Constants

CONTENTS

Objectives

At the end of this unit we will be able to:

- Create constant objects and pointers
- Write functions that will work with constant arguments
- Select the appropriate parameter passing mechanism for a function

Constants

Constants

Constants

- are declared with the keyword **const**
- must be initialized
- can not be changed
- can only be passed to functions
  - — by value, or
  - — with a reference or pointer to **const**
- can invoke only **const** member functions

References to Constants

A reference can only refer to a constant object if it is declared as a reference to **const.** For example, `f1` 's parameter may refer to the `String s`, but not to the `const String cs`. C++ enforces this rule because a reference may be used to change the object it refers to (for example, `f1` can use the reference `str` to change `s`).

A reference to **const** can refer to either a constant or a non-constant object. For example, `f2` 's parameter may refer to either the `const String cs` or the `String s`. If a function with a reference parameter does not need to change its argument, the parameter should be a reference to **const,** to allow the use of the function with either constant or non-constant objects.

References to Constants

```
#include "String.h"

void f1(String &);
void f2(const String &);

main(int, char *[])
{
    String s;
    s = "hello world";

    const String cs = s;

    f1(s);    // f1 can be called only with non-const
    f2(cs);   // f2 can be called with either a const
    f2(s);    //    or a non-const

    return 0;
}

void f1(String &str)
{
    str = "new value";  // f1 can change its argument
}

void f2(const String &str)
{
    str.print();        // f2 can not change its argument
}
```

Pointers to Constants

A pointer can only point to a constant if it is declared as a pointer to **const.** A pointer to **const** can point to either a constant or a non-constant object.

Pointers to Constants

```
#include "String.h"

void f1(String *);
void f2(const String *);

main(int, char *[])
{
    String s;
    s = "hello world";

    const String cs = s;

    f1(&s);    // f1 can be called only with non-const
    f2(&cs);   // f2 can be called with either a const
    f2(&s);    //    or a non-const

    return 0;
}

void f1(String *str)
{
    *str = "new value";  // f1 can change (*str)
}

void f2(const String *str)
{
    str->print();        // f2 can not change (*str)
}
```

Constants and Call by Value

Constants can be passed by value whether or not the functions parameter is declared as a **const.** Since the function will work with a copy, instead of the original constant, it will not be able to affect the constant, even if it does change its argument.

Constants and Call by Value

```
#include "String.h"

void f1(String);
void f2(const String);

main(int, char *[])
{
    String s;
    s = "hello world";

    const String cs = s;

    f1(cs);   // f1 can be called with either a const
    f1(s);    //    or a non-const
    f2(cs);   // f2 can be called with either a const
    f2(s);    //    or a non-const

    return 0;
}

void f1(String str)
{
    str = "new value";  // f1 can change str, but not original
}

void f2(const String str)
{
    str.print();        // f2 can not change str
}
```

Const Member Functions

A **const** object of a class type can invoke a member function only if that member function is declared as a **const** member function[1]. A **const** member function can not change its invoking object, and can be invoked by either a constant or non-constant object.

```
void Employee::set_name(const String &n)
{
    name = n;
}

String Employee::get_name() const
{
    return name;
}
```

1. **const** member functions were not supported in C++ before release 2.0. In release 2.0, the use of a non-const member function on a **const** object generates only a warning (instead of an error) to allow compilation of code written for earlier versions of C++.

Const Member Functions

```
#include "String.h"

class Employee {
public:
    void set_name(const String &);
    String get_name() const;
    //...
private:
    String name;
    float salary;
};

main(int, char *[])
{
    String jones, smith;
    smith = "Smith";
    jones = "Jones";

    Employee e;
    e.set_name(jones);
    const Employee ce = e;

    e.set_name(smith);     // called only with non-const
    ce.get_name().print(); // called with either const
    e.get_name().print();  //              or non-const

    return 0;
}
```

Overloading Based on const

Overloading can be based on **const** arguments. In this case, C++ will call the function with **const** parameters when a call is made with constant arguments, and the function without **const** parameters when a call uses non-constant arguments. This is often useful when a function returns a reference to one of its arguments, and the function is to be used with both constant and non-constant arguments.

Overloading Based on const

```
#include "String.h"

String &longer(String &, String &);
const String &longer(const String &, const String &);

main(int, char *[])
{
    String s1, s2;
    s1 = "hello world";
    s2 = "bye";
    const String cs1 = s1, cs2 = s2;

    longer(cs1, cs2).print();       //    const version
    longer(s1, s2) = "new value";   // non-const version
    longer(s1, s2).print();         // non-const version

    return 0;
}

String &longer(String &str1, String &str2)
{
    return str1.length() > str2.length() ? str1 : str2;
}

const String &longer(const String &str1, const String &str2)
{
    return str1.length() > str2.length() ? str1 : str2;
}
```

Extending class String to Work with Constants

If the defining operations of class `String` are to be used with constant `Strings`, several changes must be made. Parameters passed by reference must be changed to references to **const**, for functions to be called with constant arguments. Pointer parameters must be changed to pointers to **const**, for functions that will work will pointers to constants. Member functions must be made into **const** member functions, if they are to work with constant `Strings`. In addition, two `operator[]` functions are needed: one for use with constant `Strings`, which will return a reference to **const** (so that the character in the **const** `String` can not be changed), and one for use with non-constant `Strings`, which will allow changes to the character returned.

Extending class String to Work with Constants

```
const int max_string_length = 128;

class String {

public:
    String &operator=(const char *);
    int  length() const;
    int  read();
    void print() const;

    const char &operator [](int) const;
    char & operator [](int);
    String substring(int start, int len) const;

    friend String operator+(const String &, const String &);
    friend String operator+(const String &, const char *);
    friend String operator+(const char *, const String &);

private:
    // a String is a sequence of up to
    // max_string_length non-null characters
    // followed by a null character

    char text[max_string_length+1];
};
```

String Member Functions

The definitions of class `String's` defining operations must be changed to agree with the definitions. Functions with **const** parameters can not change those parameters, and **const** member functions can not change their invoking object.

Some of the functions are shown on the facing page, and some below:

```
String String::substring(int start, int len) const
{
    //...
}

int String::length() const
{
    //...
}


int String::read()
{
    //...
}

void String::print() const
{
    //...
}
String operator+(const String &lhs, const String &rhs)
{
    //...
}
String operator+(const String &lhs, const char *rhs)
{
    //...
}

String operator+(const char *lhs, const String &rhs)
{
    //...
}
```

String Member Functions

```
// use: s = "hello world"

String &String::operator=(const char *rhs)
{
    strncpy(text, rhs, max_string_length);
    text[max_string_length] = '\0';

    return *this;
}

char &String::operator [](int index)
{
    if (index < 0 || index >= length()) {
      fprintf(stderr,
          "Illegal index (%d) for String \"%s\".\n",
          index, text);
      exit(1);
    }

    return text[index];
}
const char &String::operator [](int index) const
{
    if (index < 0 || index >= length()) {
      fprintf(stderr,
          "Illegal index (%d) for String \"%s\".\n",
          index, text);
      exit(1);
    }
    return text[index];
}
```

Using const Strings

If we had not made the changes to class `String`, we would not have been able to use constant `Strings` with the defining operations. Constant `Strings` could not have been used with the `operator+` functions, which used call by reference. They could not have been used with the `length`, `print`, and `substring` member functions, as these were not **const** member functions. They could not have been used with the original `operator[]` function, which was also a non-const member function.

Using const Strings

```
#include "String.h"

main(int, char *[])
{
    String s;
    s = "Zaphod Beeblebrox";
    const String cs = s;
    char ch;

// These would be illegal without the
// changes we made to class String:
    cs + s;
    "Hello, " + cs;
    cs + ".";
    cs.length();
    cs.print();
    cs.substring(0, 6);
    ch = cs[0];

// These are illegal:
#if defined TRY_ILLEGAL
    cs = "";
    cs.read();
    cs[0] = ch;
#endif

    return 0;
}
```

Function Parameter Declarations

Function Parameter Declarations

Does the function need to change calling functions' data?

If so, call by address will make that fact clear to the caller:

```
void func(String *changable);
String s;
func(&s);  // s is probably changed
```

If not, does it need to change its own copy of the argument?

If so, call by value will create a copy:

```
void func(String copy);
String s;
func(s);  // s is not changed
```

If not, use call by reference to **const:**

```
void func(const String &not_changable);
String s;
func(s);  // s is not changed
```

Creating Constant Pointers

A pointer to a constant (which we saw a few pages ago) can be moved to point to a different object, but it can not be used to change the object to which it points. A constant pointer, however, can be used to change the object to which it points, but it can not be moved to another object (the pointer, not the object pointed to, is considered **const**).

You can create a constant pointer by declaring a pointer with ***const** rather than *. Note that pointers to constants are sometimes called "constant pointers" when there is enough context to make the meaning clear.

Creating Constant Pointers

```
main(int, char *[])
{
    String s, t;
    s = "Zaphod Beeblebrox";
    const String name = s;

    // regular pointer to String:
    String *ps = &s;
    *ps = "Zaphod";
    ps = &t;

    // pointer to const String
    const String *pcs = &name;
    pcs = &s;
//  *pcs = "Zaphod";   illegal

    // const pointer to String
    String *const cps = &s;
    *cps = "Beeblebrox";
//  cps = &t;   illegal

    // const pointer to const String
    const String *const cpcs = &name;
//  *cpcs = "Zaphod";  illegal
//  cpcs = &t;  illegal

    return 0;
}
```

Summary

Summary

A constant may be used

- as an argument to a function using
  - — a reference to **const** parameter
  - — a pointer to **const** parameter
  - — call by value
- as an invoking object for **const** member functions

Non-constants may also be used with the above.

Exercises 7 Ex

Object-Oriented Programming in C++

Lab Exercises

UNIT 7

Lab Exercises

1. Change to the *unit07/point* directory. Copies of the test programs *use_ref.c* and *use_point.c* from Unit 6 are in this directory. Change the test programs for your class **Point** so that they create some **const Points**. For example, in the test program *use_point.c*, **Point c** is not changed, so it can be a constant. Similarly, in the test program *use_ref.c*, **Points p1** and **p2** can be constants. Note that you will need to initialize any constant **Points** with values of type **Point**, just as constant **Strings** are initialized with values of type **String** in the examples on pages 7-7 to 7-11 of the Student Guide. Try to compile the modified test program without making any changes to class **Point** (in file *Point.h*), and observe the result.

   You can compile and execute both test programs by entering '**make**' or you can compile and execute them directly using the commands:

```
$ CC -o use_ref use_ref.c point.c print.c
$ use_ref
$ CC -o use_point use_point.c point.c print.c
$ use_point
```

   Decide which member functions in your class **Point** do not change their invoking object, and add the keyword **const** to their declarations. Change the parameters of the member and friend functions to make them work with constants, when appropriate. The **print** functions in *print.c* should also be modified to include **const** where appropriate. Compile the two test programs with the new class **Point**. Will the old test program still work too? Can you write a test program that changes a **const Point**?

   Again, you can compile and execute both test programs by entering '**make**' or you can compile and execute them directly using the commands:

```
$ CC -o use_ref use_ref.c point.c print.c
$ use_ref
$ CC -o use_point use_point.c point.c print.c
$ use_point
```

SUMMARY	
DIRECTORY	unit07/point
DECLARATION	Point.h (modify), print.h (modify)
IMPLEMENTATION	point.c (modify), print.c (modify)
TEST PROGRAM	use_ref.c (modify), use_point.c (modify)

2. Change to the *unit07/string* directory. The file *same.c* is a copy of the file from the *unit06/string* directory. Change the **operator==** and **operator!=** functions so that they can be used with constant **String**s. Compile and execute the program *const_cmp.c* to test your changes.

   You can compile and execute this program by entering '**make prob2**' or you can compile and execute it directly using the commands:

```
$ CC -o const_cmp const_cmp.o string.o same.o
$ const_cmp
```

SUMMARY	
DIRECTORY	unit07/string
DECLARATION	String.h (modify)
IMPLEMENTATION	string.c, same.c (modify)
TEST PROGRAM	const_cmp.c

FILE: **const_cmp.c**

```
#include "String.h"
#include <stdio.h>

main(int, char *[])
{
                        String a;
                        a = "hello world";

                        const String b = a;

                        if (a == b)       printf("part 1 works\n");
                        if (b == a)       printf("part 2 works\n");
                        if ( !(a != b) ) printf("part 3 works\n");
                        if ( !(b != a) ) printf("part 4 works\n");

                        return 0;
}
```

UNIT 7

Lab Exercises (Answers)

1. The test programs *use_point.c* and *use_ref.c* from Unit 6 can be modified to create a **const Point**:

FILE: **use_point.c**

```
#include "Point.h"
#include "print.h"
#include <stdio.h>

main(int, char *[])
{
        Point a, b, init_c;

        a.set_to(1, 1);
        b.set_to(10, 10);
        init_c.set_to(100,100);

        const Point c = init_c;

        printf(" initially, a, b, and c are:\n");
        print(a);
        print(b);
        print(c);

        a = b + c;
        printf(" after \"a = b + c;\"\n");
        print(a);
        print(b);
        print(c);

        a += b;
        printf(" after \"a += b;\"\n");
        print(a);
        print(b);
        print(c);

        a = b += c;
        printf(" after \"a = b += c;\"\n");
        print(a);
        print(b);
        print(c);

        return 0;
}
```

FILE: **use_ref.c**

```
#include "Point.h"
#include "print.h"
#include <stdio.h>
```

```
main(int, char *[])
{
     Point init_p1, init_p2;
     init_p1.set_to(3, 5);
     init_p2.set_to(8, 2);

     const Point p1 = init_p1, p2 = init_p2;

     print ("printing (3, 5) + (8, 2): ");
     print (p1 + p2);
     print ("\nprinting (8, 2) - (3, 5): ");
     print (p2 - p1);
     print ("\nprinting (8, 2) * 5: ");
     print (p2 * 5);
     print ("\nprinting 3 * (3, 5): ");
     print (3 * p1);
     print ("\nprinting (8, 2) / 2: ");
     print (p2 / 2);
     if(p1 == p2)
        printf("\noperator == does not work");
     else
        printf("\noperator == works");
     if(p1 != p2)
        printf("\noperator != works");
     else
        printf("\noperator != does not work");
     print ("\n");

     return 0;
}
```

When the modification are made, C++ will produce error messages for statements in which the **Point c** is used as a reference or pointer argument to a function, or as an invoking object for a non-**const** member function. For example, the expressions "a = b + c" and "a = b += c" now give errors because **c** is passed by reference to the + and += operators. In addition, if we had changed our **print(Point)** function to use call by reference, the statement "print(c);" would also become illegal. To prevent such problems, we can change functions that do not modify their arguments to use references and pointers to **const**, and make member functions that do not change their invoking object into **const** member functions:

FILE: **Point.h**

```
class Point {
public:
       int x() const;
       int y() const;
       void set_to(int x, int y);

       Point operator+(const Point &) const;
       Point operator-(const Point &) const;

       friend Point operator*(int,  const Point &);
       friend Point operator*(const Point &, int);

       Point operator/(int) const;
```

```
        int operator==(const Point &) const;
        int operator!=(const Point &) const;

        Point &operator+=(const Point &);

private:
        int _x;
        int _y;
};


inline int Point::x() const
{
   return (_x);
}

inline int Point::y() const
{
   return(_y);
}
```

FILE: **point.c**

```
#include "Point.h"
#include <stdio.h>

void Point::set_to(int x, int y)
{
   _x = x;
   _y = y;
}

Point Point::operator+(const Point &p) const
{
   Point temp;
   temp._x = _x + p._x;
   temp._y = _y + p._y;
   return temp;
}

Point Point::operator-(const Point &p) const
{
   Point temp;
   temp._x = _x - p._x;
   temp._y = _y - p._y;
   return temp;
}

Point operator*(int i, const Point &p)
{
   Point temp;
   temp._x = i * p._x;
   temp._y = i * p._y;
   return temp;
}

Point operator*(const Point &p, int i)
```

```
{
   Point temp;
   temp._x = i * p._x;
   temp._y = i * p._y;
   return temp;
}

Point Point::operator/(int i) const
{
   Point temp;
   temp._x = _x / i;
   temp._y = _y / i;
   return temp;
}

int Point::operator==(const Point &p) const
{
   return (_x == p._x && _y == p._y)?1:0;
}


int Point::operator!=(const Point &p) const
{
   return (_x != p._x || _y != p._y)?1:0;
}

Point &Point::operator+=(const Point &p)
{
   *this = *this + p;
   return *this;
}
```

Other functions that use call by reference can also be updated to use reference to **const**:

FILE: **print.h**

```
// if using C++ 1.2 or before,
// then use "overload print;" here

void print (const char *);
void print (const double);
void print (const int);
void print (const Point &);
```

FILE: **print.c**

```
#include "Point.h"
#include "print.h"
#include <stdio.h>

void print (const char *s)
{
  printf("%s", s);
}
```

```
void print (const double f)
{
  printf("%lg", f);
}

void print (const int i)
{
  printf("%d", i);
}

void print(const Point &p)
{
   printf("(%d,%d)\n", p.x(), p.y());
}
```

2. To allow the use of the comparison operators with **const Strings**, the reference arguments must be changed to references to **const**, and the member functions must be made into **const** member functions:

FILE: **String.h**

```
const int max_string_length = 128;

class String {

public:
     String &operator=(const char *);
     int  length() const;
     int  read();
     void print() const;

     const char & operator [](int) const;
     char & operator [](int);
     String substring(const int start, const int len) const;

     friend String operator+(const String &, const String &);
     friend String operator+(const String &, const char *);
     friend String operator+(const char *, const String &);

     friend int operator==(const String &s1, const String &s2);
     friend int operator!=(const String &s1, const String &s2);
     friend int operator==(const String &s1, const char *s);
     friend int operator!=(const String &s1, const char *s);
     friend int operator==(const char *s, const String &s1);
     friend int operator!=(const char *s, const String &s1);

private:
     // a String is a sequence of up to
     // max_string_length non-null characters
     // followed by a null character

     char text[max_string_length+1];
};
```

FILE: **same.c**

```
#include <string.h>
#include "String.h"

int operator==(const String &s1, const String &s2)
{
   return !strcmp(s1.text, s2.text);
}

int operator!=(const String &s1, const String &s2)
{
   return strcmp(s1.text, s2.text);
}


int operator==(const String &s1, const char *s2)
{
   return !strcmp(s1.text, s2);
}

int operator!=(const String &s1, const char *s2)
{
   return strcmp(s1.text, s2);
}

int operator==(const char *s, const String &s1)
{
   return !strcmp(s1.text, s);
}

int operator!=(const char *s, const String &s1)
{
   return strcmp(s1.text, s);
}
```