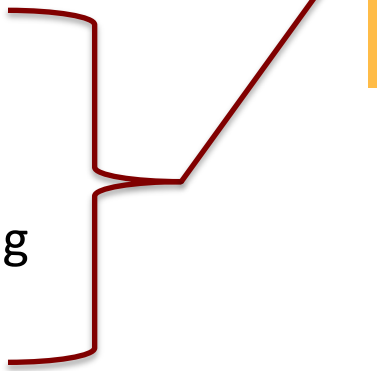
const and C++

CS3081 Program Design and Development

C++ Syntax

- C++
  - const
  - copy-constructor
  - operator overloading
  - templates



Eckel does a thorough job of laying out the intricacies of these complex constructs and concepts in C++.

It will make your head swim!

Why use const?

- Isolate code changes.
- Protect your data.

- Be explicit with your intentions.
  - Communicate with users.
  - Help the compiler help you.

- 3 audiences of your code
  - Fellow programmers. (Users and Team Members)
  - The future you.
  - The compiler (and linker).

Isolating Change

#define SIZE 100	**const int size = 100;**
• preprocessor directive • no type (no type checking) • no scope (preprocessed away)	• managed by compiler • always a type • no scope sometimes (compiled away) • constant folding (compile others away too) • internal linkage (opposite of global non-const variables) • can't define at runtime (unless part of class)

Safe Passage

- Constant Pointers
- Pointers to Constants
- Passing constant values
- Returning constant values
- Passing pointers (or references) to constants
- Returning pointers to constants
- Passing and returning “temporaries”

Constant Pointers and Pointers to Constants

- Pointer to an Integer

```
int* pointerToInt;
```

- Constant Integer

```
(const int) constInt;
```

- Pointer to a Constant Integer
  - YES: change address being pointed to (i.e. point to different constant).
  - NO: change what pointer points to (sort of).

```
(const int)* pointerToConstInt;

(int const)* pointerToConstInt2;
```

- Constant Pointer to an Integer
  - YES: change the contents of the address pointed to.
  - NO: change the address pointed to.

```
int* (const constPointerToInt);
```

- Constant Pointer to a Constant Integer
  - YES: nothing.
  - NO: everything.

```
(const int)* (const constPointerToConstInt);

(int const)* (const constPointerToConstInt);
```

Coding With Pointers to Constant Integers

- Pointer to a Constant Integer `(const int)* pointerToConstInt;`

```
// Pointer to a constant integer ----------------------
const int myConst1;
const int myConst2 = 200;
const int* pMyConst;

// initialize the integer
myConst1 = 100;

// initialize the pointer
pMyConst = &myConst2;

// change the pointer
pMyConst = &myConst2;

// change the integer
myConst2 = myConst2 + 5;

// change the integer using the pointer
*pMyConst = myConst1 + 1;

// change the integer through the back door
int* pBackDoor = (int *)&myConst2;
*pBackDoor = 50;
```

Coding With Constant Pointers to Integers

- Constant Pointer to an Integer

int* (const constPointerToInt);

```
// Constant Pointer to an integer ----------------------
int myInt_3;
int myInt_4 = 400;
int* const cpMyInt_3 = &myInt_3;
int* const cpMyInt_4;

// initialize the integer
myInt_3 = 300;

// initialize the pointer
cpMyInt_4 = &myInt_4;

// change the integer
myInt_4 = myInt_4 + 5;

// change the integer using the pointer
*cpMyInt_3 = myInt_3 + 1;

// change the pointer
cpMyInt_3 = &myInt_4;

// change the pointer through the back door
int** ppBackDoor = (int **) &cpMyInt_3;
*ppBackDoor = cpMyInt_4;
```

Rules About Constants and Parameter Passing

- Pass by value is safe and easy, but can be inefficient.

- Passing a pointer is a common means of getting data back.
  - It is explicit and a good way to communicate to the user.

```
- void updatePos( Robot robot, Pos* updatedPos );
- updatePos( robot, &newPos );
```

- Pass-by-reference is efficient.
  - It is quite hidden and NOT reassuring to the user.
  - Pass-by-reference with *const* is safe, efficient, but might ripple through code.

```
- void updatePos( const Robot& robot, Pos* updatedPos );
- updatePos( robot, &newPos );
```

Rules About Constants and Parameter Passing

- Passing non-constants to const parameters is OK.
- Passing const to a non-const parameter is NOT OK.

You can always get more restrictive, but not less.

- Compilers create temporary objects that are ALWAYS *const*.
- You cannot get the address of a temporary object.

```
int calcDist( const Pos& p1, const Pos& p2);
dist = calcDist( incPos( pos1 ), pos1 );
```

This comes into play when you use a function result as a parameter.

- Returning a const user-defined object means it cannot be an lvalue.

```
Pos incPos( const Pos& p );
incPos(pos1).setX(20);

const Pos incPos( const Pos& p );
incPos(pos1).setX(20);
```

The Chain Reaction of Adding Pass-By-Reference

```
class Pos {
private:
  int x;
  int y;
public:
  Pos() : x(0), y(0) {}
  Pos( int inX, int inY ) : x(inX), y(inY)
  void setX( int inX ) { x=inX; }
  void setY( int inY ) { y=inY; }
  int getX() { return x; }
  int getY() { return y; }
};
```

```
Pos incPos( Pos p ) {
  int x = p.getX() + 2;
  int y = p.getY() + 3;
  Pos pos( x, y );
  return pos;
}
```

```
int calcDist( Pos p1, Pos p2) {
  // some calculations here
  return 23;
}
```

```
int main() {
  Pos pos1(2,5);
  int dist;

  int x = pos1.getX() + 2;
  int x2 = incPos(pos1).getX();
  dist = calcDist( incPos( pos1 ), pos1 );
```

const and Classes

```
class Robot {
private:
  const int radius;
  const int color;
  int speed;
public:
  Robot();
  Robot( int r, int c, int s ) : radius(r), color(c), speed(s) {}

  void setSpeed( int inS ) { speed = inS; }
  void setColor( int inC ) { color = inC; }

  int getSpeed() { return speed; }
  int getRadius() { return radius; }
  int getColor() { return color; }
};
```

```
Robot::Robot() {
  radius = 50 ;
  color = 0xFF0000 ;
  speed = 250 ;
}
```

```
int main() {
  Robot robot1( 20, 0x00FF00, 200 );
  const Robot robot2;

  robot1.setSpeed( 45 );
  int s = robot1.getSpeed();

  robot2.setSpeed( 250 );
  int s2 = robot2.getSpeed();
}
```

Rules About Constants and Classes

- *const* data members must be initialized with an initialization list (except for static, which is initialized at compile time).
  - `Robot::Robot() : radius(50), color(0xFF0000) {}`
- Class methods follow same rules for *const* passing and returning as other methods.
- Using *const class* objects requires assurances to the compiler.
  - `int Robot::getSpeed() const { return speed; }`
- Calling a const method with a non-const class object is OK.
- Calling a non-const method with a const class object is NOT OK.

The Chain Reaction of Adding Pass-By-Reference

```
class Pos {
private:
  int x;
  int y;
public:
  Pos() : x(0), y(0) {}
  Pos( int inX, int inY ) : x(inX), y(inY)
  void setX( int inX ) { x=inX; }
  void setY( int inY ) { y=inY; }
  int getX() const { return x; }
  int getY() const { return y; }
};
```

const to play nice with const objects.

```
Pos incPos( const Pos& p ) {
  int x = p.getX() + 2;
  int y = p.getY() + 3;
  Pos pos( x, y );
  return pos;
}
```

const to assure users.

& for Efficiency

```
int calcDist( const Pos& p1, const Pos& p2) {
  // some calculations here
  return 23;
}
```

```
int main() {
  Pos pos1(2,5);
  int dist;

  int x = pos1.getX() + 2;
  int x2 = incPos(pos1).getX();
  dist = calcDist( incPos( pos1 ), pos1 );
```

const to use temporary parameters.