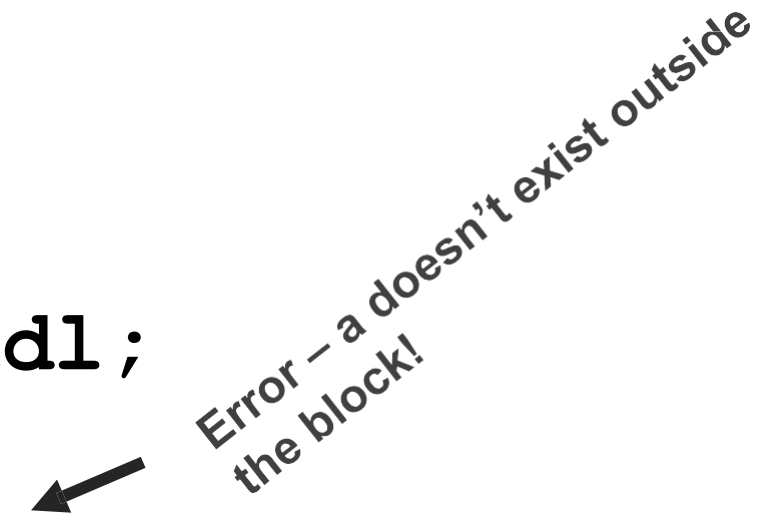
Scope

- The ***scope*** of a variable is the portion of a program where the variable has meaning (where it exists).
- A **global** variable has global (unlimited) scope.
- A **local** variable's scope is restricted to the function that declares the variable.
- A **block** variable's scope is restricted to the block in which the variable is declared.

Block Scope

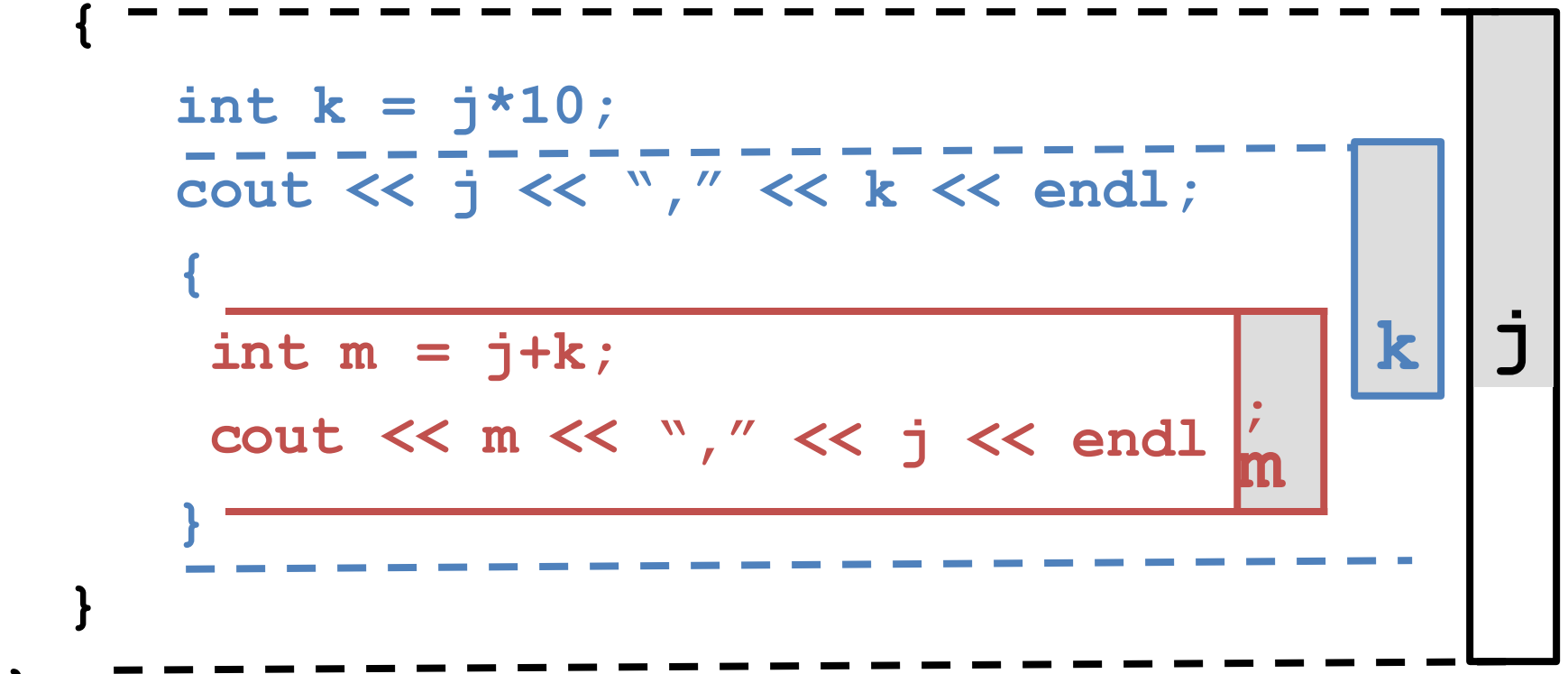
```
int main(void)
{
  int y;

  {
     int a = y;
     cout << a << endl;
  }
  cout << a << endl;
}
```



Nested Blocks

```
void example()
 {
  for (int j=0;j<10;j++)
  {
      int k = j*10;
      cout << j << ",” << k << endl;
      {
       int m = j+k;
       cout << m << ",” << j << endl;
      }
  }
}
```



Storage Class

- Each variable has a *storage class*.
  - Determines the **life time** during which the variable exists *in memory*.
  - Some variables are **created only once**
    - Global variables are created only once.
  - Some variables are **re-created** many times
    - Local variables are re-created each time a function is called.

Storage Classes

- **`auto`** – created each time the block in which they exist is *entered*.
- **`register`** – same as **`auto`**, but tells the compiler to make as fast as possible.
- **`static`** – created only once, even if it is a local variable.
- **`extern`** – global variable declared elsewhere.

Argument Passing

- Pass by Value
- Pass by Reference

Inline Functions

“ Inline functions are those whose **function body** is inserted **in place** of the **function call** statement during the compilation process.”

- **Syntax:**

  inline return_dt func_name(formal parameters)

  {

  function body

  }

Inline Functions

- **Frequently** executed interface functions.
- Expanding **function calls** inline can produce **faster** run times.
- Like the **register** specifier, **inline** is actually just a ***request****, not a* ***command,*** to the compiler.

Function Overloading

" Multiple functions to share the **same name** with **different signatures(types** or **numbers)**."

```
int myfunc(int i)
{
return i;
}
```

```
int myfunc(int i, int j)
{
return i*j;
}
```

Function Templates

- “A **generic function** defines a general set of operations that will be applied to various types of data.”

- A single general procedure can be applied to a wide range of data.

Function Templates

- **Syntax:**

  template <class *Ttype*> *ret-type func-name(parameter list)*

  {

  // body of function

  }