Introduction to C, C++, and Unix/Linux

CS 60

Lecture 14 Miscellaneous C++

Today

→ C++ operators, namespace

→ Reading [KR] Chapters 1-7

→ Read [So] chapters 1, 3, 4 (Boolean), 9, 13, 14 & 18.

Notes

- Questions

Constant declarations

- We've already seen **`const`** for declaring variables
  - It is illegal (warning) to change a **`const`** variable
- Why use **`const`** rather than **`#define`** ?
  - Normal variable scoping rules
  - Type checking (is that number an integer or a short?)
  - Can be used for almost any type of C++ construct (e.g., a **`const struct`**)
- **`#define`** is still useful, especially for conditional compilation (and, of course, macro definitions)

Constant parameters and return values

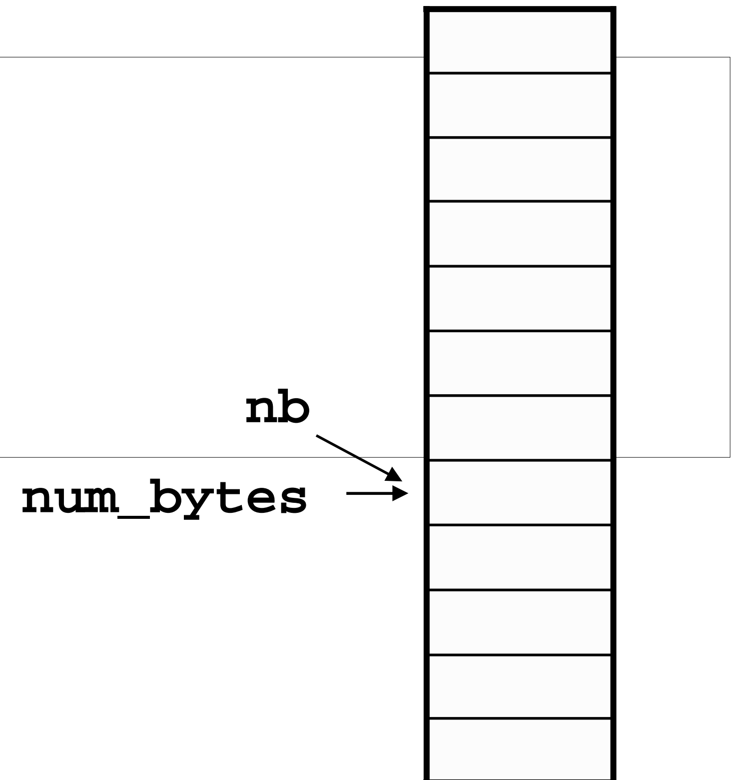
- Functions can take **`const`** parameters and return **`const`** values

```
const double Area(const double radius)
{
    double area = PI*radius*radius;
    return(area);
}
```

Reference declarations

- Another special variable type: the reference type
- A reference variable uses **&** to define another name for (an alias for) a variable

```
int num_bytes = 1024;
int& nb = num_bytes;
```



Not all that useful for regular variables

Function parameter passing

Pass-by-reference

```
void ClearIm(Image& x)
{
  x.data = 0;
}
```

Pass-by-value

```
void ClearIm(Image x)
{
  x.data = 0;
}
```

Pass-by-pointer

```
void ClearIm(Image *x)
{
  x->data = 0;
}
```

→

```
(*x).data = 0;
```

Differences?

Reference parameters and return values

- Reference parameters are used in a function parameter list to accomplish pass-by-reference (rather than pass-by-value)
  - But without the mess of having to do everything with pointers

```
void swap(int& x, int& y)
{
  int temp = x;
  x = y;
  y = temp;
}
```

Reference parameters and return values (cont.)

- Reference declarations can also be used for return values

```
int& smaller(int& x, int& y)
{
  if (x < y) return(x);
  else return(y);
}

int x = 5, y = 7;
smaller(x, y) = 0;
```

Reference parameters and return values (cont.)

- Reference declarations are efficient – just like passing a pointer rather than all the data
- If you want the efficiency but don't want the return value modified, then:

```
const int& smaller(int& x, int& y);

int x = 5, y = 7;
smaller(x, y) = 0;   ← Illegal
```

Function parameter types (Table 9.2)

Type	**Declaration**
Call by value	func(int var)
Call by address	func(int *var)
Constant call by value	func(const int var)
Reference	func(int& var)
Constant reference	func(const int& var)
Array	func(array[])

Function return types

Type	Declaration
Value	int func( )
Address	int *func( )
Constant address	const int *func( )
Constant value	const int func( )
Reference	int& func( )
Constant reference	const int& func( )

Function overloading

- In C, it’s an error to define two functions of the same name. // Except when static on diff files.
- In C++ it’s the norm!
- Multiple functions can be defined that operate on different types of data or accept different parameters
- By convention, all functions with the same name should perform the same basic function

```
int sqrt(int x);
float sqrt(float x);
double sqrt(double x);
long double sqrt(long double x);
```

```
void print(int);
void print(char);
void print(char *);
void print(float);
void print(double);
```

Function overloading (cont.)

- Important: Only the parameter list differentiates functions, ***not the return type***

```
void print(int);
void print(char);
void print(char *);
void print(float);
void print(double);
```

OK

```
char rand();
short rand();
int rand();
float rand();
double rand();
```

NO!

Default arguments

- Function parameters can be given default arguments that are used if not overridden

```
void SetPixel(Image im, int val=0);
int Combine(int x, int y, int z=0);
bool Reserve(Rest r, int Time=1930, int Num=2);
void Draw(int color=1, int size=10,
  int width=2);
```

C++ default arguments

- Function parameters can have default values

```
void Draw(int color, int size=1, int width=2);
```

```
Draw(2, 1, 2);
Draw(2, 1);
Draw(2);
```

These are equivalent calls

It's not possible to specify **`width`** without specifying **`color`** and **`size`**

C++ default arguments (cont.)

- What's wrong with this?

```
void Draw(int color=1, int size, int width=2);
```

All the parameters with defaults must be to the right.
The compiler assumes that the N parameters you specify are the *first N* parameters

```
({ args w/out defaults },{ args w/defaults })
```

Inline functions

- There is an overhead cost to call a function
  - Copy parameters to the stack, jump to the function, pull parameters off of the stack, copy results to the stack, jump back to the calling routine, pull results off of the stack...
- For efficiency, we can request the compiler to "inline" some functions – put the entire body of the function in the code instead of generating a call to the function
  - Typically used for small functions that are called frequently

Inline functions (cont.)

- For example, a function to access or modify image pixel values
  - Called thousands of times per image

```
inline void Product(double &p, double fac)
{
  p *= fac;
  return;
}
```

The C++ **`new`** and **`delete`** operators

- C++ uses the **`new`** operator to allocate heap memory, rather than **`malloc`**
- And uses **`delete`** to return the memory, rather than **`free`**
- There are really four operators:
  `new, delete, new[], delete[]`
- **`new`** and **`delete`** are part of the language, not library functions (as are **`malloc`** and **`free`**)

```
int *num;
num = new int;

int *list;
list = new int[100];

struct CourseInfo *info;
info = new CourseInfo;

class Player *players;
players = new Player[15];
```

```
delete num;

delete[] list;

delete info;

delete[] players;
```

New/delete and malloc/free

- You can use malloc and free in C++, but don't mix them with new and delete

```
int *arr = (int *)malloc(40);
delete arr;
```

```
int *val = new int;
free(val);
```

Illegal – though the compiler won't catch this!

Namespaces

- A namespace is a way to group variables and functions together under a common label
- Rather than
  - **`mylib_var1, mylib_funct2, mylib_funct3`**

  use
  - **`mylib::var1, mylib::funct2, mylib::funct3`**
- Or
  - **`using namespace mylib`**
  - **`var1, function2, function3`**

Namespaces (cont.)

```
namespace mylib {
  // declare variables
  int val;
  // declare function
  int print(void);
}
```

```
mylib::val = 3;
mylib::print();
```

```
using namespace mylib;
val = 3;
print();
```

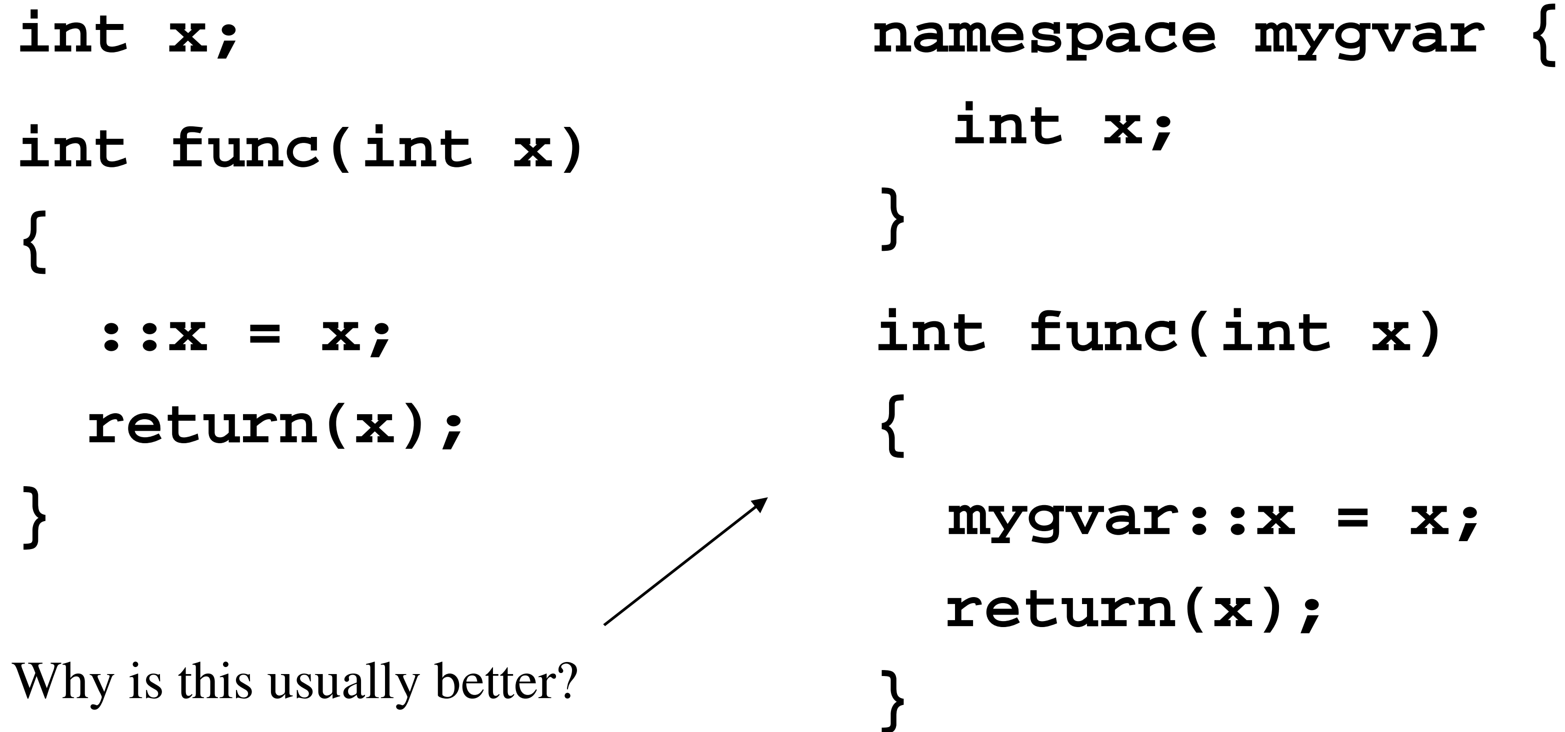
Nesting

```
namespace project {
  namespace group1 {
    namespace module {
      int version;
    }
  }
  namespace group2 {
    namespace module {
      int version;
    }
  }
}
```

```
project::group2::module::version = 3;
```

Global namespace

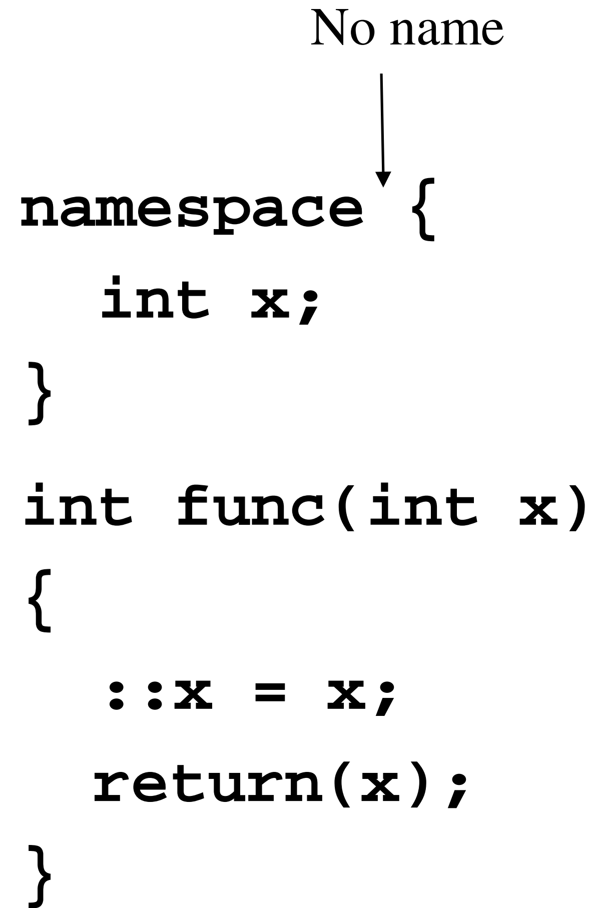
- To access a global variable, use **`::var`**



File-specific namespace

```
namespace mygvar {
  int x;
}

int func(int x)
{
  mygvar::x = x;
  return(x);
}
```



In this case, the namespace is unique to the file. So the global variable x cannot be seen outside the file.