CIS 330
C/C++ and Unix

Lecture 11

Constants

Luks Competition

he 24th Annual UO Eugene Luks Programming Contest will be held on **Saturday, May 14, 12-4pm**.

The CIS Department holds this contest every year as an opportunity to have fun while challenging your programming skills. There are two divisions of teams: undergraduate and graduate. **Undergrad teams may have up to three team members**, and grad teams may have two. The contest will be held in **Deschutes room 100** and each team will be assigned to a Mac workstation running PC^2 contest software (this may be updated to allow you to use a laptop). There will be five to six programming problems, and the goal is for each team to program correct solutions for as many of the problems as they can during the three hour contest. Programming can be done in Java, C++, Python or any other language available on the department server.

In each division, the team solving the most problems will be the division winner, with ties broken by submission times. A further description of the contest rules can be found at https://www.cs.uoregon.edu/Activities/Luks_Programming_Contest/rules.php

Previous problem sets can be found at https://www.cs.uoregon.edu/Activities/Luks_Programming_Contest/

In order to compete in the contest, you must register your team - to register, send the name and email to Arnita Albertson (arnita@uoregon.edu) or to Chris Wilson (cwilson@cs.uoregon.edu).

Previously

Initialization and cleanup - constructors & destructors

Continuing on initialization…

Aggregate Initialization for Structs

Because C-style struct has all its member as public (by default), they can be assigned directly

```
struct X {
  int i;
  float f;
  char c;
};
X x1 = { 1, 2.2, 'c' };
```

Aggregate Initialization for Structs

Or if you have an array of such structs

```
struct X {
  int i;
  float f;
  char c;
};
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
// What about x2[2]?
```

Aggregate Initialization for Structs

Or if you have an array of such structs

```
struct X {
  int i;
  float f;
  char c;
};
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
// x2[2] will be initialized with 0s
```

Aggregate Initialization in C++

If you have constructors, whether members are **private or public**, all initializations **must** go through the **constructor**

```
struct Y {
  float f;
  int i;
  Y(int a);
};
Y y1[] = { Y(1), Y(2), Y(3) };  // each will call a constructor
```

Default Constructors

```
struct Y {
  float f;
  int i;
  Y(int a) { i = a; };
};
Y y2[2] = { Y(1) }; // what would happen here?
```

Default Constructors

```
struct Y {
  float f;
  int i;
  Y(int a) { i = a; };
};
Y y2[2] = { Y(1) }; //  compiler will complain that there is no
                    // default constructor
```

Default Constructors

Default constructors are so important that the compiler will create one for you automatically, IF AND ONLY IF, no constructor has been specified AT ALL

```
class V {
  int i;  // private by default
}; // No constructor

int main() {
  V v, v2[10]; // default constructor will be used
}
```

Function Overloading

Name decoration - functions

```
void f();

class X { void f(); };
```

do not clash because compiler manufactures different names internally

This is also known as "**name mangling**"

Similarly, functions with the same name, but different parameters can be defined - we saw this in function overloading

Unions

Unions can also have member functions and access control (including constructors and destructors)

In the below example, `int i` and `float f` are stored in the s**ame memory location**

```
union U {
private:
  int i;
  float f;
public:
  U(int a);
  U(float b);
  ~U();
  int read_int();
  float read_float();
};
```

Function Overloading

We can use function overloading to "automatically" call the correct function to initialize the union members (instead of using conditionals)

```
U::U(int a) { i = a; }
U::U(float b) { f = b;}

U::~U() { cout << "U::~U()\n"; }

int U::read_int() { return i; }
float U::read_float() { return f; }

int main() {
  U X(12), Y(1.9F);
  cout << X.read_int() << endl;
  cout << Y.read_float() << endl;
}
```

Function Overloading

However, there is still no way to prevent the user from accessing the "wrong" member (e.g., `X.read_int() -> X.read_float()`)

Can we fix this?

Yes, encapsulate the union in a class

Function Overloading

```
class SuperVar {
  enum {
    character,
    integer,
    floating_point
  } vartype;
  union {  // Is this correct?
    char c;
    int i;
    float f;
  };
public:
  SuperVar(char ch);
  SuperVar(int ii);
  SuperVar(float ff);
  void print();
};
```

Function Overloading

```
class SuperVar {
  enum {
    character,
    integer,
    floating_point
  } vartype;
  union {  // Anonymous union - no type or variable name
    char c;
    int i;
    float f;
  };
public:
  SuperVar(char ch);
  SuperVar(int ii);
  SuperVar(float ff);
  void print();
};
```

Anonymous Union

Anonymous unions have no type or variable name.

Remember that normally,

```
union num_union_t {
    int i;
    float j;
};
num_union_t x; x.i = 100;
```

However, they can be declared anonymous, in which case the members can be accessed directly

```
int main() {
  union {
    int i;
    float f;
  };
  i = 12;
  f = 1.22; // i and f are still in the same memory location
}
```

Function Overloading

```
SuperVar::SuperVar(char ch) {
  vartype = character;  c = ch;
}
SuperVar::SuperVar(int ii) {
  vartype = integer;  i = ii;
}
SuperVar::SuperVar(float ff) {
  vartype = floating_point;  f = ff;
}
```

```
void SuperVar::print() {
  switch (vartype) {
    case character:
      cout << "character: " << c
           << endl;
      break;
    case integer:
      cout << "integer: " << i
           << endl;
      break;
    case floating_point:
      cout << "float: " << f
           << endl;
      break;
  }
}
```

Overloading

```
Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
  size = sz;
  quantity = initQuantity;
  next = 0;
  storage = 0;
}
```

`Stash(int)` is a special case of `Stash(int, int)`. Can we make this more compact?

Overloading With Default Arguments

```
Stash::Stash(int sz);

Stash::Stash(int sz, int initQuantity);
```

Can be replaced with:

```
Stash::Stash(int sz, int initQuantity = 0); // default
argument
```

With **default arguments,**

- Only the trailing arguments may be defaulted (why?)
- Default arguments are typically only placed in the **declaration** (e.g., in the header file)
  - Sometimes, you will see commented default value in the function definition

```
void fn(int x /* = 0 */) { // ...
```

- *If you have a function declaration (with the default argument), you **cannot** have it on the function definition*

Overloading With Default Arguments

```
void f(int i, int j = 100);
int main()
{
    f(10);
    f(10, 1000);
}

void f(int i, int j)
{
    cout << "i is " << i << endl;
    cout << "j is " << j << endl;
}
```

OKAY

```
void f(int i, int j = 100);
int main()
{
    f(10);
    f(10, 1000);
}

void f(int i, int j = 100)
{
    cout << "i is " << i <<
endl;
    cout << "j is " << j <<
endl;
}
```

NOT OKAY

Overloading With Default Arguments

```
void f(int i, int j);
int main()
{
    f(10);
    f(10, 1000);
}

void f(int i, int j = 100)
{
    cout << "i is " << i << endl;
    cout << "j is " << j << endl;
}
```

OKAY or NOT OKAY?

```
void f(int i, int j = 100)
{
    cout << "i is " << i << endl;
    cout << "j is " << j << endl;
}

int main()
{
    f(10);
    f(10, 1000);
}
```

OKAY or NOT OKAY?

Placeholder Arguments

In C++, arguments can be declared without identifiers

```
void f(int x, int = 0, float = 1.1);
```

You also don't need them for function definitions

```
void f(int x, int, float flt) { /* ... */ }
```

Function calls must still provide values for these arguments, even if the function does not use them.

This is typically done in case those are needed later, without changing all the code that calls this function.

- You could still have a name for these arguments and not use it, but the compiler will give you a warning - this method suppresses those warnings

Questions

Any questions about initialization?

const Keyword

First motivation for using const is to eliminate the use of #define macros (value substitution)

It is now used for pointers, function arguments, return types, class objects, and member functions

We will talk about how `const` should be used in C++ to maintain good **coding style** and **safety**

Value Substitution

Remember, #define is an exact textual replacement wherever it is used

- It can lead to mistakes if not used carefully

```
#define CircleArea(r) (3.14 * r * r)
```

- There is also no concept of type checking

```
double CircleArea(double r) { return 3.14 * r * r;
}
```

- This can hide bugs and make them difficult to find

Value Substitution

Using `const` brings value substitution into the domain of the compiler

```
#define bufsize 100 -> const int bufsize = 100;
```

Now the compiler knows the value at compile time, and can perform optimizations like "*constant folding*" (i.e., calculates the constant expression at compile time, and *folds* it into the code, instead of calculating them at runtime)

You can use const for all built-in types and their variants (e.g., class objects)

`const` must be **initialized when defined** (there are exceptions).

Questions?

We will take a slight detour and talk about something different (but related)

Linkage

Two types of linkage - *internal* and *external*

If a variable has **internal** linkage,

- Variable is visible only to the file it was defined in
- **Storage** is (usually) created to represent the identifier only **for the file** being compiled
- Other files may use the same identifier name (also with internal linkage) without conflict (because separate storage is created for each)
- Internal linkage is specified by the keyword `static`

If a variable has **external** linkage

- Single piece of storage is created (once) that represents the identifier for **ALL files** being compiled
- Global variables have external linkage by default
  - But to access them from outside the declared file, it must be specified as `extern`

Linkage Example

main.cc

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

int globe;
int func();

int main()
{
    globe = 100;
    func();
    cout << globe << endl;
    return 0;
}
```

func.cc

```
#include <iostream>
int globe;

int func()
{
    globe = 10;
    return globe;
}
```

```
/usr/bin/ld: /tmp/ccXTLKo9.o:(.bss+0x0): multiple definition of `globe';
/tmp/ccq9rqm2.o:(.bss+0x0): first defined here
collect2: error: ld returned 1 exit status
```

External Linkage

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

int globe;
int func();

int main()
{
    globe = 100;
    func();
    cout << globe << endl;
    return 0;
}
```

```
#include <iostream>
extern int globe;
int func()
{
    globe = 10;
    return globe;
}
```

External Linkage

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

extern int globe;
int func();

int main()
{
    globe = 100;
    func();
    cout << globe << endl;
    return 0;
}
```

```
#include <iostream>
int globe;
int func()
{
    globe = 10;
    return globe;
}
```

Linkage Example

main.cc

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

static int globe;
int func();

int main()
{
    globe = 100;
    func();
    cout << globe << endl;
    return 0;
}
```

```
100
```

func.cc

```
#include <iostream>
int globe;

int func()
{
    globe = 10;
    return globe;
}
```

External Linkage (in C)

```c
#include <stdio.h>

int globe;
int func();

int main()
{
    globe = 100;
    func();
    printf("%d\n", globe);
    return 0;
}
```

```c
#include <stdio.h>

int globe;

int func()
{
    globe = 10;
    return globe;
}
```

const in Header Files

Since #define can be used in header files, `const` must also be usable in header files (i.e., global variable).

`const` **defaults** to internal linkage (i.e., only visible within the file it was defined in and cannot be seen at link time by other units), since it tries to avoid allocating memory (otherwise, constant folding becomes difficult)

So, define it as `extern` if you want to give it external linkage

- `extern const int bufsize;`
- this forces the `const` variable to have storage (so other files can see it)

You must always initialize a `const` when you define it, except when it is made `extern`

- Since memory is allocated, you can change it whenever you want to

const variable as external

main.h

```
extern const int globe;
```

main.cc

```
#include <iostream>
#include "main.h"

using namespace std;

void func();
void test();

const int globe = 100;
int main()
{
  cout << globe << endl;
  test();
  func();
  return 0;
}
```

func.cc

```
#include <iostream>
#include "main.h"

void func()
{
  std::cout << globe << std::endl;
}
```

test.cc

```
#include <iostream>
#include "main.h"

void test()
{
  std::cout << globe << std::endl;
}
```

```
./a.out
100
100
100
```

const variable as internal

main.h

```
extern const int globe;
```

main.cc

```
#include <iostream>
#include "main.h"

using namespace std;

void func();
void test();

const int globe = 100;
int main()
{
  cout << globe << endl;
  test();
  func();
  return 0;
}
```

func.cc

```
#include <iostream>
#include "main.h"

const int globe = 10;
void func()
{
  std::cout << globe << std::endl;
}
```

test.cc

```
#include <iostream>
#include "main.h"

const int globe = 1000;
void test()
{
  std::cout << globe << std::endl;
}
```

```
./a.out
100
1000
10
```

const in Header Files

Normally, C++ **avoids** allocating storage for `const` but when you use `extern` with `const` you force C++ to allocate storage

- `extern` suggest external linkage, so for other units to be able to refer to this item, it needs some sort of storage

Storage is also allocated when `const` is used inside **complicated** structures

When storage is allocated, constant folding is prevented - because the compiler does not know **exactly** what would be in that storage location

Summary

`const` variables TYPICALLY do not require storage

- They must be initialized at definition
- They are constant folded into the code
- They are internal to the file
- They can be treated as `#define`

However, they can have storage

- For example, when they are passed in as reference to a function, or if they are part of a more complicated data structure, or used as an `extern` global variable
- In this case, constant folding is prevented, and it acts more like "**a variable whose value should not be changed**"

`const` in global variables (e.g., header files)

- Regular global variables are `extern` by default
- Thus, `const` must declare as `extern` explicitly if you want other files to see it
- This forces C++ to give it storage
- Otherwise, you can have **multiple copies** of const among different files

Safety

Another reason for using `const` is for safety

If you initialize a variable with a value produced at runtime and you know it will not change (or should not change), for the lifetime of the variable, it is a good idea to make it `const`

- Compiler will give an error message, if you try to change it
- Makes it more difficult to introduce bugs

For example,

```
const int i = 100;  // Typical constant
const int j = i + 10; // Value from const expr
char buf[j + 10]; // Still a const expression

int main() {
  cout << "type a character & CR:";
  const char c = cin.get(); // Can't change after this
  const char cx;
  cx = cin.get(); // Not allowed
  const char c2 = c + 1; // Allowed
  c = c + 1; // Not allowed
  cout << c2;
  // ... }
```

Questions?