CIS 330
C/C++ and Unix

Lecture 12

Constants contd.

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

Aggregates

```
// Constants and aggregates
const int i[] = { 1, 2, 3, 4 };
//! float f[i[3]]; // Illegal
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
//! double d[s[1].j]; // Illegal
int main() {//...}
```

Aggregates are complicated, so storage will be allocated (therefore, value is **not known at compile time**, and cannot be used then).

If these were in the main() function, it would actually be allowed, as the code in the main function are assumed to be known only at **runtime**

- It's only global variables that this matters - because variables are 'defined' but no code is 'executed.'

Differences with C

In C, `const` always occupies storage and NOT a compile time constant

Because storage is allowed, it's okay to say

```
const int bufsize;
```

`const` defaults to external linkage in C (vs. internal in C++)

Pointers and const

Pointers can be made `const` as well

- The compiler will try to prevent storage allocation (because its value can’t be changed, new address cannot be assigned to it)
- The compiler will also try to do constant folding

When using pointers, you have two options

`const` is applied to what the pointer is pointing to, or

`const` is applied to the address stored in the pointer

Pointer to const

As with pointers, read it at the identifier and work your way out

- const binds to the thing it is closest to
- `const int* u;`
  - `u` is a pointer to a constant int
  - `int` is the one that cannot be changed
  - this is why `u` **does not have to be initialized**, as `u` **IS ALLOWED to change** (i.e., the address it is pointing to can change)
- `int const* v;`
  - v is a constant pointer to int?

Pointer to const

As with pointers, read it at the identifier and work your way out

- const binds to the thing it is closest to
- `const int* u;`
  - `u` is a pointer to a constant int
  - `int` is the one that cannot be changed
  - this is why `u` **does not have to be initialized**, as `u` **IS ALLOWED to change** (i.e., the address it is pointing to can change)
- `int const* v;`
  - ~~v is a constant pointer to int~~
  - v is an (ordinary) pointer to an int that happens to be `const`
  - `const` is bound to `int` again (i.e., integer value can't change)

Stick to the first form (which is less confusing)

const Pointer

To make the pointer itself `const`, `const` must be placed **right after the ***

```
int d = 1;
int* const w = &d;
```

`w` is a `const` pointer to int

You can still change the value of `d (e.g., *d = 100);`

const Pointer to a const object

What would then be the correct method to make the **pointer** AND **the value held in that address** both `const`?

```
int d = 1;
```

const Pointer to a const object

What would then be the correct method to make the **pointer** AND **the value held in that address** both `const`?

```
int d = 1;

const int* const x = &d;   // (1) legal

int const* const x2 = &d; // (2) also legal, but less preferred (confusing)
```

Remember that

value inside the address stored in a pointer is **constant** - `const` is **before** *

```
const int* u;

int const* v;
```

Pointer is constant - `const` is **after** *

```
int* const w
```

Type Checking

C++ is VERY particular about type checking

- You don't have the same freedom as C
- For example, in C++ can you:
  - Assign the address of a non-`const` object to `const` pointer?
  - Assign the address of a `const` object to a non-`const` pointer?
  - Which is allowed?

Type Checking

C++ is VERY particular about type checking

- You don't have the same freedom as C
- For example, in C++ can you:
  - Assign the address of a non-`const` object to a `const` pointer
    - You're promising NOT to change something (the `const` pointer) that is okay to change (non-`const` object address)
  - Assign the address of a `const` object to a non-`const` pointer
    - You may change something (non-`const` pointer) that is not allowed (`const` object address) to be changed

Example

```
int d = 1;
const int e = 2;
int* u = &d; // OK?
int* v = &e; // OK?
int* w = (int*)&e; // OK?
int main() {}
```

Example

```
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
// int* v = &e; // Illegal -- e const
int* w = (int*)&e; // Legal but bad practice (and undefined
behavior)
int main() {}
```

Example

1)

```
const int e = 2;
const int* v = &e;
```

2)

```
const int e = 2;
int* const v = &e;
```

Which is legal?

Example

1)

```
const int e = 2;
const int* v = &e;  // v is a pointer to a const int (int cannot be changed)
```

2)

```
const int e = 2;
int* const v = &e; // v is a constant pointer to int (int can still change)
```

Function Arguments

You can specify function arguments as const when passing-by-value

```
void f1(const int i) {
  i++; // illegal
}
```

- You are making sure that i is something that should not change inside the function
- You are also making a promise that original value will not be changed by the function
  - However, this is implicitly kept since the value is passed-by-value

Function Arguments

Alternatively, you could say:

```
void f2(int ic) {
  const int i = ic;
  i++;  // still illegal
}
```

- Caller makes no assumption about the value being passed (other than that it's passed by value, so a copy is made inside)
- Inside the function, you are still not allowed to change the value of `ic`

Quiz

Quiz

Question 1 - Which of the following are NOT true?

A. `const` can be used to replace `#define`
B. `const` allows type checking whereas `#define` does not
C. constant folding is where you reuse an identical calculation that has already been done before to skip calculation at runtime
D. `const` are allowed in header files

Quiz

Question 2 - Given the following piece of code, what will be printed?

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

static 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;
}
```

A. 10
B. 100
C. Compile error
D. Undefined behavior

Quiz

Question 3 - Given the following piece of code which of the following options list the lines in the code that will cause a compile error?

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

1.    const int i[] = {1, 2, 3, 4};
2.    float f[i[0]];

3.    int main(int argc, char** argv)
4.    {
5.        const int j[] = {1, 2, 3, 4};
6.        float g[i[0]];
7.        float h[j[0]];
8.    }
```

A. Only lines 2, 6, and 7
B. Only lines 2 and 6
C. Only line 2
D. Only line 6
E. Only line 7

Quiz

Question 4 - Given the following piece of code which of the following options list the lines in the code that will cause compile error?

```
1.    int d = 1;
2.    const int e = 2;
3.    int* const f = &e;
4.    int* u = &d;
5.    int* v = &e;
6.    int* w = (int*)&e;
7.    int main() {}
```

A. Only lines 3, 4, 5, and 6
B. Only lines 4 and 5
C. Only line 5
D. Only lines 5 and 6
E. Only lines 3 and 5

Questions?

Previously…

Constants

- Designed to replace #define
- It will try to textual replace, and evaluate constant expressions (e.g., after textual replacement 10 + 20 = 30) at **compile** time
- Normally not given any storage
- All of the above changes when being used as a global variable with external linkage

const keyword

- If “const” goes before the *, than it applies to the data
- If “const” goes after the *, than it applies to the pointer variable

Questions?

Return Values

If you say that a function's return value is const

```
const int g() {int a = 1; return a;}
```

You are promising that the original variable (`a` from inside the function) will not be modified

- Again, this is implicit, since you're returning by value (it's a **copy** of the original variable `a` from inside the function, which will be **destroyed** when the function ends)

For **built-in types**, avoid returning value as const (as it can lead to more confusion and has no real impact)

However, things are different for **objects**

Return Values

`const` becomes more important when using user-defined types (e.g., classes)

If a function returns a class object, then it can be an l-value (unlike fundamental types, such as int, char, etc.)

If a function returns a class object as `const`, the return value of that function **cannot** be an l-value (i.e., it cannot be assigned or modified)

L-value - something that points to a memory location

- Exists as variables and lives longer

R-value - something that does not point to anything

- Temporary and short lived

```
int x = 666; // x is l-value, 666 is r-value
```

Left operand of an assignment operator must be an l-value,

L-value vs. R-value

```
int func()
{
    int x = 100;
    return x;
}

int main()
{
    func() = 200;
    return 0;
}
```

You are returning a **copy** of x, which is a **temporary**

main.cc: In function '**int main()**':
main.cc:12:14: error: lvalue required as left operand of assignment
    func() = **200**;

```
int& func()
{
    int x = 100;
    return x;
}

int main()
{
    func() = 200;
    return 0;
}
```

You are returning a **reference** to x, which makes it an l-value

However, not a good idea, since x will be "destroyed" when the function ends

main.cc: In function '**int& func()**':
main.cc:6:9: warning: reference to local variable '**x**' returned [**-Wreturn-local-addr**]
    int **x** = 100;
        ^

Example

```
class X {
  int i;
public:
  X(int ii = 0) { i = ii; }
  void modify() { i++; }
};
X f5() {
  return X();
}
const X f6() {
  return X();
}
void f7(X& x) {
  x.modify();
}
```

```
int main() {
  f5() = X(1); // OK -- non-const return value (usable as lvalue)
  f5().modify(); // OK

// Causes compile-time errors:
//!  f7(f5());
//!  f6() = X(1); // (not an lvalue)
//!  f6().modify();
//!  f7(f6());
} ///:~
```

Questions?

Returning a fundamental data type (int, char, etc.) creates an r-value (which are **short-lived** and **unmodifiable**)

Returning a class object creates an l-value (which can be **modified**)

- However, return it as **const** makes it an r-value (which **cannot be modified**)

Example

```cpp
class X {
  int i;
public:
  X(int ii = 0) { i = ii; }
  void modify() { i++; }
};
X f5() {
  return X();
}
const X f6() {
  return X();
}
void f7(X& x) {
  x.modify();
}
```

```cpp
int main() {
  f5() = X(1); // OK -- non-const return value (usable as lvalue)
  f5().modify(); // OK
// Causes compile-time errors:
//!  f7(f5());
//!  f6() = X(1); // (not an lvalue)
//!  f6().modify();
//!  f7(f6());
} ///:~
```

f7() will cause compile error because f5() and f6() create a **temporary**

Temporaries

During the evaluation of an expression, compiler must create temporary objects

- `int x = a * y + z;`
- `a * y` must be stored in a **temporary** before `z` is added
- Temporaries requires storage, and if they are objects, they are constructed and destroyed (like any other objects)
- **Temporaries** are **automatically** `const`
- **Modifying** a temporary is most likely a mistake/bug (even though the compiler may allow it)

Temporaries

In the example,

```
void f7(X& x) {
  x.modify();
}
X f5() {
  return X();
}
f7(f5());
```

A temporary object is required to **hold the return value** of `f5()`, so it can be **passed** to `f7()`

If `f7()` took argument by **value** (i.e., `void f7(X x)`), it would be **okay** - the temporary would be passed in normally

However, `f7()` takes in a **reference** (i.e., address of the temporary, kind of), and since it's **not** a `const` reference to `X`, it has **permission to modify** the temporary object

Since temporary will vanish (after it has served its purpose), compiler does not allow it

Questions?

Temporaries

Remember that the following was **allowed**:

```
f5() = X(1);
f5().modify();
```

This will also **cause problems**:

- `f5()` will return an object of type X
- It will disappear once it has served its purpose, so any modification will likely be lost
- The first `f5()` and second `f5()` are actually **two different** objects

Passing and Returning Addresses

Whenever you pass in a reference to a function (for efficiency), make it `const` (if possible)

Otherwise, it prevents the function from accepting arguments that are `const` (temporaries **can** be `const` reference)

For example,

```
void f13(int& x)
{
    cout << x << endl;
}
f13(100);
```

```
main.cc:27:4: error: no matching function for call to 'f13'
f13(100);
```

`void f13(const int& x)` will work

Questions?

Again, best way to test the inner-workings of const is to try and write code snippets to see which works and which doesn't

Example

```
void t(int*) {}

void u(const int* cip) {

//!  *cip = 2; // Illegal

  int i = *cip; // OK -- copies
value

//!  int* ip2 = cip; // Illegal:
ip2 is non-const

}

const char* v() {

  // Returns address of static
character array:

  return "result of function v()";

}

const int* const w() {

  static int i;

  return &i;

}
```

```
int main() {

  int x = 0;

  int* ip = &x;

  const int* cip = &x;

  t(ip);   // OK

//!  t(cip); // Not OK

  u(ip);   // OK

  u(cip); // Also OK

//!  char* cp = v(); // Not OK

  const char* ccp = v();  // OK

//!  int* ip2 = w(); // Not OK

  const int* const ccip = w();
// OK

  const int* cip2 = w();  // OK

//!  *w() = 1; // Not OK

} ///:~
```

Example

```
void t(int*) {}

void u(const int* cip) {

//!  *cip = 2; // Illegal

  int i = *cip; // OK -- copies value

//!  int* ip2 = cip; // Illegal: non-const

}

const char* v() {

  // Returns address of static character array:

  return "result of function v()";

}

const int* const w() {

  static int i;

  return &i;

}
```

```
int main() {

  int x = 0;

  int* ip = &x;

  const int* cip = &x;

  t(ip);  // OK

//!  t(cip); // Not OK

  u(ip);  // OK

  u(cip); // Also OK

//!  char* cp = v(); // Not OK

  const char* ccp = v(); // OK

//!  int* ip2 = w(); // Not OK

  const int* const ccip = w(); // OK

  const int* cip2 = w(); // OK

//!  *w() = 1; // Not OK

} ///:~
```

Example

```
void t(int*) {}

void u(const int* cip) {

//!  *cip = 2; // Illegal

  int i = *cip; // OK -- copies
value

//!  int* ip2 = cip; // Illegal:
non-const

}

const char* v() {

  // Returns address of static
character array:

  return "result of function v()";

}

const int* const w() {

  static int i;

  return &i;

}
```

```
int main() {

  int x = 0;

  int* ip = &x;

  const int* cip = &x;

  t(ip);  // OK

//!  t(cip); // Not OK

  u(ip);  // OK

  u(cip); // Also OK

//!  char* cp = v(); // Not OK

  const char* ccp = v(); // OK

//!  int* ip2 = w(); // Not OK

  const int* const ccip = w();
// OK

  const int* cip2 = w(); // OK

//!  *w() = 1; // Not OK

} ///:~
```

Example

```
void t(int*) {}

void u(const int* cip) {

//!  *cip = 2; // Illegal

  int i = *cip; // OK -- copies
value

//!  int* ip2 = cip; // Illegal:
non-const

}

const char* v() {

  // Returns address of  static
character array:

  return "result of function v()";

}

const int* const w() {

  static int i;

  return &i;

}
```

```
int main() {

  int x = 0;

  int* ip = &x;

  const int* cip = &x;

  t(ip);   // OK

//!  t(cip); // Not OK

  u(ip);   // OK

  u(cip); // Also OK

//!  char* cp = v(); // Not OK

  const char* ccp = v(); // OK

//!  int* ip2 = w(); // Not OK

  const int* const ccip = w(); // OK

  const int* cip2 = w(); // Why is
this OK?

//!  *w() = 1; // Not OK

} ///:~
```

Standard Argument Passing

It is possible to pass a **temporary** object to a function that takes `const` **reference** (but not a function that accepts a reference)

Temporary (remember, a temporary is always a `const`) can have its **address** passed to a function

```
class X {};
X f() { return X(); } // Return by value
void g1(X&) {} // Pass by non-const reference
void g2(const X&) {} // Pass by const reference
int main() {
  // Error: f() creates a temporary (which is const)
  //!  g1(f()); // g1 accepts non-const reference, so this is illegal

  // OK: g2 takes a const reference, so it's legal
  g2(f());
}
```

Questions?

const inside Classes

Let's say you want to `#define` an array size inside a class

- A `const int` inside the class does not produce the desired effect
- It reverts to the C definition - it's a storage allocated inside the object, initialized once, and cannot be changed for the lifetime of the **object**
- However, **each object may hold their own, different value** for this `const int`
- By its definition (i.e., `#define`), it should not change across objects for the same class

const in Classes

When you create a const inside a class, you cannot give it an initial value (and it's okay, since storage is allocated for it)

Initialization must occur in the constructor (like other variables), but in a special place in the constructor

Because `const` must be initialized when it is created, it must already be initialized when you reach the main body of the constructor (remember, storage is allocated for an object, and then the constructor is called to initialize it)

const in Classes

Use the **constructor initializer list**

Originally put in place for use in inheritance

Occurs only in the definition of the constructor

It is a list of constructor calls that occur after the function argument list (and a colon), but **BEFORE** the opening braces of the constructor body

const in Classes

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

class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) : size(sz) {} // same as size = sz;
void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
}
```

Variable Initialization

```
int a = 5; /* Regular */
int b(6);  /* constructor init */
int c{9};  /* uniform init */
```

- Uniform init requires new C++ standard
- This makes it easier to differentiate because {} and () – functional form
- This will make more sense when we start talking about classes and objects (and constructors)

const in Classes

To answer the original question - what is the proper way of **#define** constant value inside a class?

- Use the `static` keyword in addition to `const` - means only **one instance** of this data member, regardless of how many objects are created
- Similar to static variables in functions - they exist across multiple function calls, which can be seen as only **one copy** existing across all function calls
- `static const` of a built-in type is treated as a compile-time constant

const in Classes

```
class StringStack {
  static const int size = 100;
  const string* stack[size];
  int index;
public:
  StringStack();
  void push(const string* s);
  const string* pop();
};
```

Questions?

const Objects and Member Functions

const objects are created similarly to those of built-in types

```
const int i = 1;
const blob b(2);
```

Since blob is a const, **no member of the blob object must change during its lifetime**

How do we know which member functions can be executed safely (i.e., and not change its data)?

Declare a member function also `const`, and the compiler knows this function can be called safely

const Objects and Member Functions

```
class X {
  int i;
public:
  X(int ii);
  int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; } // reiterate const at definition
// error if you try to change any member of the object
// or call another non-const member function

int main() {
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
}
```

const Objects and Member Functions

Exceptions -

- Constructors and destructors are not const (even for a const object), since it almost always changes things
- But these are still called, since they are only called at initialization and at end of scope/life

Questions?

volatile

- As before, these are variables that can change by external forces (i.e., the compiler does not know)
- Even if you read the data at point A (to variable x) and not change it between A and B, you cannot assume the data has not changed
- Therefore, you cannot optimize away the read at point B and reuse the value in x
- You can also make objects `volatile`
- You can also create `const volatile` - you can't change its value but it may still change (by external forces)

volatile

```
class Comm {
  const volatile unsigned char byte;
  volatile unsigned char flag;
  enum { bufsize = 100 };
  unsigned char buf[bufsize];
  int index;
public:
  Comm();
  void isr() volatile;
  char read(int index) const;
};
Comm::Comm() : index(0), byte(0),
flag(0) {}
```

```
// Only a demo; won't actually workas
an interrupt service routine:
void Comm::isr() volatile {
  flag = 0;
  buf[index++] = byte;
  // Wrap to beginning of buffer:
  if(index >= bufsize) index = 0;
}
char Comm::read(int index) const {
  if(index < 0 || index >= bufsize)
    return 0;
  return buf[index];
}
int main() {
  volatile Comm Port;
  Port.isr(); // OK
//!  Port.read(0); // Error, read()
not volatile
} ///:~
```

Questions