FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 3

Types

Literal Constants

- **Integer Literals**
  - What means 20, 024, 0x14?
  - unsigned: 128u, long 1L (assign a negative number to unsigned!)
- **Floating-Point Literals**
  - 0. , 0e0 , .001f, 1E-3F (type of constants 1 and 1.0?)
- **Boolean and Literals**
  - Bool: true, false
  - Character: ´a´
  - string: “a“

Definitions

literal constant: A value such as a number, a character, or a string of characters.

- value cannot be changed
- Literal characters are enclosed in single quotes, literal strings in double quotes.

Escape sequences

escape sequence: Alternative mechanism for representing characters.

- Usually used to represent nonprintable characters such as newline or tab.
- An escape sequence is a backslash followed by a character, a three-digit octal number, or a hexadecimal number.
- Escape sequences can be used as a literal character (enclosed in single quotes) or as part of a literal string (enclosed in double quotes).
- Examples: \n, \t, \\, \b

Variables

Object: A region of memory that has a type. A variable is an object that has a name.

Declaration: Asserts the existence of a variable, function, or type defined elsewhere in the program.

- Some declarations are also definitions; only definitions allocate storage for variables.

Definition: Allocates storage for a variable of a specified type and optionally initializes the variable.

Names may not be used until they are defined or declared!

Variables

type specifier: Part of a definition or declaration that names the type of the variables that follow.

Identifier: A name.

- A nonempty sequence of letters, digits, and underscores that must not begin with a digit.
- Identifiers are case-sensitive: Upper- and lowercase letters are distinct.
- Identifiers may not use C++ keywords.

Type-checking

statically typed: Term used to refer to languages such as C++ that do compile-time type checking.

C++ verifies at compile-time that the types used in expressions are capable of performing the operations required by the expression.

type-checking: Process by which the compiler verifies that the way objects of a given type are used is consistent with the definition of that type.

Variable Initialization

variable initialization: Rules for initializing variables and array elements when no explicit initializer is given.

- For class types, objects are initialized by running the class's default constructor. If there is no default constructor, then there is a compile-time error: The object must be given an explicit initializer.
- For built-in types, initialization depends on scope. Objects defined at global scope are initialized to 0; those defined at local scope are uninitialized and have undefined values.

Scope


```
std::string s1 = "hello";

int main()
{
	std::string s2 = "world";

	std::cout << s1 << " " << s2 << std::endl;

	int s1 = 42;

	std::cout << s1 << " " <<  s2 << std::endl;
	return 0;
}
```

Scope

- **Scope**: A portion of a program in which names have meaning. C++ has several levels of scope:
  - global— names defined outside any other scope.
  - class— names defined by a class.
  - namespace— names defined within a namespace.
  - local— names defined within a function.
  - block— names defined within a block of statements, that is, within a pair of curly braces.
  - statement— names defined within the condition of a statement, such as an if, for, or while.
  - Scopes nest. For example, names declared at global scope are accessible in function and statement scope.

References

Reference: An alias for another object.

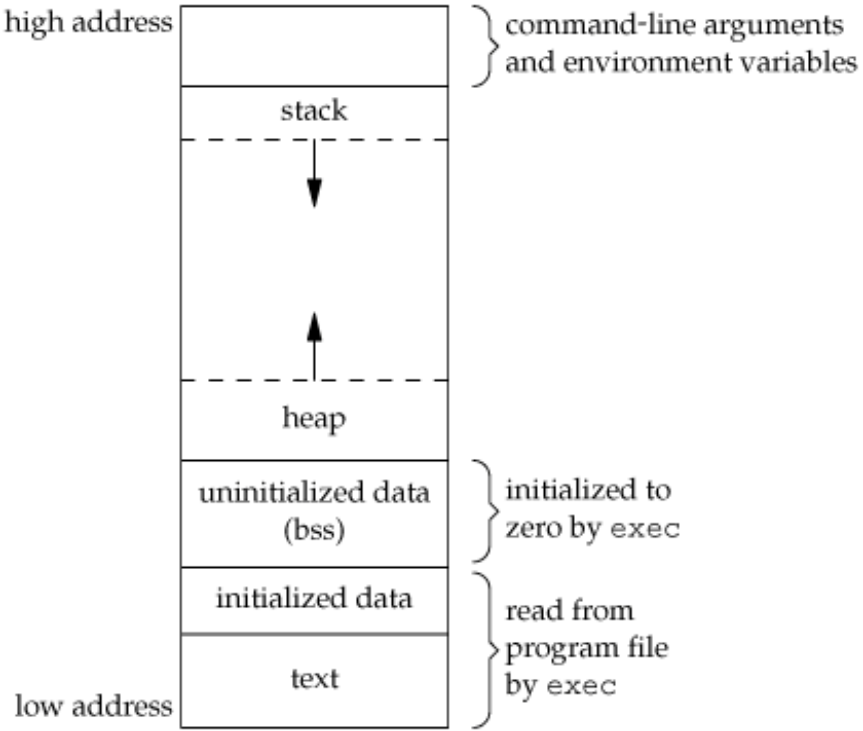
Defined as follows: type &id = object;

- Defines id to be another name for object. Any operation on id is translated as an operation on object.
- There is no way to rebind a reference to a different object
- Nonconst reference may be attached only to an object of the same type as the reference itself

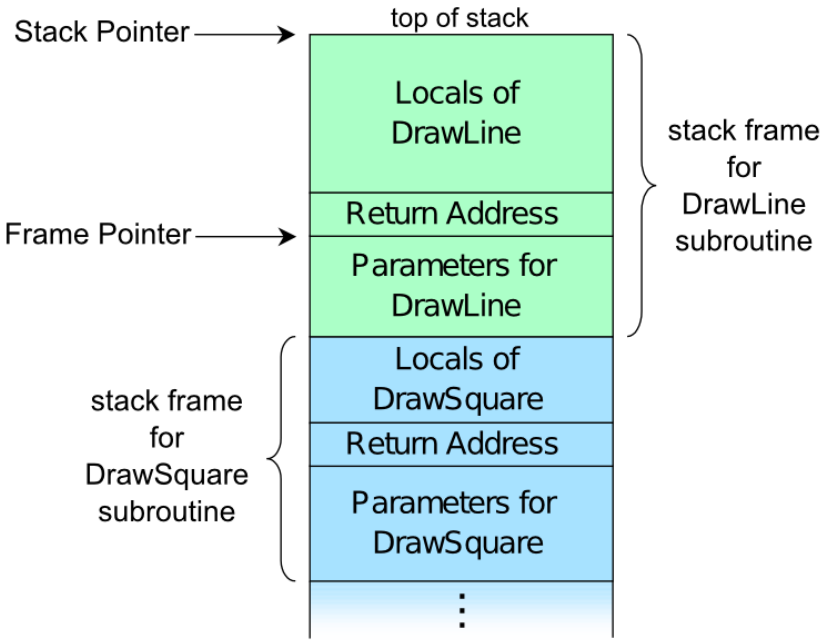
const reference: A reference that may be bound to a const object, a nonconst object, or the result of an expression.

- A const reference may not change the object to which it refers

Memory region



Stack


Stack Pointer
top of stack
Locals of DrawLine
Return Address
Frame Pointer
Parameters for DrawLine
stack frame for DrawLine subroutine
Locals of DrawSquare
Return Address
Parameters for DrawSquare
stack frame for DrawSquare subroutine

Pointer

Address	Content	Name	Type	Value
90000000	00			
90000001	00	anInt	int	000000FF(255_{10})
90000002	00			
90000003	FF			
90000004	FF	aShort	short	FFFF(-1_{10})
90000005	FF			
90000006	1F			
90000007	FF			
90000008	FF			
90000009	FF	aDouble	double	1FFFFFFFFFFFFFFF
9000000A	FF			($4.4501477170144023E-308_{10}$)
9000000B	FF			
9000000C	FF			
9000000D	FF			

- memory: linear address space
- C/C++ gives you the power (responsibility!) to access arbitrary memory regions
- OS prevents access to certain memory regions (SEGFAULT)
- Pointer store memory addresses

Pointer

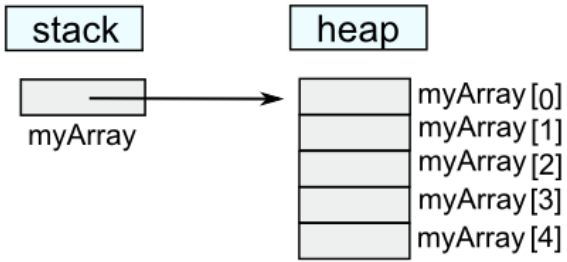
Address	Content	Name	Type	Value
90000000	00	anInt	int	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	aShort	short	FFFF(-1_{10})
90000005	FF			
90000006	1F	aDouble	double	1FFFFFFFFFFFFFF ($4.4501477170144023E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrAnInt	int*	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

- memory: linear address space
- C/C++ gives you the power (responsibility!) to access arbitrary memory regions
- OS prevents access to certain memory regions (SEGFAULT)
- Pointer store memory addresses

Heap

```
int * myArray = new int[5];
```



Memory Leaks

- Memory which is no longer needed is not released.

```
int main( int argc, char**argv)
{
    int * arr = 0;
    arr = new int[20]; //Leak!!

    return 0;
}


int main( int argc, char**argv)
{
    int * arr = 0;
    arr = new int[20];

    delete [] arr;

    return 0;
}
```

Pointers I

Pointer: An object that holds the address of an object.

Example: int * p;

Values used to initialize or assign to a pointer:

- A constant expression with value 0 or better nullptr
- An address of an object of an appropriate type
- The address one past the end of another object
- Another valid pointer of the same type

Pointers II

- Pointers are iterators for arrays
- **void*:** A pointer type that can point to any nonconst type.
  - Only limited operations are permitted on void* pointers:
  - They can be passed or returned from functions and they can be compared with other pointers.
  - They may not be dereferenced.

Operators

*** operator:** Dereferencing a pointer yields the object to which the pointer points.

Assigning to the result of a dereference assigns a new value to the underlying object.

Example: int * p; *p = 2;

& operator: The address-of operator.

Yields the address in memory to which it is applied.

Pointers vs. references


```
int i = 1, j = 2;
int *pi = &i, *pj = &j;
pi = pj;

int &ri = i, &rj = j;
ri = rj;
```

- Comparing pointers and references
  - References always refer to an object
  - Assigning to a reference changes the underlying object

Pointers and the const qualifier


```
const double* cptr;
*cptr = 42;

int ierr = 0;
int *const curErr = &ierr;
curErr = curErr;

const double pi = 3.14;
const double* const pi_ptr = &pi;
```

- Pointers to const objects
  - Pointers that think they are const
- const pointers
- const pointers to const objects

Typedef

Typedef: Introduces a synonym for some other type.

Form: *typedef type synonym;* defines synonym as another name for the type named type.

Alternative (C++11): using synonym = type;

Purposes:

- Hide implementation of a given type and emphasize instead the purpose for which the type is used
- Streamline complex type definitions, making them easier to understand
- Allow a single type to be used for more than one purpose while making the purpose clear each time the type is used

decltype

- Type specifier that deduces the type of a variable or an expression
- Example:
  - const int ci = 0;
  - decltype(ci) x = 0; // x has type const int

auto

- Type specifier that deduces the type of a variable from its initializer
- Example:
  - auto i = 10; // i is an int

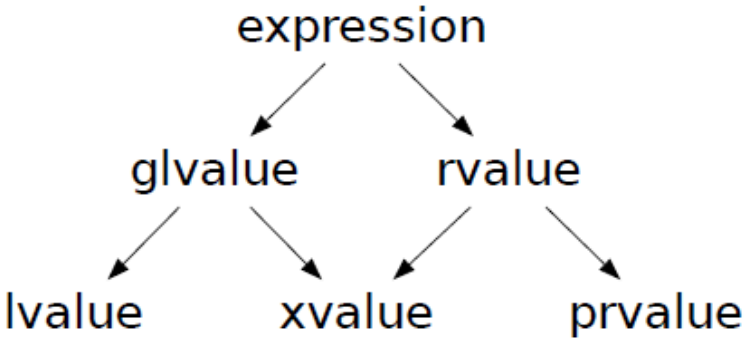
Moving objects

- One of the major features in the new standard is the ability to move rather than copy an object!
- Moving instead of copying can provide a significant performance boost
- To support move operations a new kind of reference is introduced, an rvalue reference
- Example
  - int i = 42;
  - int &&rr = i * 42;

Rvalue reference

- Rvalue references refer to objects that are about to be destroyed
- While lvalues have persistent state, rvalues are either literals or temporary objects created in the course of evaluating expressions
- Note that variables are lvalues!

Expression category taxonomy



Expression category I

lvalue: An expression that yields an object or function.

A nonconst lvalue that denotes an object may be the left-hand operand of assignment.

xvalue: An xvalue (an “eXpiring” value) refers to an object, usually near the end of its lifetime (so that its resources may be moved).

Certain kinds of expressions involving rvalue references yield xvalues.

prvalue: A “pure” rvalue that is not an xvalue

glvalue: A “generalized” lvalue

rvalue: An expression that yields a value but not the associated location, if any, of that value, an xvalue or a temporary object.

Expression category II


```
int i;
i = 1;
```
lvalue

```
int* ip;
*ip = 1;
```
lvalue

```
int i;
i = 1;
```
prvalue

```
float& f() {
 float f;
 return f;
}
```
lvalue

```
float f() {
 return 2;
}
```
prvalue

```
float&& f() {
 return 2;
}
```
xvalue