

FUNCTIONS (CONT.)

- ❑ Short functions, may be declared as **"inline"**. Useful only for very short functions, with number of parameters perhaps.

- ❑ Example:

```
#include <iostream.h>
inline int sum(const int a, const int b, const int c);
int main()
{
    cout << sum(1,2,3)<<endl;
}
int sum(const int a, const int b, const int c)
{ return a+b+c; }
```

- ❑ Note: "inline" is **advice** to compiler, to replace a function call with the function body itself!

Programming 1

Lecture 8 -- 1

SCOPE AND LIFETIME

- ❑ Variables come into **existence** when the program **starts** (but are uninitialized) and **go away** when the program **ends**

Programs so far are a collection of variables that can be accessed by any instruction in any part of the program

This raises problems:

- the programmer has to make and enforce all decisions over how variables are used and accessed
- the number of variables is fixed
- the programming language cannot enforce rules over the way variables are used

- ❑ We want to give variables

- A **scope** - defining which parts of a program can access a variable
- A **lifetime** - defining when a variable is created and destroyed

Programming 1

Lecture 8 -- 3

FUNCTIONS (CONT.)

So compiler will translate above example as:

```
#include <iostream.h>
int main()
{
    cout << (1+2+3)<<endl;
}
```

- ❑ Typically, a function declarations and declarations of a constants are in "header" file (extension ".h" or ".hh").
- ❑ System wide header files (also called "include" files) are in directory /usr/include. If you are looking for a specific function details, you can find it in header file there (eg. assert.h)
- ❑ Hint: C++ has defined function sizeof(variable), which returns the number of bytes variable occupies.

Programming 1

Lecture 8 -- 2

SCOPE

- ❑ Files, function bodies and compound statements define a **scope**
 - A **scope** is a region of a program text in which declarations are valid

- ❑ The **concept of locality** states that declarations are **local** to their scope

- A name cannot be used outside of its scope
- A name can only be defines ONCE in any scope (for variables)

Programming 1

Lecture 8 -- 4

FILE SCOPE

- A C++ source file defines a **file scope**
 - Splitting a program into different files restricts variables defined in the file to the file scope
- Programs have a **global scope**
 - Variables in the global scope may be accessed from **any** part of the program
 - By default any name declared in file scope is **GLOBAL**
 - The scope of a global name extends from its point of declaration to the end of the file it is declared in
 - A global name can be **declared** in many files but can only be **defined** once
- Try to avoid the use of global variables

Programming 1

Lecture 8 -- 5

NESTING SCOPES

- Scopes can be **nested**

```
{
    int x;
    int y;
    // some statements
    {
        int x; // hide outer x
        x = 1; // use the local x
        y = 2; // y still visible
    }
    // more statements
}
```
- **Note** - Names in an outer scope are accessible unless they are hidden by a local declaration of the same name

Programming 1

Lecture 8 -- 7

LOCAL SCOPE

- Names declared in a function body or compound statement are local to that scope

```
{
    int local_x;    // local to this scope
    // some statements
}
```
- The name `local_x` is valid from its point of declaration to the end of the scope - the closing bracket
- Outside of the compound statement `local_x` is not accessible

Programming 1

Lecture 8 -- 6

A PROGRAM TO DEMONSTRATE SCOPE

```
#include <iostream.h>
int var1 = 10;    // variable is global
int main
{
    int var2 = 20; // variable is local
    cout << "var1 = " << var1 << endl;           // 10
    cout << "var2 = " << var2 << endl;           // 20
    {
        int var1 = 30;    // variable is local
        var2 = 40;        // changing variable
        cout << "var1 = " << var1 << endl;       // 30
        cout << "var2 = " << var2 << endl;       // 40
    }
    cout << "var1 = " << var1 << endl;           // 10
    cout << "var2 = " << var2 << endl;           // 40
    return 0;
}
```

Programming 1

Lecture 8 -- 8

LIFETIMES OF VARIABLES

- A variable is **created** when its scope is entered during the execution of a program
- A variable is **destroyed** when its scope is left during the execution of a program

```
{ // enter scope here
  int local_x;
  // some statements using local_x
} // leave scope here
```
- When the program enters the scope local_x is created
- When the program leaves the scope local_x is destroyed
- When the program enters the scope again a **NEW** variable called local_x is created
- The **lifetime** of a global variable is the same as the lifetime of the program
- A global **scope** is created when a program is started and destroyed when the program finishes

Programming 1

Lecture 8 -- 9

EXAMPLE

```
/* File compute.h */
long computeAvg(long, long);
long computeAvg(long, long, long, long);

/* File compute.cc */
#include "compute.h"
static long maxNumber = 100;
static long minNumber = 1;
long computeAvg(long a, long b)
{ return (a + b + maxNumber + minNumber) / 4; }
long computeAvg(long a, long b, long max, long min)
{ return (a + b + max + min) / 4; }
```

Programming 1

Lecture 8 -- 11

GENERAL RULES

- Use local variables in a function for variables that are needed only in that function.
 - names of local variables must be unique within a function
- Use arguments/parameters for data that must be shared between a small number of functions.
 - parameter names in a function must be different from the names of local variables in that function
- Use global variables for data that must be shared between many functions.
 - names of global variables must be unique across all functions
- If a function uses the same name for a local variable or parameter as the name of a global variable, the global variable is not accessible in that function.

Programming 1

Lecture 8 -- 10

EXAMPLE (CONT.)

```
/* File main.cc */
#include <iostream.h>
#include "compute.h"
int main()
{
  long one = 10, two = 20, three = 30, four = 40;
  cout << "Function with 2 arguments " << computeAvg(one, two) << endl;
  cout << "Function with 4 arguments " <<
    computeAvg(one, two, three, four) << endl;
}
We can compile above example like this:
g++ main.cc compute.cc -o result
or
tutg++ main.cc compute.cc -o result
```

Programming 1

Lecture 8 -- 12