Object-Oriented Programming

Unit #3

Object Lifecycle (Construction and Destruction)
Type Conversions

1

Meeting Outline

- **Construction and Destruction**
- Type Conversions

2

Learning Outcomes

- Knowing what and how to perform the basic steps when creating/destroying an object
- Being able to use the appropriate new C++-style cast operators where appropriate

3

Constructors

- **Constructor**: code that executes each time an object is created
  - **Default Constructor**: an object is created without supplying any arguments
  - **Copy Constructor**: an object is created as a copy of already existing object of the same type
  - **Conversion Constructor**: an object is created from another object of a different type
  - **Regular Constructor**: an object is created from a set of objects of various types
- **Destructor**: code that executes each time an object is destroyed
- **Assignment Operator**: code that executes each time a state of an object should be a copy of a state of another object

4

On The House Code Generation

The following code will be automatically generated by a compiler if not supplied by a programmer:

- **Default constructor**: calls to default constructor of each member.
- **Default copy constructor**: calls to copy constructors for each member
- **Default assignment operator**: performs memberwise assignment
- **Default Destructor**: calls to destructor for each member

5

Default Constructor Example

```
class MyString
{
public:
  MyString();
private:
  char * strBuf;
  int    nLength;
};
```

```
MyString::MyString()
{
  nLength=0;
  strBuf = new char[1];
  strBuf[0] = '\0';
}
```

Each value setting from within the constructor body is an **assignment**

```
MyString str1;
MyString
  *pstr2 = new MyString(),
  *pstr3 = new MyString;
```

6

Member Initialization List

```
class MyString
{
public:
  MyString();
private:
  char * strBuf;
  int    nLength;
};
```

```
MyString::MyString()
  : nLength(0),
    strBuf(new char[1])
{ strBuf[0] = '\0'; }
```

- Each value set from within the member initialization list is a **member constructor call**
- Order execution of initializations depends on order of members' declaration

```
MyString str1;
MyString
 *pstr2 = new MyString(),
 *pstr3 = new MyString;
```


Destructor Example

```
class MyString
{
public:
  ~MyString();
private:
  char * strBuf;
  int    nLength;
};
```

```
MyString::~MyString()
{
  delete[] strBuf;
}
```

Members destructors will be called anyway

Conversion Constructor Example

```
class MyString
{
public:
  MyString(const char *);
  MyString(const double &);
private:
  char * strBuf;
  int    nLength;
};
```

```
MyString::MyString(
  const char *o)
  : nLength(strlen(o)),
    strBuf(strdup(o))
{}

MyString::MyString(
  const double & d)
{
  char tmp[64];
  sprintf(tmp, "%f", d);
  nLength = strlen(tmp);
  strBuf = strdup(tmp);
}
```

```
void f(const MyString & str)
{
  cout << str << endl;
}

main() {f("Conversion Test");}
```

A temporary **MyString** instance will be created automatically

9

Bad Conversion Constructor Example

```
class MyString
{
public:
  MyString(
    const double &d);
private:
  char * strBuf;
  int    nLength;
};
```

```
MyString::MyString(
  const double &d)
{
  char tmp[64];
  sprintf(tmp, "%f", d);
  nLength = strlen(tmp);
  strBuf = strdup(tmp);
}
```

```
MyString translate(MyString str)
{ ... }

translate(100.);
```

A temporary **MyString** instance containing *"100"* will be created

10

Good Conversion Constructor Example

```
class MyString
{
public:
  explicit MyString(
    const double &d);
private:
  char * strBuf;
  int    nLength;
};
```

```
MyString::MyString(
  const double &d)
{
  char tmp[64];
  sprintf(tmp, "%f", d);
  nLength = strlen(tmp);
  strBuf = strdup(tmp);
}
```

```
MyString translate(MyString str)
{ ... }
```

```
translate(100.);
translate(MyString(100.));
```

Error! No implicit conversions are allowed

Explicit conversions are allowed

11

Bad Default Copy Constructor Example

```
class MyString
{
public:
private:
  char * strBuf;
  int    nLength;
};
```

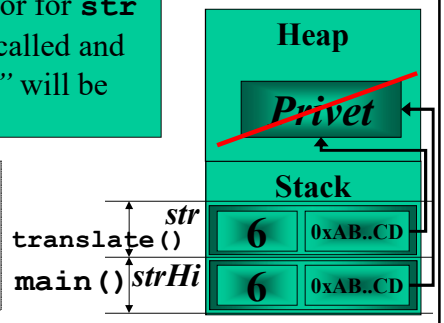
No copy constructor defined

```
MyString translate(MyString str)
{ ... }
```

str is a memberwise (shallow) copy of **strHi**

At the end of the block the destructor for **str** will be called and *"Privet"* will be freed

```
MyString strHi("privet");
cout << "privet is :"
     << translate(strHi)
     << endl;
```



12

Good Copy Constructor Example

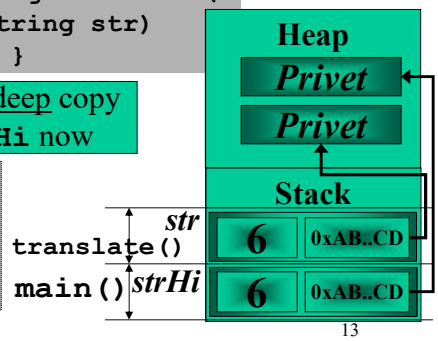
```
class MyString
{
public:
  MyString(
    const MyString &o);
private:
  char * strBuf;
  int    nLength;
};
```

```
MyString::MyString(
  const MyString &o)
  : nLength(o.nLength),
    strBuf(strdup(o.strBuf))
{}
```

```
MyString translate(
  MyString str)
{ ... }
```

str is a deep copy of **strHi** now

```
MyString strHi("privet");
cout << "privet is :"
     << translate(strHi)
     << endl;
```



13

Bad Assignment Operator Example

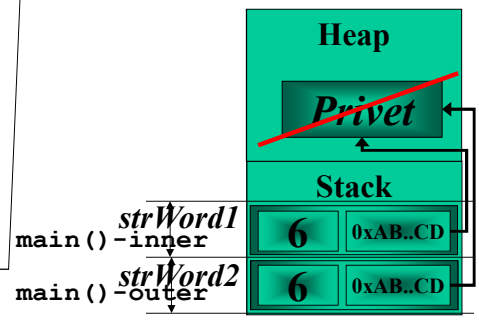
```
class MyString
{
public:
  char * strBuf;
  int    nLength;
};
```

No assignment operator defined

```
MyString strWord1("privet");
{
  MyString strWord2(...);
  strWord2 = strWord1;
}
```

At the end of the inner block the destructor for **strWord2** will be called and *"Privet"* will be freed

strWord2 is a memberwise (shallow) copy of **strWord1**



14

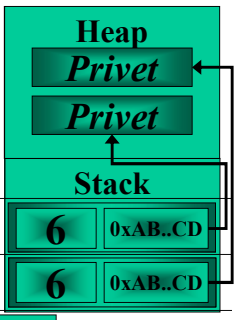
Good Assignment Operator Example

```
class MyString
{
public:
  MyString&
    operator=(
      const MyString& strSrc);
private:
  char * strBuf;
  int    nLength;
};
```

Custom assignment operator

```
MyString strWord1("privet");
{
  MyString strWord2(...);
  strWord2 = strWord1;
}
```

strWord2 is a deep copy of **strWord1** now




```
MyString&
 MyString::operator=(
      const MyString& strSrc)
{ ... }
```

Implementation explanation pending

15

Assignment Operator Implementation

```
class MyString
{
public:
  MyString& operator= (const MyString &);
private:
  char * strBuf;
  int    nLength;
};
```

```
MyString& MyString::operator = (const MyString & o)
{
  if (this == &o)      return *this;
  if (strBuf != NULL) free(strBuf);
  nLength = o.nLength;
  strBuf = strdup(o.strBuf);
  return *this;
}
```

16

Trinity Rule Of Thumb

If one of the following should be explicitly defined then all the rest should be defined as well:

- Copy constructor
- Destructor
- Assignment operator

17

Default Constructor Example

```
class MyString
{
public:
  MyString();
private:
  char * strBuf;
  int    nLength;
};
```

```
MyString::MyString()
{
  nLength=0;

  strBuf = new char;

  strBuf[0] = '\0';
}
```

Each value setting from within the constructor body is an **assignment**

```
MyString str1;
MyString
 *pstr2 = new MyString(),
 *pstr3 = new MyString;
```

18

Member Initialization List

```
class MyString
{
public:
  MyString();
private:
  char * strBuf;
  int    nLength;
};
```

```
MyString::MyString()
  : nLength(0),

    strBuf(new char)

{ strBuf[0] = '\0'; }
```

- Each value set from within the member initialization list is a **member constructor call**
- Order execution of initializations depends on order of members' declaration

```
MyString str1;
MyString
  *pstr2 = new MyString(),
 *pstr3 = new MyString;
```

19

Meeting Outline

- Construction and Destruction
- **Type Conversions**

20

Conversion

Representing a value of a specific type as a value of an another type

21

Conversion Types

Issue	Implicit	Explicit
carried by	*compiler*	*programmer*
complexity	*trivial*	*non-trivial*
data loss	*no (usually)* *possible (demotion)*	*possible*
robustness loss	*no (usually)* *possible (void conversions)*	*possible*

22

Implicit Conversion

Promotion (no data loss)

Original types	Converted types (in significance order)
`unsigned/signed char` `unsigned short`	`int` `unsigned int`
`wchar_t` `enum`	`signed/unsigned int` `signed/unsigned long`
`bool` *(C++ only)*	`int`
`float`	`double`

Demotion (guaranteed data loss)

23

Implicit Conversion (cont)

Less robust → More robust

Original types	Converted types (in significance order)
T	`const` *T*
T `*`	`const` *T* `*`
T `&` *(C++ only)*	`const` *T* `&`

More robust → Less robust

Original types	Converted types (in significance order)
T `*`	`void *`

24

C Explicit Conversion

- Advantages (very few)
  - Simple syntax
  - Less code to type
- Disadvantages (a lot)
  - Same syntax for many purposes
    - Compiler verification is impossible
    - Misleading for a maintaining programmer

```
double d = 3.14;
int n =
  (int) d;
const int *pn1 = &n;
int * pn2 =
  (int *) pn1;
double *pd =
  (double *) n;
```

Relative types, data is lost

Const away, robustness is harmed

Non-related types, data reinterpretation, not robust at all

C++ has much more powerful casting operators

25

Explicit Conversion – Old C Style

Do not use it, but a new C++-style, presented on the next slide

- Advantages (very few)
  - Simple syntax
  - Less code to type
- Disadvantages (a lot)
  - Same syntax for many purposes
    - Compiler verification is impossible
    - Misleading for a maintaining programmer

```
double d = 3.14;
int n =
  (int) d;
const int *pn1 = &n;
int * pn2 =
  (int *) pn1;
double *pd =
  (double *) n;
```

Relative types, data is lost

Const away, robustness is harmed

Non-related types, data reinterpretation, not robust at all

26

Explicit Conversion – New C++ Style

- Disadvantages (very few)
  - More complicate syntax
  - A bit more code to type
- Advantages (a lot)
  - Different syntax for different purposes
    - Compiler verification is possible
    - The code is easier to maintain

Non-related types, data reinterpretation, not robust at all

```
double d = 3.14;
int n = static_cast<int>(d);
const int *pn1 = &n;
int * pn2 = const_cast<int *>(pn1);
double *pd = reinterpret_cast<double *>(n);
```

Relative types, data is lost

Const away, robustness is harmed

27

Explicit Conversion – New C++ Style (cont)

- **`const_cast`** – casts away the **`const`**-ness
- **`static_cast`** – casts between related types
- **`reinterpret_cast`** – nonstandard casts, are **solely** on programmer's responsibility
- **`dynamic_cast`** – to be explained later

28

Summary

- Constructor (default, copy, conversion)
- Member initialization list
- Destructor
- Trinity rule of thumb

29