Literal Constants

A literal string constant is composed of zero or more characters enclosed in double quotation marks.

Examples:
```
"" (null string)
"x"
"hello"
"Hi,
How are you?"
```

Literal Constants

A string literal can be written across multiple lines. You can use a backslash as the last character on a line to indicate that the string continues on the next line.

Example:
```
"This is an 
excellent 
multi-line string literal."
```

Variables

Variables provide us with named memory storage that we can
• write to,
• read from, and
• manipulate throughout the course of our program.

Each variable has a specific type, which determines
• the size and layout of its associated memory,
• the range of values that can be stored, and
• the set of operations that can be applied to it.

Variables are also referred to as objects.

Variables

There are two values associated with a variable:
1. Its data value, which is stored at some memory address. It is also called the rvalue (read value) of the variable.
2. Its address value, indicating the location in memory where its data value is stored. This value is also referred to as the variable's lvalue (location value).

While literal constants also have an rvalue, they do not possess an lvalue.

Variables

Names (identifiers) of variables can be made up of letters, digits, and underscores.
• Names cannot start with a digit.
• Notice that C++ compilers distinguish between lower- and uppercase letters.
• C++ keywords such as if, else, class, etc. cannot be used as variable names.

Variables

Naming conventions:
• Object-oriented style: Use lowercase letters for names of objects; capitalize the first letter of each embedded word in a multiword identifier.

Examples: year, robotCameraModule

• “Microsoft style”: Also provide type information in object names.

Examples: iYear, strFilename

General advice: Use mnemonic names, that is, names describing the purpose of the object.
Variables

Initialization of variables:
• If a variable is defined at global scope, it automatically receives an initial default value (e.g., zero).
• Variables at local scope and dynamically allocated variables receive an undefined value.
• It is generally recommended that you assign an initial value to all variables.
  Example: int loopCounter = 0;
• Class objects are automatically initialized through their default constructor.

Further examples of initialization:
• int year = 2002;
• string myName = "Peter";
• int year(2002);
• string myName("Peter");
• int loopCounter = int();  // sets loopCounter to 0
• double myWeight = double(); // sets myWeight to 0.0
• int value = 3*3;
• int value = GetValue();

Pointers

• A pointer holds the memory address of another object.
• Through the pointer we can indirectly manipulate the referenced object.
Pointers are useful for
• Creating linked data structures such as trees and lists,
• management of dynamically allocated objects, and
• as a function parameter type for passing large objects such as arrays.

• Every pointer has an associated type.
• The type of a pointer tells the compiler how to interpret the memory content at the referenced location and how many bytes this interpretation includes.

Examples of pointer definitions:
• int *pointer;
• int *pointer1, *pointer2;
• string *myString;

The dereference operator (*) dereferences a pointer variable so that we can manipulate the memory content at the location specified by the pointer.

The address-of operator (&) provides the memory address (a pointer) of a given object.

Example: Correct or incorrect?
int var1 = 333, var2 = 444, *pvar1, *pvar2;  // correct.
pvar1 = var1;                          // correct.
pvar2 = &var2;                        // correct.
*pvar1 = var2;                        // correct. int = int
*pvar2 = *pvar1 + 100;                // correct. int = int

Question: Considering only correct lines, what is the final value of var2?
Answer: var2 = 544
### Pointers

Notice that in pointer definitions the `*` symbol indicates the pointer type and is **not** the dereference operator.

**Example:**

```cpp
int var;
int *pvar1 = var;  // Incorrect!
```

During initialization a pointer can only be assigned an address:

```cpp
int var;
int *pvar1 = &var;  // Correct!
```

You can use **pointer arithmetic** to iterate through an array:

```cpp
int ia[10];
int *iter = &ia[0];
int *iter_end = &ia[10];
while (iter != iter_end)
{
    do_something_with_value(*iter);
    ++iter;
}
```

### References

- References (aliases) can be used as alternative names for objects.
- In most cases they are used as formal parameters to a function.
- A reference type is defined by following the type specifier with the `address-of operator`.

**Example:**

```cpp
int val1 = 333;
int &refVal1 = val1;
```

- A reference must be initialized.
- Once defined, a reference cannot be made to refer to another object.
- All operations on the reference are actually applied to the object to which the reference refers.

**Example:**

```cpp
int val1 = 333;
int &refVal1 = val1;
val1++;
refVal1 += 100;
cout << "Result: " << refVal1;
```

Result: 434

### The C++ string Type

To use the **C++ string type**, you must include its associated header file:

```cpp
#include <string>
```

Different ways to initialize strings:

```cpp
string myString("Hello folks!");
string myOtherString(myString);
string myFinalString;  // empty string
```

The length of a string is returned by its `size()` operation (without the terminating null character):

```cpp
cout << myString.size();  // 12
```

We can use the `empty()` operation to find out whether a string is empty:

```cpp
bool isEmptyString = myString.empty();
```

Use the equality operator to check whether two strings are equal:

```cpp
if (myString == myOtherString)
    cout << "Wow, the strings are equal."
else
    cout << "Nope, the strings are not equal."
```

Copy one string to another with the assignment operator:

```cpp
myFinalString = myOtherString;
```
The C++ string Type

Use the `plus operator` to concatenate strings:

```cpp
string s1 = "Wow! ", s2 = "Ouch! ";
const char *s3 = "Yuck! ";
s2 += s1 + s3 + s2;
cout << s2;
Ouch! Wow! Yuck! Ouch!
```

The const Qualifier

The `const type qualifier` transforms an object into a constant.

**Example:**

```cpp
const double pi = 3.1416;
```

- Constants allow you to store parameters in well-defined places in your code.
- Constants have an associated type.
- Constants must be initialized.
- Constants cannot be modified after their definition.
- Constants replace the `#define "technique"` in C.

Default Arguments

Notice that only *trailing* parameters of a function can have default values.

- This rule allows the compiler to know which arguments are defaulted when the function is called with fewer than its complete set of arguments.

**Examples:**

```cpp
void F1(int i, int j = 7); legal
void F2(int i = 3, int j); illegal
void F3(int i, int j = 3, int k = 7); legal
void F4(int i = 1, int j = 2, int k = 3); legal
void F5(int i, int j = 2, int k); illegal
```

Functions as Arguments

- Functions in C++ can be thought of as the addresses of the compiled code residing in memory.
- Functions are therefore a form of pointer.
- Functions can be passed as a pointer-value argument into another function.

```cpp
double F(double x)
{
    return (x*x + 1.0/x);
}

void plot(double Fcn(double), double x0, double incr, int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << "x: " << x0 << " f(x): " << Fcn(x0) << endl;
        x0 += incr;
    }
}

int main()
{
    plot(F, 0.01, 0.01, 100);
    return 0;
}
```

Overloading Functions

- Overloading refers to using the same name for multiple meanings of an operator or a function.

**Example:**

```cpp
double Average(const int a[], int size)
{
    int sum = 0;
    for (int i = 0; i < size; ++i)
    {
        sum += a[i];
    }
    return static_cast<double>(sum) / size;
}

double Average(const double a[], int size)
{
    double sum = 0.0;
    for (int i = 0; i < size; ++i)
    {
        sum += a[i];
    }
    return (sum / size);
}
```
Overloading Functions
The following code shows how Average() is invoked:

```cpp
ing main()
{
    int a[5] = {1, 2, 3, 4, 5};
    double b[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
    cout << Average(a, 5) << " int average" << endl;
    cout << Average(b, 5) << " double average" << endl;
    return 0;
}
```

Inlining
- C++ provides the keyword `inline` to preface a function declaration.
- For functions declared as inline, the compiler will generate inline code instead of function calls.
- Inline functions lead to increased time efficiency and decreased memory efficiency.

**Question:** How about recursive functions?
**Answer:** There are no recursive inline functions because they would require infinite memory.

Inlining has a similar effect as macro expansion:

**Inlining example:**
```cpp
inline double cube(double x)
{
    return (x * x * x);
}
```

**Macro example:**
```cpp
#define CUBE(X)   ((X)*(X)*(X))
```

So what is the difference between inlining and macro expansion?
**Answer:** Macro expansion provides no type safety as is given by the C++ parameter-passing mechanism. Therefore, it is advisable to use inlining instead of macro expansion.

Enumeration Types
Sometimes you may want to define for your object a set of states or actions.
For example, you could define the following states for a Student Admonisher Robot:
- observeStudent
- shoutAtStudent
- followStudent
- rechargeBattery

Using the const qualifier, you could define the following constants:
- `const int observeStudent = 1;`
- `const int shoutAtStudent = 2;`
- `const int followStudent = 3;`
- `const int rechargeBattery = 4;`

A function `SetRobotState` could then be defined as follows:
```cpp
bool SetRobotState(int newState)
{
    ... 
    int currentState = newState;
    return executionSuccessful;
}
```
Enumeration Types

February 26, 2019 CS410 – Software Engineering
Lecture #3: C++ Basics I

However, mapping states onto integers has certain disadvantages:

• You cannot restrict the range of values that are passed to SetRobotState.
• There is no useful typing – if you define individual sets of states for multiple objects, each object could formally be set to any of these states, not only its individual ones.

This problem can be solved with enumeration types.

Enumeration Types

February 26, 2019 CS410 – Software Engineering
Lecture #3: C++ Basics I

Enumeration types can be defined as follows:

```cpp
enum robotState {observeStudent = 1,
                 shoutAtStudent, followStudent, rechargeBattery};
```

This way we defined a new type `robotState` that can only assume four different values. These values still correspond to integers. For example, `cout << followStudent;` gives you the output ‘3’.

Enumeration Types

February 26, 2019 CS410 – Software Engineering
Lecture #3: C++ Basics I

However, we are now able to restrict the values that are passed to SetRobotState to the four legal ones:

```cpp
bool SetRobotState(robotState newState)
{
    ... robotState currentState = newState;
    return executionSuccessful;
}
```

Any attempt to call SetRobotState with an integer value or a value of a different enumeration type will cause an error at compile time.