Type Casting

CONTENTS

Objectives

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

- create constructors for type casting
- create type cast operators
- distinguish between casting values, pointers, and references

Type Casting

Type casting does not affect the object being cast: it generates a new object of another type, with the equivalent value. If we cast the `int` variable `i` to type `float`, `i` is unchanged: the compiler simply generates a temporary value of type `float` that has the same value as `i`. That temporary may be stored on the stack (as if it were a local variable), or it may be kept in a register if the machine has an available register of the right type.

Type Casting

To perform a type cast, the compiler

- allocates temporary storage
- initializes temporary with value being cast

```
float f(int i, int j)
{
    return (float) i / j;
}

// compiler generates:
float f(int i, int j)
{
    float temp_i = i, temp_j = j;
    return temp_i / temp_j;
}
```

Casting to Type String

To cast a value of type `char *` to type `String`, C++ must allocate a temporary `String`, and initialize that temporary with the `char *` value. C++ can only do this if we have provided a constructor to initialize a `String` with a `char *` value.

Note that there are two syntaxes for type casting to a class type: The parenthesis may be placed around the type name (as in C), or around the value being cast. The new syntax allows the casting of more than one value:

```
class complex {
public:
    complex();
    complex(float real_part);
    complex(float real_part, float imaginary_part);

private:
    float real, imag;
// or: float angle, distance;
};

void func(const complex &);

main(int, char *[])
{
    func( complex(4.0) );
    func( (complex) 4.0 );

    func( complex(0.5, 1.0) );

//  func(0.5, 1.0) illegal
//  func( (complex) 0.5, 1.0 )  illegal

    return 0;
}
```

Casting to Type String

```
#include "String.h"

main(int, char *[])
{
    String name;
    name = (String) "Zaphod " + String("Beeblebrox");

    name.print();

    ( (String) "Zaphod Beeblebrox" ) .print();

    return 0;
}
```

Simplifying Class String

Now that C++ can cast a `char *` value to type `String`, we can remove some of the defining operations from class `String`. We no longer need special operations to assign a `char *` to a `String`, or to add `char *` values to `Strings`.

Simplifying Class String

```
const int max_string_length = 128;

class String {

public:
    String();
    String(const char *);
    String(const String &);
    ~String();

    String &operator=(const String &);
    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 &);

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];
};
```

Using Class String

Even though we have removed the special `operator=` that assigned a `char *` value to a `String`, we can still perform these assignments. C++ will implicitly cast the value on the right from type `char *` to `String`. Similarly, we can still add `char *` and `String` values: C++ will cast the `char *` into a `String` before calling `operator+(const String &, const String &)`

We have simplified our class `String`, but lost some efficiency. The concatenation of a `String` and a character array used to require only a call to the special `operator+` that took those arguments. Now, it requires both a cast (using the constructor) and a call to the `operator+` for `Strings`.

C++ will not implicitly cast a value that is to be used as the invoking object of a member function. For this reason, `operator+` must still be a friend of class `String`. If it were a member, C++ would not allow the expression `" Hello " + name`, which requires an implicit cast of the left operand, which would be the invoking object of `operator+`.

Using Class String

```
#include "String.h"

main(int, char *[])
{
    String firstname, name;

    firstname = "Zaphod";
    name = firstname + " " + "Beeblebrox";

    name.print();
    ( (String) "Zaphod Beeblebrox" ) .print();

    return 0;
}
```

Constructors and Argument Passing

In C++, parameter passing and value returning are, by definition, equivalent to initialization. When a function is called, its formal arguments are initialized with the actual arguments passed. Upon return from a function, the value of the temporary that holds the function's result is initialized with the returned value. For this reason, class-type function arguments and return values will be copied by the class's constructor functions, even if there is no cast required.

For example, C++ uses the constructor `String::String(const String &)` to copy the `String` argument `statement` to the `sentence` function's formal argument `words`.

Constructors and Argument Passing

```
#include "String.h"

String sentence(String words, char *punctuation = ".");

main(int, char *[])
{
    String statement = "Hello, Zaphod";

    sentence(statement).print();
    sentence("Do you have any Tea", "?").print();

    return 0;
}

String sentence(String words, char *punctuation)
{
    return words + punctuation;
}
```

Ambiguities

If there is more than one possible match for an overloaded function name, C++ may require an explicit cast.

When trying to match an overloaded name to one particular function, C++ looks for the "best" match according to the following rules[1]:

1. exact match
2. match with promotions: integral conversions, `float` to `double`, or "trivial" conversions (e.g., *type*`[]` to *type*`*`, or *type* to **const** *type*)
3. match with standard conversions (e.g., `int` to `float`, pointer type to `void *`)
4. match with conversions requiring temporaries
5. match with user-defined conversion
6. match with "..."

If there is no "best match" according to the above rules (e.g., two functions that both require user-defined conversion, and no function that is higher on the list), the call is considered ambiguous, and C++ produces an error message.

1. Excerpted from **AT&T C++ language System Release 2.0 Product Reference Manual**

Ambiguities

```
#include "String.h"

class example {
public:
    example(const char *);
    //...
};

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

main(int, char *[])
{
//   f1("hello, world");  is ambiguous

     f1( (String)  "hello world" );
     f1( (example) "hello world" );
//   or provide void f1(const char *)
}
```

Type Casting from Class Types

Constructors can not be written for type casts from class types to built-in types, because there is no way to write a constructor for a built-in type. To allow casts from a class type to a built-in type, overload the type casting operator. We have provided an `operator const char*` rather than `operator char*` to keep users from changing the characters in the `String`. If we had provided `operator char*`, the users could have created an illegal `String`:

```
String s;
char very_large_array[1000];
// fill very_large_array

strcpy(s, very_large_array);
```

Type Casting from Class Types

```
const int max_string_length = 128;

class String {

public:
     String();
     String(const char *);
     String(const String &);
     ~String();

   operator const char*() const;

     String &operator=(const String &);
     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 &);

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::operator char*

The facing page shows the definition of the type cast operator to cast a String to type char *.

String::operator char*

```
#include "String.h"

//
// operator const char* for Strings
//
// use:
//     (const char *) s
//  or a String used where a const char * is needed:
//     open(s, O_RDONLY);
//

String::operator const char*() const
{
    return text;
}
```

More Ambiguities

There is a problem with defining both a rule for casting `char *` to `String`, and a rule for casting `String` to `char *`: Expressions involving both `String` and `char *` values may be ambiguous, because C++ does not know whether to cast the `String` to type `char *`, or the `char *` to type `String`. For example, if we had an `operator-` to subtract `Strings`, the expression `name - " Beeblebrox "` could be evaluated either (a) by casting `name` to type `char *` and subtracting pointers, or (b) by casting `" Beeblebrox "` to type `String` and subtracting `Strings`. Note that the expression `name + " Beeblebrox "` would not be ambiguous, because addition of pointers is illegal (so the only way to evaluate the expression is to cast `" Beeblebrox "` to type `String`).

More Ambiguities

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

String operator-(const String &, const String &);

main(int, char *argv[])
{
    int fd;
    String filename = "/tmp/test";

  // cast filename to type char *
    fd = open(filename, O_WRONLY | O_CREAT, 0666);
    write(fd, "test", 4);
    close(fd);

  // not legal, since we can cast only to const char *
  // strcpy(filename, "zbc");

    String name = "Zaphod Beeblebrox";
  // name - "Beeblebrox";   // is now ambiguous.

    return 0;
}
```

Avoiding Ambiguities

If we allow type casting from type `char *` to type `String`, but not the other way around, we can avoid the whole class of ambiguities introduced on the previous page. We still need a way to use `String` variables with functions with `char *` arguments (e.g., `open`), so we will provide a member function `as_char_pointer`.

```
#include "String.h"

//
// use:
//     open(s.as_char_pointer(), O_RDONLY);
//

const char *String::as_char_pointer() const
{
    return text;
}
```

Avoiding Ambiguities

```
const int max_string_length = 128;

class String {

public:
    String();
    String(const char *);
    String(const String &);
    ~String();

  const char *as_char_pointer() const;

    String &operator=(const String &);
    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 &);

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];
};
```

Converting Strings to Type char *

Users of class `String` can use the member function `as_char_pointer` to convert a `String` into an equivalent `char *` value, but C++ will not be able to perform a type cast. Therefore, there will be no ambiguity about which operand should be cast, in our use of the hypothetical `operator-`.

Converting Strings to Type char *

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

String operator-(const String &, const String &);

main(int, char *argv[])
{
    int fd;
    String filename = "/tmp/test";

  // convert filename to type char *
    fd = open(filename.as_char_pointer(),
          O_WRONLY | O_CREAT, 0666);
    write(fd, "test", 4);
    close(fd);

  // not legal
  // strcpy(filename.as_char_pointer(), "zbc");

    String name = "Zaphod Beeblebrox";
    name - "Beeblebrox";   // is no longer ambiguous.

    return 0;
}
```

Casting Pointers & References

When cast a value of one type into another type, C++ creates a temporary object of the appropriate type, and gives that temporary the value that is equivalent to the value being cast (e.g., it generates the value 3 as the integer equivalent of 3.14). It will generate this "equivalent" value according to rules built into the compiler (for built-in types) or rules defined in the class (for class types), so the process of conversion does not violate data encapsulation.

When we cast a pointer to a different kind of pointer, C++ converts the pointer value, but not the object pointed to. In our example `* ( (int *) &f_var )`, C++ converts the pointer `&f_var` to type `int *`, but does not do anything with `f_var` itself. When we use the resulting pointer, we will be interpreting the bi[ht pattern that makes up the `float` object `f_var` as an `int`. The result we get depends on the representation of `int` and `float` objects. Using a reference type in a cast causes a similar re-interpretation of a `float` object as if it were an `int`.

Note that none of the casts changes `f_var`.

Pointer casting can be useful, as long as you remember that you are introducing implementation dependencies. For example, the following statements will write out or read in floating point numbers very efficiently, but the program and machine reading in the numbers must use the same representation of `float` that was used by the program and machine that wrote the numbers out.

```
// to write a float to file descriptor fd:
write(fd, &f_var, sizeof(float));

// to read it in again:
read(fd, &f_far, sizeof(float));
```

Casting Pointers & References

Casting a value:

```
float f_var = 3.14;
printf("%d", (int) f_var);
```

- creates a temporary object
- does not violate data encapsulation

Casting a pointer or reference

```
printf("%d", *((int *) &f_var));
printf("%d", (int &) f_var);
```

- re-interprets representation of object
- violates data encapsulation
- results are implementation dependent

Summary

Summary

Type casting

- uses constructor
- or type cast operator
- reduces the number of functions needed
- can cause ambiguities, if over used

Exercises 9 Ex

Object-Oriented Programming in C++

Lab Exercises

UNIT 9

Lab Exercises

1. Change to the *unit09/point* directory. How can the function **add_ten** in the file *use_point2.c* be simplified by taking advantage of the constructors added in exercise 1 of the Unit 8 lab? Modify this file to take advantage of the constructors. Then compile and execute the test program *use_point2.c*.

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

```
$ CC -o use_point2 use_point2.c point.c print.c
$ use_point2
```

SUMMARY	
DIRECTORY	unit09/point
DECLARATION	Point.h, print.h
IMPLEMENTATION	point.c, print.c
TEST PROGRAM	use_point2.c (modify)

FILE: **use_point2.c**

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

Point add_ten(const Point &p)
{
        Point p1;
        p1.set_to(10, 10);
        return p + p1;
}


main(int, char *[])
{
        Point a(1, 1);

        printf(" a is:\n");
        print(a);

        printf(" add_ten(a) is:\n");
        print(add_ten(a));

        return 0;
}
```

2. Change to the *unit09/string* directory. This exercise is a continuation of exercise 2 of Unit 8. The *unit09/string* directory contains the same modified versions of *string.c* and *encap.c* created in the Unit 8 lab exercises. Examine the files *lab2a.c*, *lab2b.c* and *lab2c.c* and predict which constructors and assignment operators will be called. Compile and execute the programs to see if your predictions are correct.

   You can compile and execute each test program by entering '**make prob2*n***' (*n* = a, b, or c); you can compile and execute all of them by entering '**make**'; or, you can compile and execute them directly using the commands:

```
$ CC lab2a.c string.c encap.c -o lab2a
$ lab2a
$ CC lab2b.c string.c encap.c -o lab2b
$ lab2b
$ CC lab2c.c string.c encap.c -o lab2c
$ lab2c
```

SUMMARY	
DIRECTORY	unit09/string
DECLARATION	String.h
IMPLEMENTATION	string.c, encap.c
TEST PROGRAM	lab2*x*.c (x = a, b, c)

UNIT 9

Lab Exercises (Answers)

1. The function can be simplified by omitting the temporary **Point** variable:

FILE: **use_point2.c**

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

Point add_ten(const Point &p)
{
        return p + Point(10, 10);
}


main(int, char *[])
{
        Point a(1, 1);

        printf(" a is:\n");
        print(a);

        printf(" add_ten(a) is:\n");
        print(add_ten(a));

        return 0;
}
```