**Compilation**

Compilers

Compilers translate the source code of modules into binary code that is assembled to form a single executable version of program.  The source code of a module is written in a specific language.  Compilers are language specific programs.

Many languages are defined by an international standard.  A language standard is an agreement amongst compiler writers, consultants, application programmers, and academics to implement a well-defined, minimal subset of the language used by the programming community at large.  Coding to a language standard maximizes source code *portability*: a fully portable program compiles and runs without modification in a switch from one compiler to another.

Type System

Compilers use the type system of their programming language to translate source code into binary code.  The type system provides consistency across the language's building blocks and identifies the operations that are admissible in forming expressions.  The type system enables the compiler to check whether or not the relations between the building blocks are well-formed.  The type system describes how to interpret the bit strings in memory.  Source code that breaks the type system exposes the underlying bit strings in memory and thereby introduces uncertainty.  In such cases how the compiler interprets the contents of memory is undefined.

The admissible operations on fundamental and built-in types are defined as part of the core of a language.  The admissible operations on the user-defined types of an object-oriented language are specified by the programmer in the definition of those types.  The compiler reports errors where the relations between these types are ill-formed.

Compiling, Linking and Executing

Source code may include some instructions that depend on information that is only available during the execution of the binary code.  Programming such source code involves distinguishing times at which to check for compliance with the type system.

Verification of type compliance can occur at

* compile-time - during the compiler's translation of a module's source code into binary code
* link-time - during the linker's assembly of the binary code components for the modules that constitute an application, or
* run-time - during the user's execution of the binary executable.

Statically and Dynamically Allocated

In these notes, the terms *statically* and *dynamically* distinguish what can be determined at compile-time from what needs to be determined at run-time.

The term *statically* refers to anything that the compiler itself can determine for the module being translated without any link-time or run-time information.  As programmers, we seek to translate and type-check as much of the source code as possible at compile-time.  We refer to this as static type-checking and refer to a language that performs type-checking at compile-time as a statically typed language.

The term *dynamically* refers to anything that is determined during execution.  As programmers, we refer to type-checking of dynamically allocated memory as dynamic type-checking and refer to a langauge that performs type-checking at run-time as a dynamically typed language.

Memory Distinctions

When an operating system loads an executable program into memory, it places the different parts of the program into different segments:

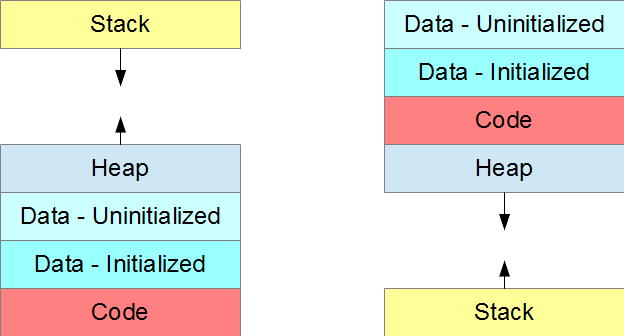
* code segment - stores the program instructions
* data segments - store data that survives the lifetime of the program
* stack segment - stores local data that is, statically allocated
* heap segment - stores local data that is dynamically allocated

The size of *statically allocated memory* is known at compile-time.  The compiler inserts the code to allocate this memory on the *stack* or *data* segment.  Statically allocated memory is fast and deallocates automatically but cannot grow or shrink in size.

The size of *dynamically allocated memory* may be of variable size.  This memory is allocated at run-time.  The operating system provides this memory to the executable program in the size requested at that time.  This memory is allocated on the *heap*, managed by the freestore manager and needs to be returned to the manager before it can be reused by the executable for some other purpose.

Most systems allocate heap and stack memory for an executable next to one another.  In fact, on most systems, heap and stack memory expand in one another's direction as required.

Memory organization varies between operating systems and compilers.  Possible organizations are shown below.

  
  
Allocation of Memory for an Application by the Operating System

Linkage

External Linkage

A name with *external linkage* refers to an entity that is declared in a different scope within another translation unit.  The C++ keyword extern identifies external linkage.

|  |  |
| --- | --- |
| extern int share\_me; // declaration |  |

We omit this linkage keyword in the translation unit that defines and initialize the named entity:

|  |
| --- |
| int share\_me = 0; // definition |

C++ ignores the extern keyword if an initialization is present.

In the following program the name share\_me refers to the same entity across two translation units.  The variable share\_me is accessed in Module\_a.cpp and defined in Module\_b.cpp.

|  |  |
| --- | --- |
| // External Linkage  // Module\_a.cpp  #include <iostream>  extern int share\_me; // external linkage declaration  void display() {  std::cout << "share\_me at " << &share\_me << '\n';  std::cout << "share\_me is " << share\_me++ << '\n';  } | cl Compiler  share\_me at 00FE4A24  share\_me is 0  share\_me at 00FE4A24  share\_me 1  share is at 00FE4A24  share is 2 |
| // External Linkage  // Module\_b.cpp  #include <iostream>  void display();  int share\_me = 0; // variable definition  int main() {  display();  display();  std::cout << "share\_me at " << &share\_me << '\n';  std::cout << "share\_me is " << share\_me++ << '\n';  } |

Internal Linkage

A name with *internal linkage* refers to an entity that is invisible outside its own translation unit, but visible to other scopes within its translation unit.  The C++ keyword static identifies internal linkage.

|  |
| --- |
| static int local = 2; |

The following program allocates separate memories for the variables named local in Module\_a.cpp and Module\_b.cpp.  The same name (local) refers to two distinct variables in the two translation units.

|  |  |
| --- | --- |
| // Internal Linkage  // Module\_a.cpp  #include <iostream>  static int local = 4;  void display() {  std::cout << "local is at " << &local << '\n';  std::cout << "local is " << local++ << '\n';  } | cl Compiler  local is at 010D3000  local is 4  local is at 010D3000  local is 5  local is at 010D323C  local is 2 |
| // Internal Linkage  // Module\_b.cpp  #include <iostream>  void display();  static int local = 2;  int main() {  display();  display();  std::cout << "local is at " << &local << '\n';  std::cout << "local is " << local++ << '\n';  } |

Note the different addresses for the same name in the two translation units.

Type

A declaration associates the name of an entity with a type.  A C++ *type* describes how to interpret the bit string in memory associated with the name and identifies the operations that are admissible on that type.  The type may be fundamental, built-in or user-defined.

A C++ type may exhibit specific storage properties.  We identify these properties using *cv-qualifiers*.

cv Qualifiers

The storage properties of a type differ with respect to mutability and side-effects.  The *cv-qualifiers* include:

* none - the type is modifiable and not subject to any side-effects
* const - the type is unmodifiable and not subject to any side-effects
* volatile - the type is modifiable and subject to some side-effects
* const volatile - the type is unmodifiable and subject to some side-effects

A value stored in a const type cannot be changed.  Including this keyword in the entity's declaration informs the compiler that it should reject any code that attempts to modify the value of the object.

A value stored in a volatile type is subject to side-effects and needs its memory location updated every single time that the CPU changes that value.  This qualification is important in hardware-related code that keeps the value in register memory as the CPU updates that value.  Declaring a type as volatile instructs the compiler to retain all intermediate instructions and avoid optimizations that assume no side-effects.

A type without any qualifier is *cv-unqualified*.

Type Definition

A type definition is an alias for a type.  A type definition can simplify code readability for compound types.

The C++ keyword typedef identifies a synonym for a specified type.  A type definition takes the form

|  |
| --- |
| typedef compoundType Synonym; |

compoundType is the original type along with all of the type specifiers.  Synonym is the alias for compoundType.

For example,

|  |
| --- |
| typedef const int constInt; // defines the const int type as a constInt |

We can use the alias to declare a variable of the compound type:

|  |
| --- |
| constInt myConstant; // myConstant is a const int |

All type specifiers must be included in the type definition.  We cannot insert new specifiers when we invoke the synonym type.

A typedef cannot include a linkage specifier.